串口&以太网转 CAN 通信 SDK 说明文档

目录

1.版本信息

版本	日期	修改内容
V1.0.0	2018-04-17	第一版本

2.目的

本文档为 INNFOS 执行器 SDK 说明文档，用于执行器二次开发，编写查看、控制、调节执行器的应用。

3.目录结构

- 主目录 serialPort&Ethernet2CAN_sdk_vx.x.x，其中 x.x.x 为当前的 sdk 版本号
- …\example 为示例程序，…\example\src 为示例程序源码
- …\sdk 为 SDK 相关的头文件和库文件，其中…\sdk\include 包含了 SDK 需要的头文件，…\sdk\lib 包含了 windows64 位系统和 linux 64 位系统的库文件
- …\tools 包含了 windows 下用到的 vs2015 64 位版本运行时库
- …\ readme.txt 包含了一些使用 SDK 需要注意的事项

4.执行器连接

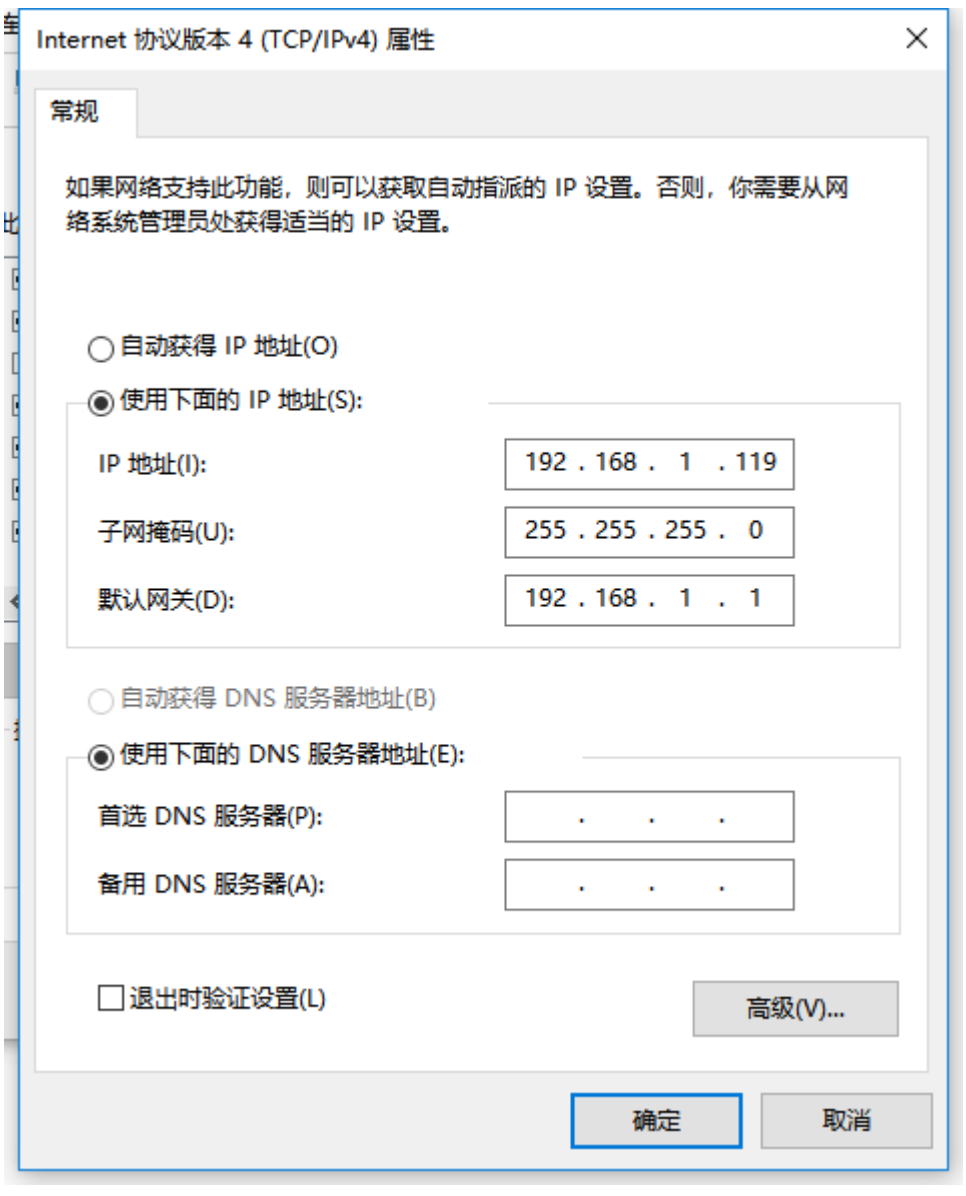
5.环境配置和示例代码编译运行

5.1windows 平台

环境配置

1.SDK 需要 win7 sp1 以上的 64 位 windows 操作系统
2.运行安装 tools 文件夹中的 vcredist_x64.exe，安装 vs2015 运行时库
3.安装 cmake：在 Cmake 官网 https://cmake.org/download/下载最新版本 Cmake 安装
4.ip 地址配置，打开控制面板，选择网络和 Internet,再选择网络和共享中心，再选择更改适配器设置，右键单击以太网，选择属性

选择 TCP/IPv4,然后选择属性，配置如下图：

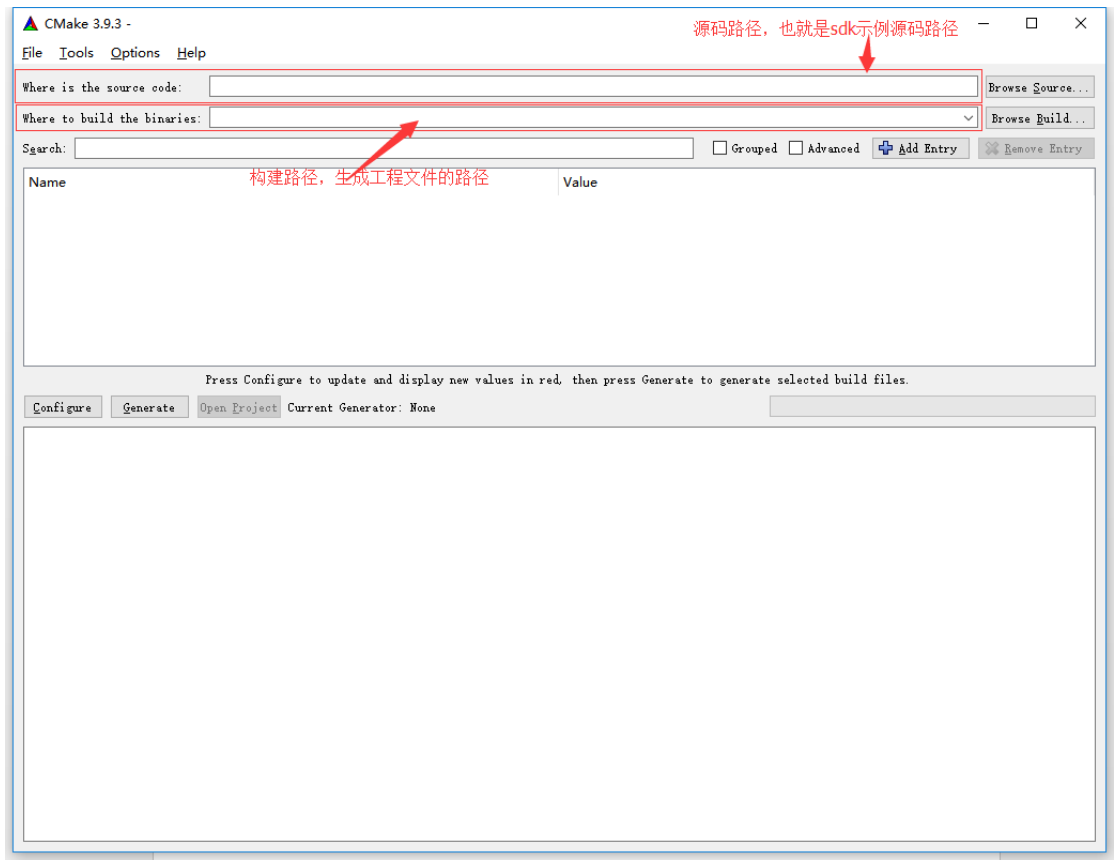


其中 ip 地址中的 192.168.1.119 中的 119 可以替换成 100~200 之间的任意整数，配置完成点击确定

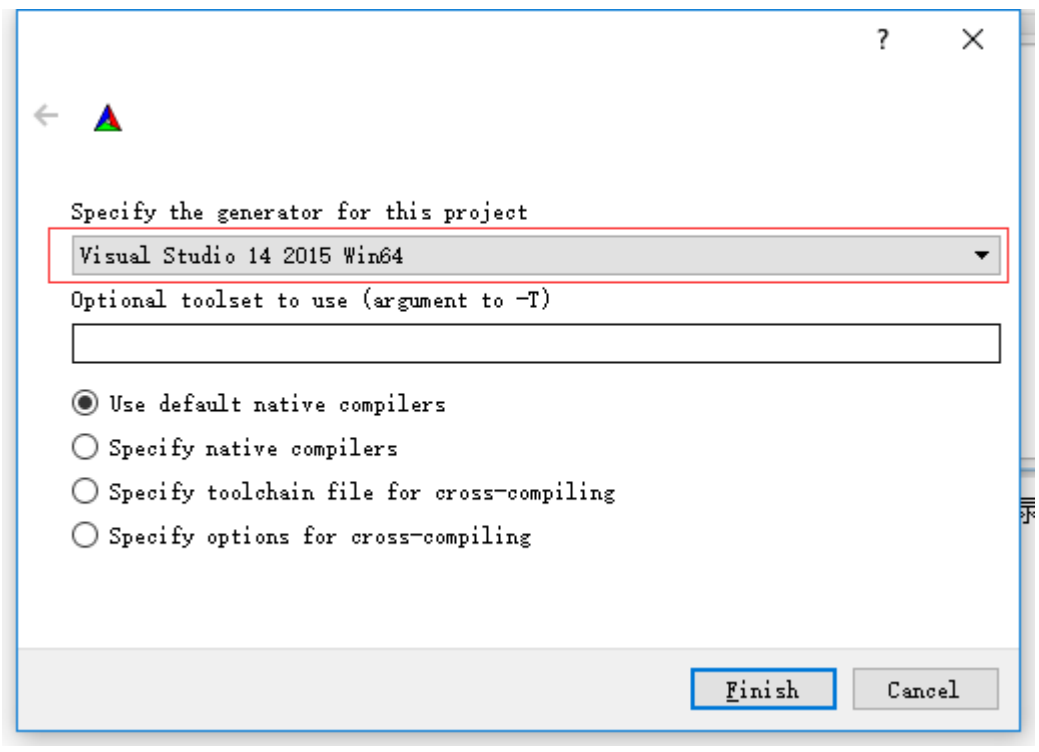
SDK 编译

运行 cmake-gui 

出现如下界面：



其中源码路径就是目录结构中的…\example 所在的路径，该目录下包含了 CMakeLists.txt 文件；
构建路径可自行定义，用于生成工程文件
两个路径配置完成后点击 Generate 按钮弹出如下界面



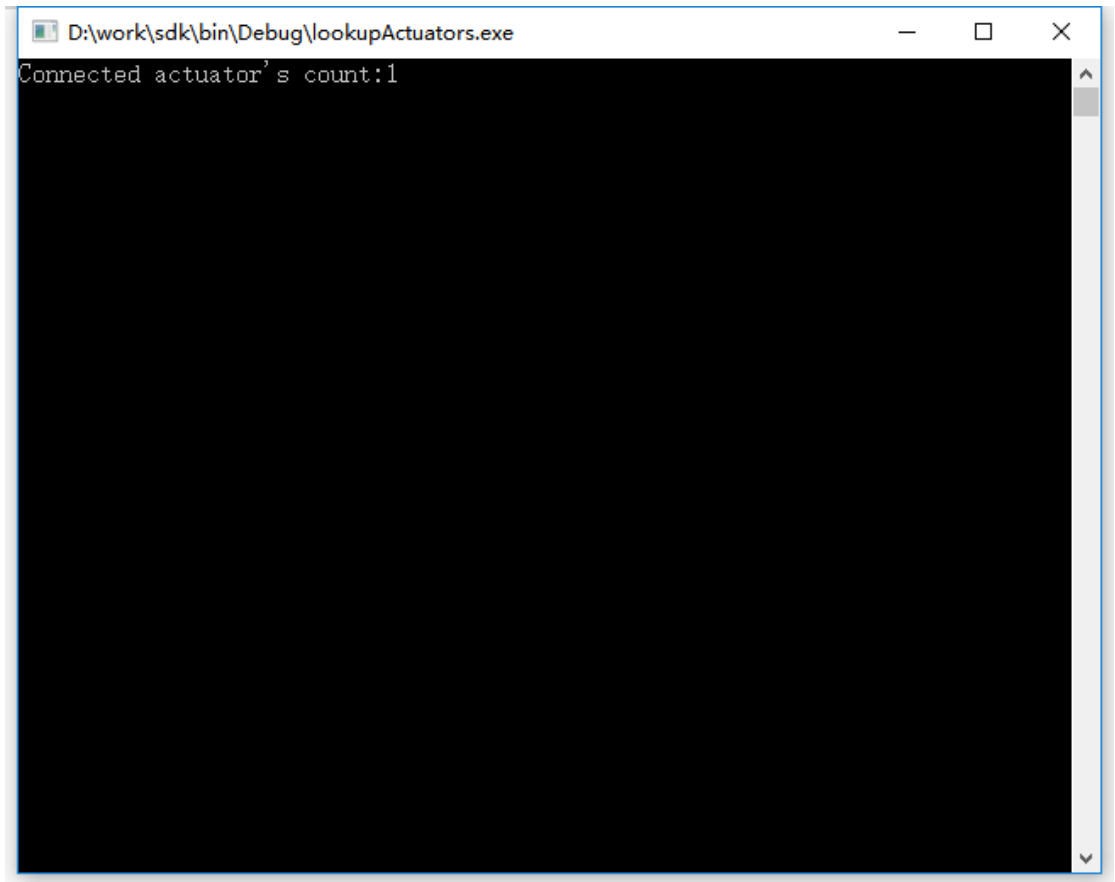
如果红色框内不是 64 位生成器，点击下拉三角，选择 64 位生成器，然后点击 Finish 按钮，生成成功后就生成了 Visual Studio 的工程文件，可用 Visual Studio 打开编译。编译完整个工程，在工程目录下会生成一个 bin 目录，里面有 Debug 或者 Release 文件夹（对应于编译的版本），将目录结构中的…\sdk\lib\windows_x64\debug 或…\sdk\lib\windows_x64\release 中的文件复制到对应版本的 bin 下面的 Debug 或者 Release 目录中，双击该目录中的 exe 就可正常运行示例程序了。

示例程序测试

确认执行器正确连接并供电以后，执行器会有黄色指示灯闪烁，此时可以测试示例代码。

1. 查找已连接的执行器

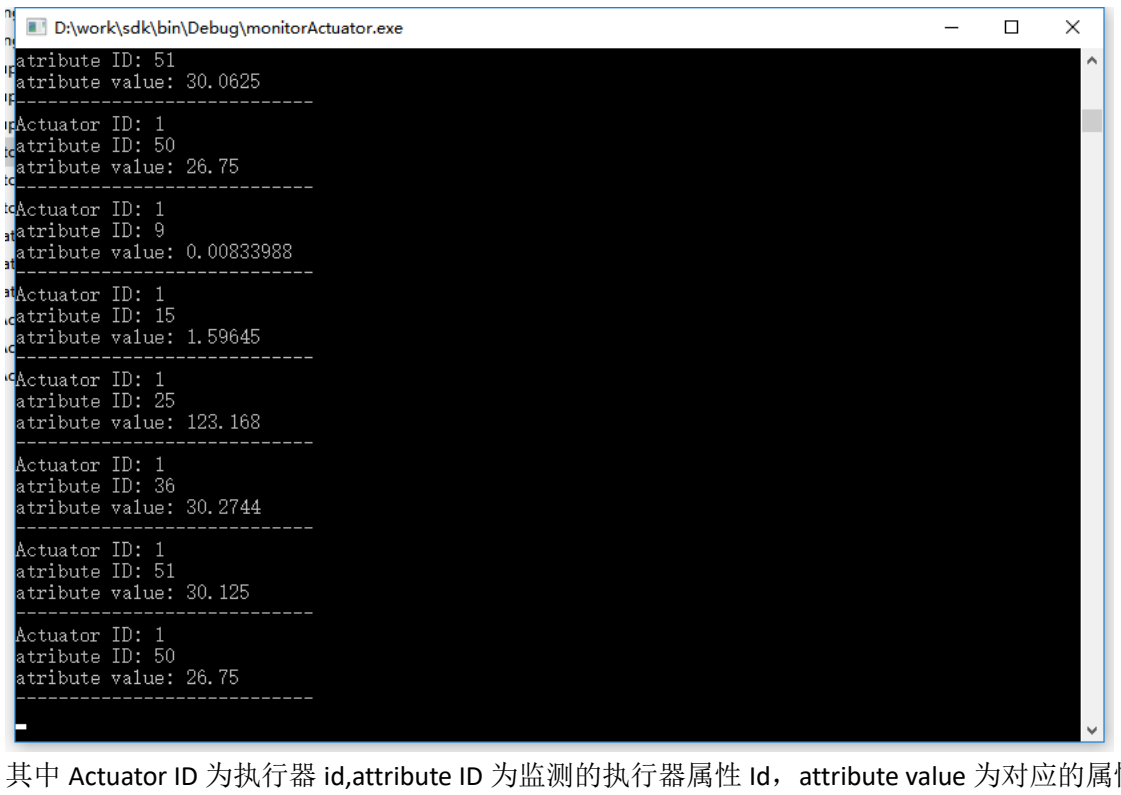
双击 bin 目录下的 lookupActuators.exe，弹出 cmd 窗口



此窗口会显示当前已连接的执行器数量，可以 ctrl+c 结束程序

2. 监测执行器状态

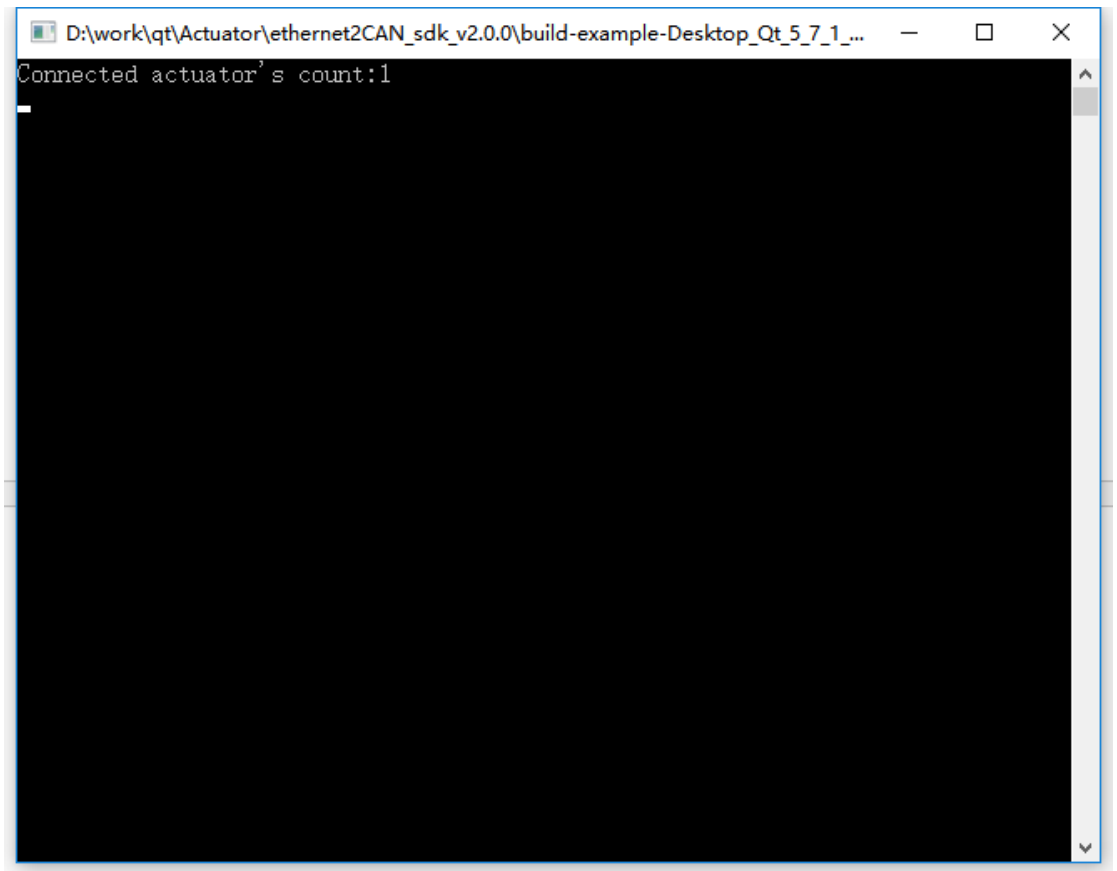
双击 monitorActuator.exe，弹出 cmd 窗口



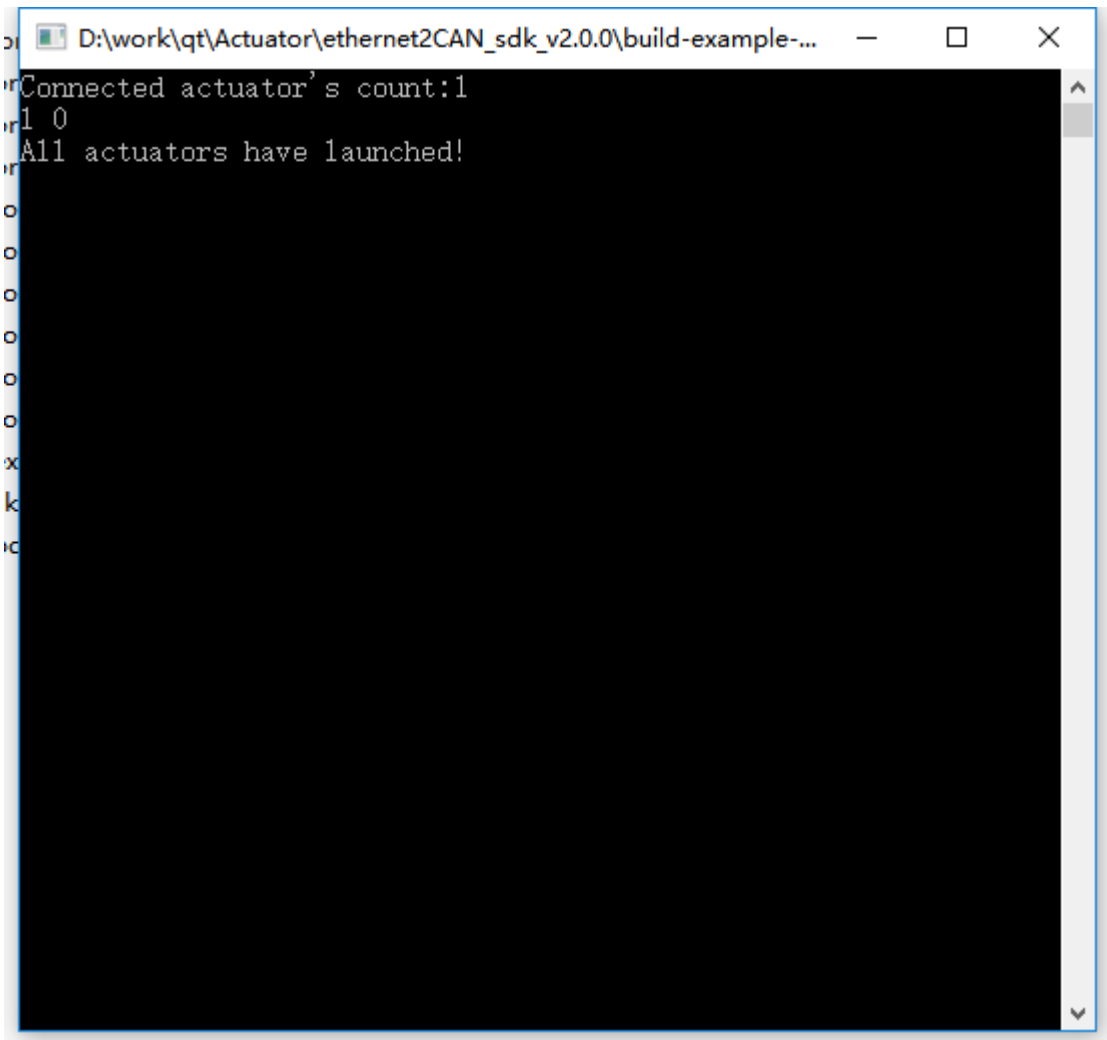
其中 Actuator ID 为执行器 id,attribute ID 为监测的执行器属性 Id，attribute value 为对应的属性值，可以 ctrl+c 结束程序

3. 控制执行器

双击 operateActuator.exe，弹出 cmd 窗口



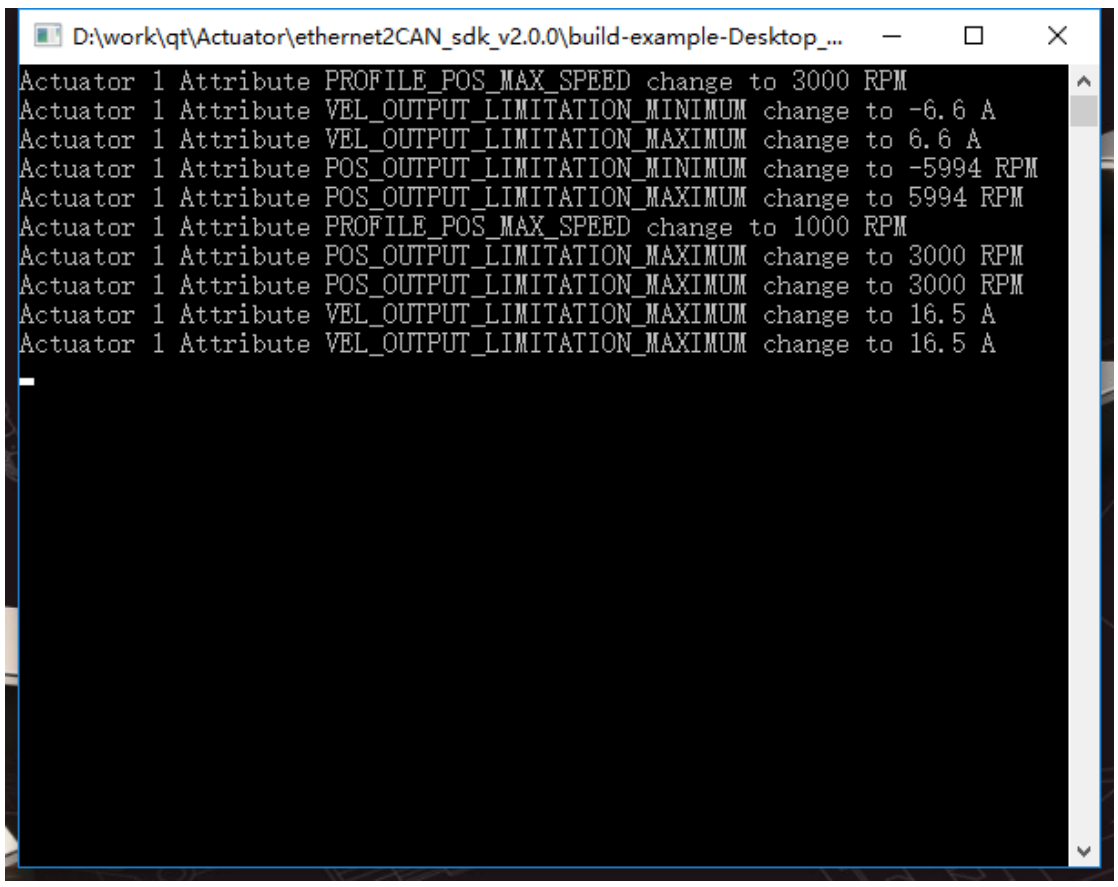
表示执行器已经找到，输入命令 l 0，该命令会启动所有已连接的执行器，如果启动成功，执行器会有绿色指示灯闪烁，表示已经启动成功，cmd 窗口如下显示



此时可激活执行器对应模式，比如输入 a 6 可以激活 profile position 模式，再输入 p 10，执行器会转动到 10 圈的位置；输入 a 7 可以激活 profile velocity 模式，再输入 v 500，执行器将以 500RPM 的速度转动，停止转动输入 v 0,；输入 a 1 可以激活电流模式，再输入 c 0.6，执行器将以恒定 0.6A 的电流转动（如果执行器不动，可用手轻轻转动一下执行器），可以 ctrl+c 结束程序

4. 控制器参数调整

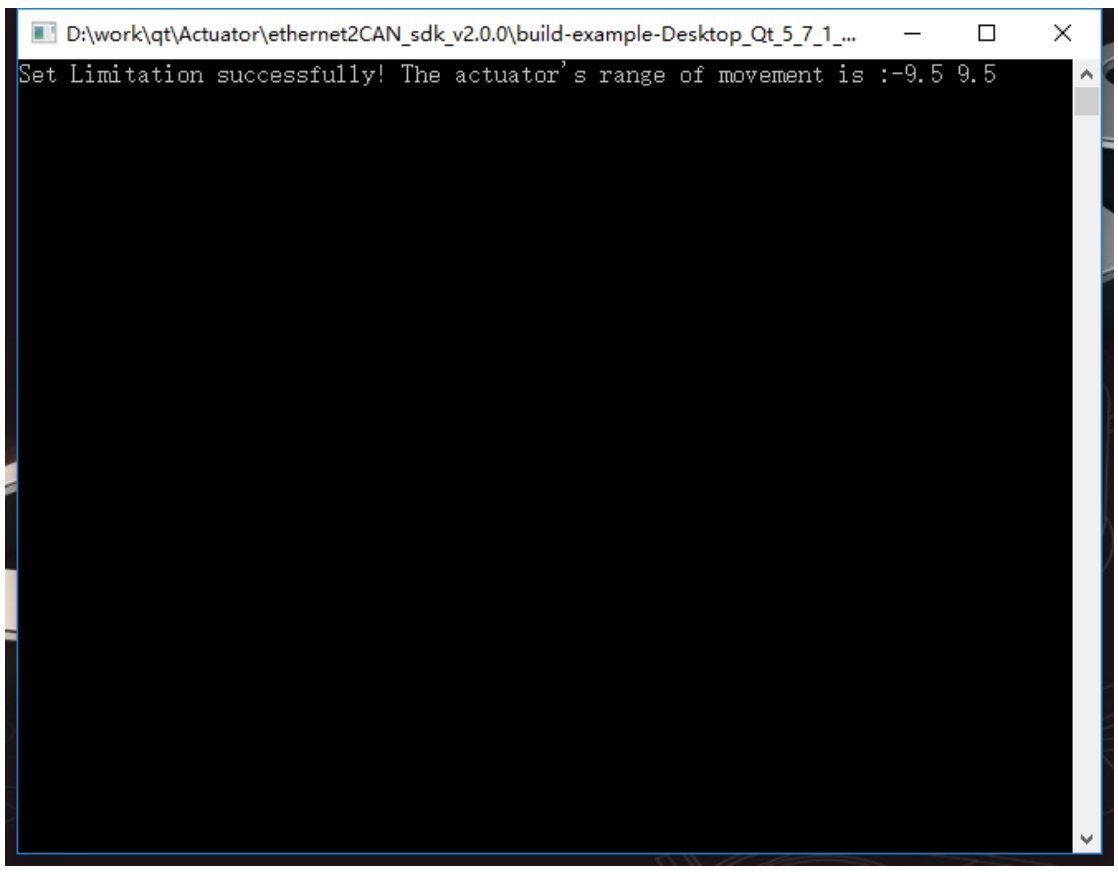
双击 tuneActuator.exe,弹出 cmd 窗口



此示例程序自动启动执行器并将位置环输出设置为 3000RPM,速度环的电流最大输出为 16.5A,如果使用 profile position 模式转动执行器，执行器的最大速度不会超过 3000RPM;如果使用 profile velocity 模式转动执行器，执行器最大电流不会超过 16.5A，可以 ctrl+c 结束程序

5. 执行器归零

双击 homingActuator.exe，弹出 cmd 窗口

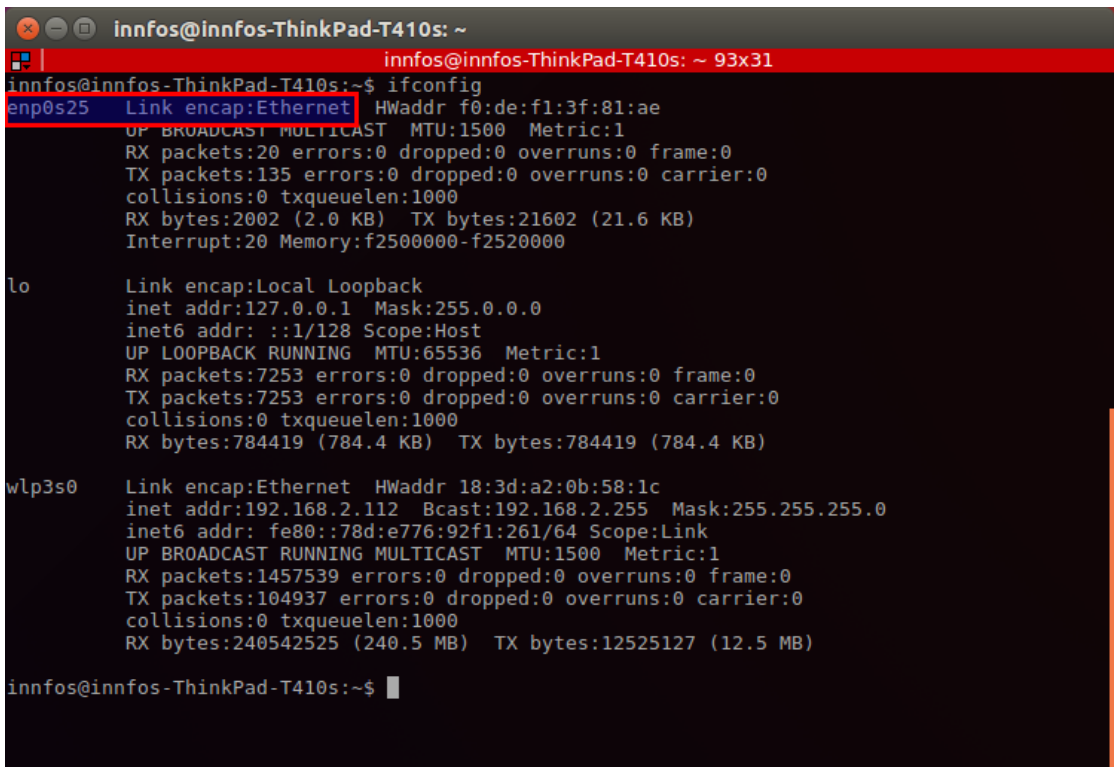


表示已经将执行器当前位置设置为零位，范围是-9.5R 到 9.5R，并且开启了位置限制，如果 profile position 模式下，输入此范围之外的位置，执行器不会转动，可以 ctrl+c 结束程序

5.2 linux 平台

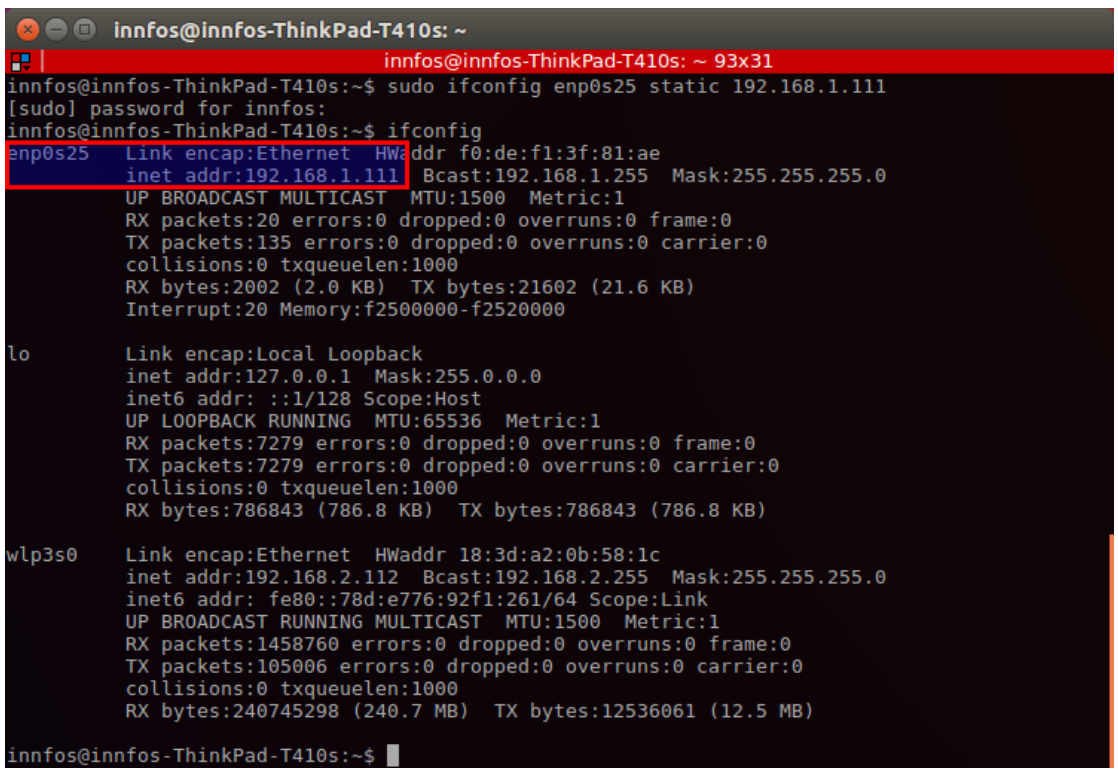
环境配置

1.本文档使用的是 ubuntu16.04 LTS 系统。
2.cmake 安装：打开终端输入命令 sudo apt-get install cmake
3.ip 地址配置：打开终端输入 ifconfig，查看网络配置



示例中有线网卡的名字是 enp0s25,输入命令 sudo ifconfig enp0s25 static 192.168.1.111

配置完成后输入 ifconfig，可看到配置成功后的 ip 地址



SDK 编译

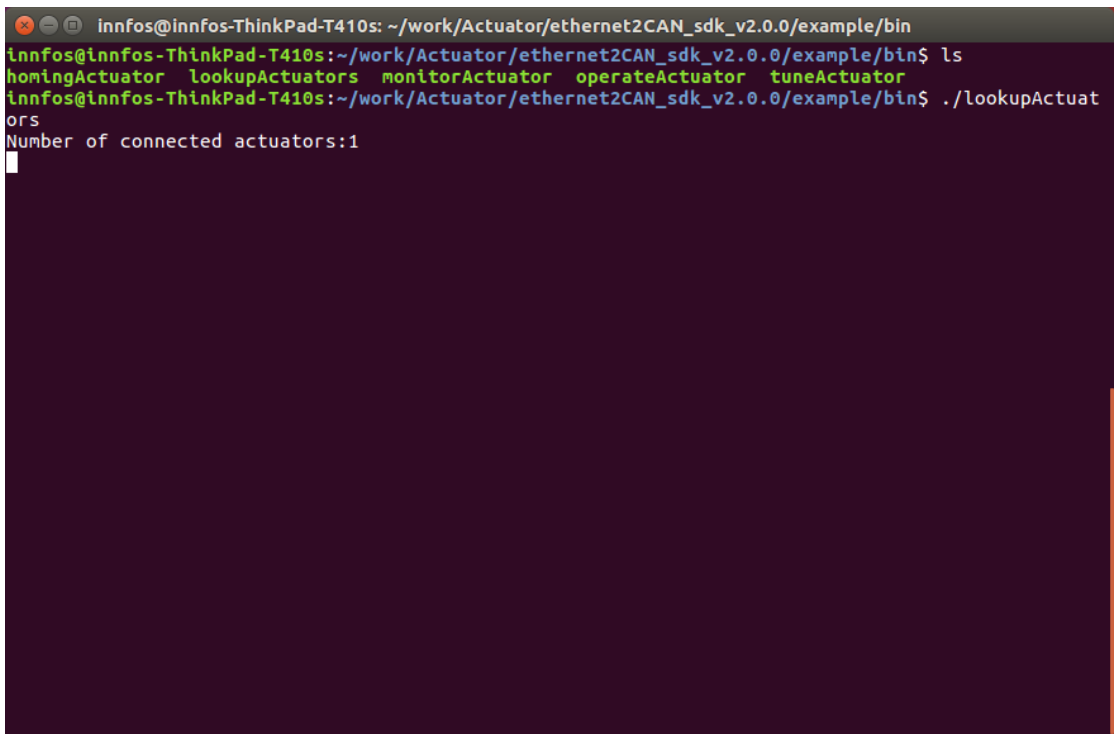
打开终端进入...\example 目录，该目录下有 CMakeLists.txt，输入命令 cmake CMakeLists.txt，执行成功后，再输入命令 make，执行完成后，在该目录下会生成一个 bin 文件夹，该目录存放了生成的示例程序。

确认执行器正确连接并供电以后，执行器会有黄色指示灯闪烁，此时可以测试示例代码。

示例程序测试

1. 查找已连接的执行器

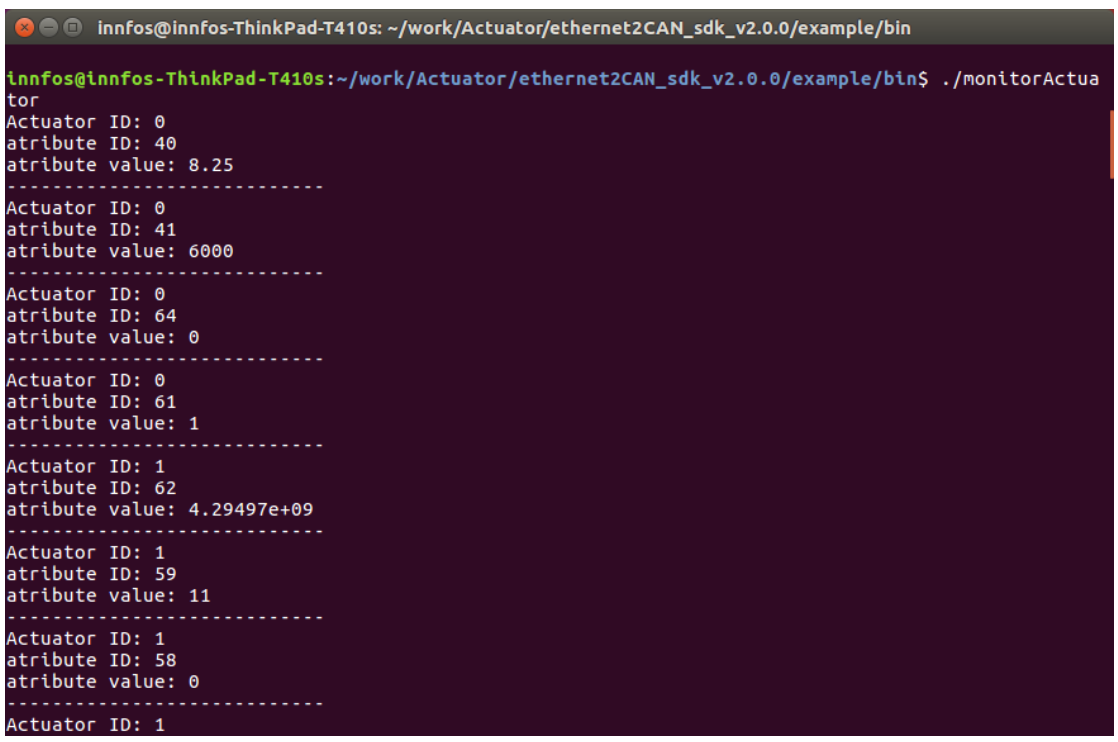
打开终端，进入 example/bin 目录，输入命令./lookupActuarors



此窗口会显示当前已连接的执行器数量，可以 ctrl+c 结束程序

2. 监测执行器状态

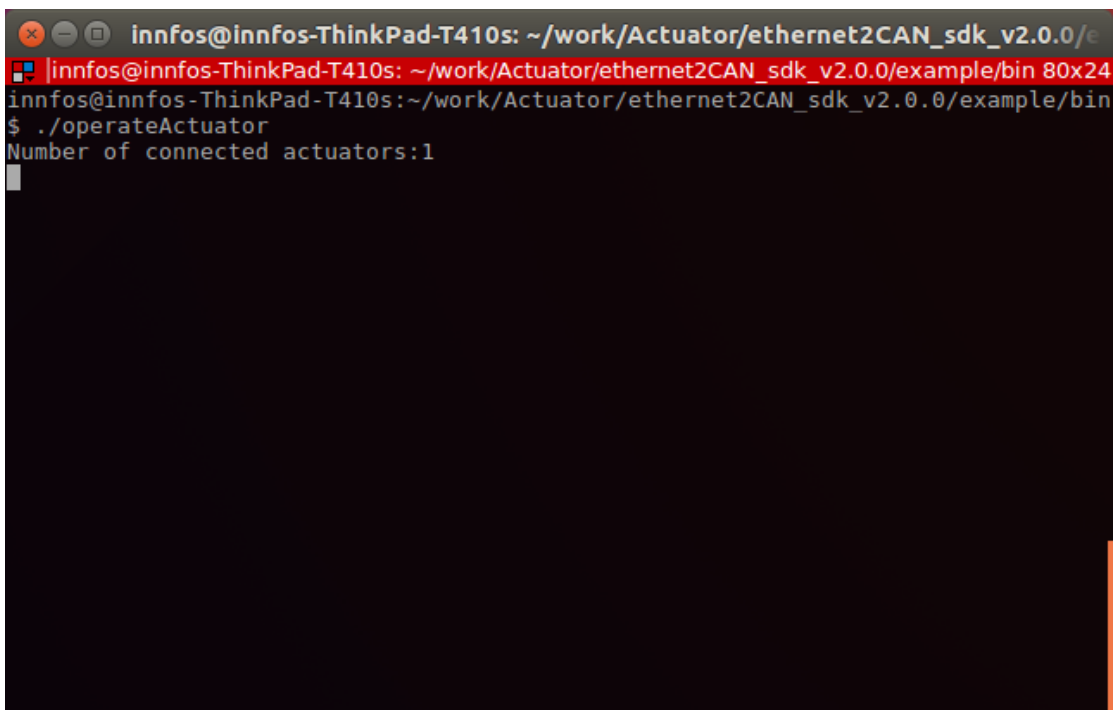
打开终端，进入 example/bin 目录，输入命令./monitorActuator



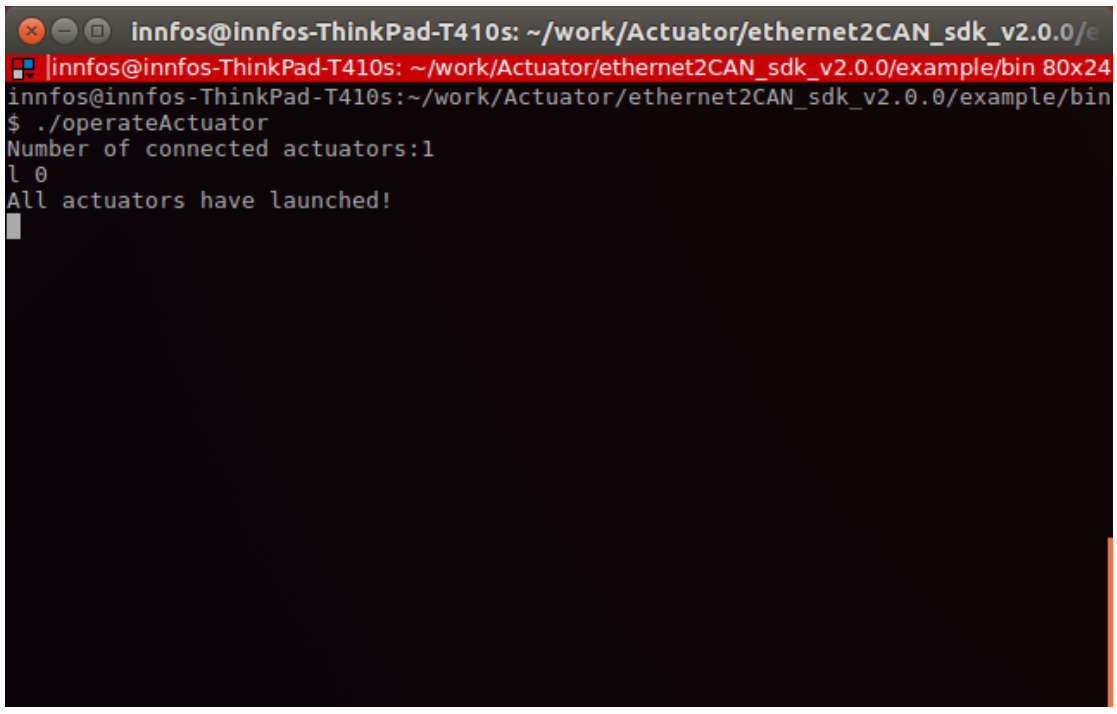
其中 Actuator ID 为执行器 id,attribute ID 为监测的执行器属性 Id，attribute value 为对应的属性值，可以 ctrl+c 结束程序

3. 控制执行器

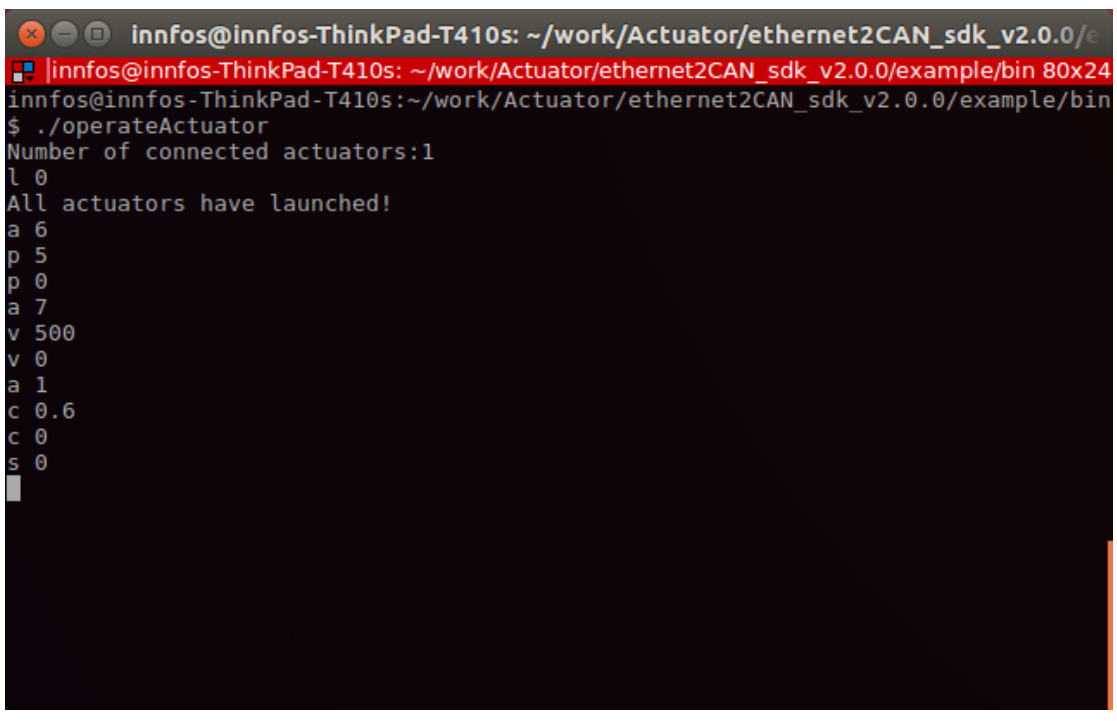
打开终端，进入 example/bin 目录，输入命令./operateActuator



表示执行器已经找到，输入命令 l0，该命令会启动所有已连接的执行器，如果启动成功，执行器会有绿色指示灯闪烁，表示已经启动成功，终端窗如下显示

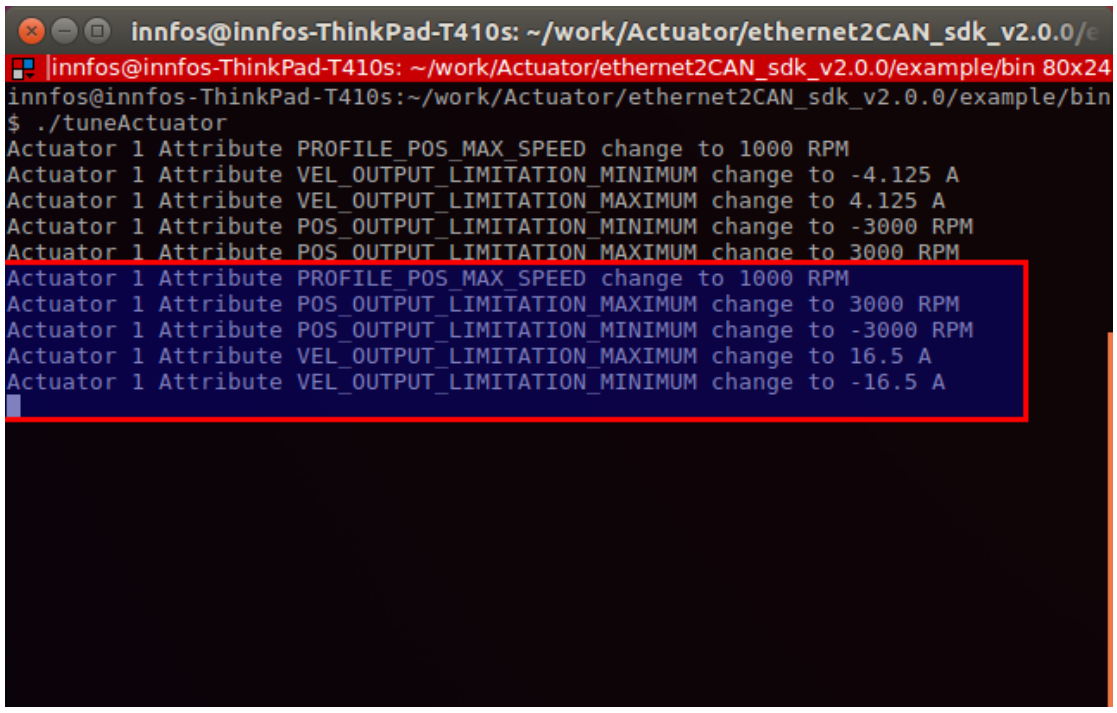


此时可激活执行器对应模式，比如输入 a 6 可以激活 profile position 模式，再输入 p 5，执行器会转动到 5 圈的位置；输入 a 7 可以激活 profile velocity 模式，再输入 v 500，执行器将以 500RPM 的速度转动，停止转动输入 v 0,；输入 a 1 可以激活电流模式，再输入 c 0.6，执行器将以恒定 0.6A 的电流转动（如果执行器不动，可用手轻轻转动一下执行器），可以 ctrl+c 以后再 ctrl+d 结束程序（因为有多线程等待键盘输入）



4. 控制器参数调整

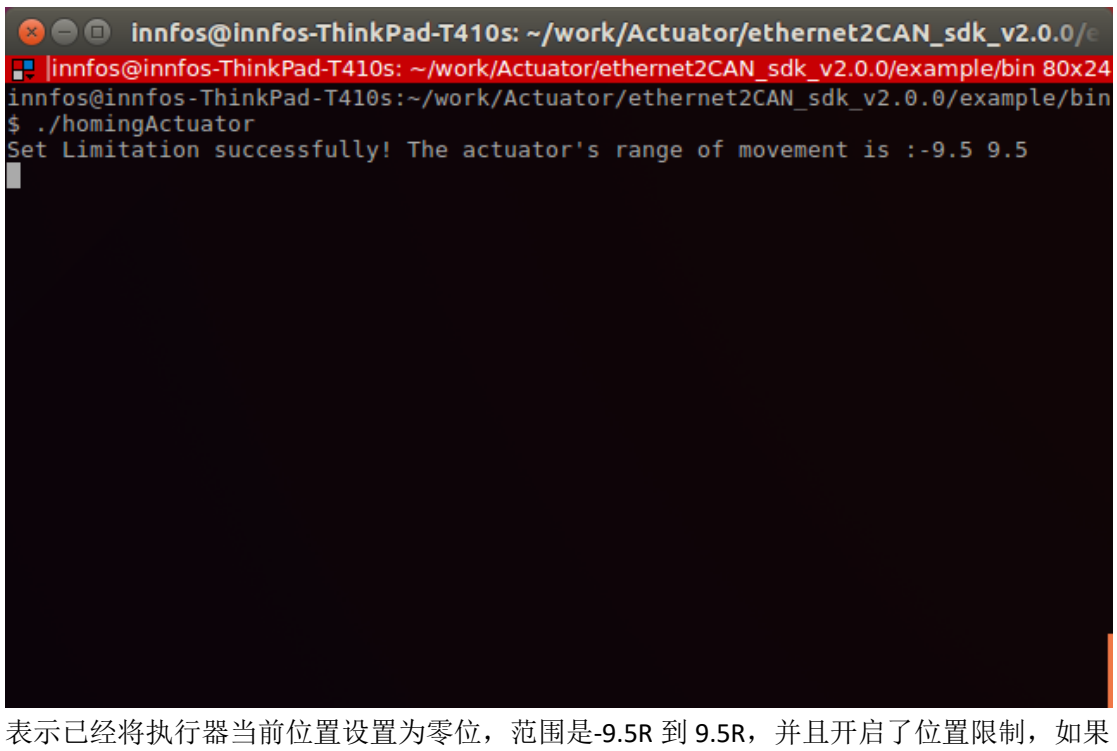
打开终端，进入 example/bin 目录，输入命令./tuneActuator



此示例程序自动启动执行器并将位置环输出设置为 3000RPM,速度环的电流最大输出为 16.5A,如果使用 profile position 模式转动执行器，执行器的最大速度不会超过 3000RPM;如果使用 profile velocity 模式转动执行器，执行器最大电流不会超过 16.5A，可以 ctrl+c 结束程序

5. 执行器归零

打开终端，进入 example/bin 目录，输入命令./homingActuator



表示已经将执行器当前位置设置为零位，范围是-9.5R 到 9.5R，并且开启了位置限制，如果 profile position 模式下，输入此范围之外的位置，执行器不会转动，可以 ctrl+c 结束程序

6.SDK 说明

1.介绍

本 SDK 提供了与 INNFOS 执行器通信的接口，可通过串口或者以太网对已经连接好的执行器进行查找、状态查询、属性调整和自定义控制。如果想快速了解 sdk 基本内容和使用方法，请查看 example/src 中的相关代码

2.项目中使用 sdk

本 sdk 遵循 c++11 标准，所以在构建项目之前请确认编译选项支持 c++11（比如 gcc 中使用-std=c++11）;
将 sdk 集成到项目中的基本步骤（最好先参考 example 中的 CMakeLists.txt）:

1. 将 sdk/include、sdk/include/QtCore、sdk/include/QtNetwork、sdk/include/QtSerialPort 加入到项目的包含目录，用于关联共享库中的方法;
2. 将库文件目录 sdk/lib/linux_x86_64（windows 目录为 sdk/lib/debug 和 sdk/lib/release），以便可执行文件能链接到共享库，并保证运行时能够关联到共享库;
3. 将必要的元素加入到构建过程中（比如 CMake 中的 target_link_libraries）

3.命名空间

在../sdk/include/actuatordefine.h 定义了命名空间 Actuator,并且枚举了 sdk 中所有用到的类型和类型值：

连接状态，用于执行器和 CAN 的连接状态判断：

```
enum ConnectStatus{//connect status
    NO_CONNECT,
    CAN_CONNECTED=0x02,
    ACTUATOR_CONNECTED=0x04,
};
```

通道 ID,用于标识执行器图表数据的通道索引：

```
enum Channel_ID{
    channel_1=0,
    channel_2,
    channel_3,
    channel_4,
    channel_cnt
};
```

错误类型定义，定义了执行器内部和连接等错误代码：

```
enum ErrorsDefine
{
    ERR_NONE = 0,
    //执行器过压错误
    ERR_ACTUATOR_OVERVOLTAGE=0x01,
    //执行器欠压错误
    ERR_ACTUATOR_UNDERVOLTAGE=0x02,
    //执行器堵转错误
    ERR_ACTUATOR_LOCKED_ROTOR=0x04,
    //执行器过温错误
    ERR_ACTUATOR_OVERHEATING=0x08,
    //执行器读写错误
    ERR_ACTUATOR_READ_OR_WIRTE=0x10,
    //执行器多圈计数错误
    ERR_ACTUATOR_MULTI_TURN=0x20,
    //执行器逆变器温度器错误
    ERR_INVERTOR_TEMPERATURE_SENSOR=0x40,
    //执行器 CAN 通信错误
    ERR_CAN_COMMUNICATION=0x80,
    //执行器温度传感器错误
    ERR_ACTUATOR_TEMPERATURE_SENSOR=0x100,
    //执行器 DRV 保护
    ERR_DRV_PROTECTION=0x400,
    //执行器 ID 不唯一错误
    ERR_ID_UNUNIQUE=0x800
    //执行器未连接错误
    ERR_ACTUATOR_DISCONNECTION=0x801,
    //CAN 通信转换板未连接错误
    ERR_CAN_DISCONNECTION=0x802,
    //无可用 ip 地址错误
    ERR_IP_ADDRESS_NOT_FOUND=0x803,
    //执行器非正常关机错误
    ERR_ABNORMAL_SHUTDOWN=0x804,
    //执行器关机时参数保存错误
    ERR_SHUTDOWN_SAVING=0x805,
    ERR_UNKOWN=0xffff
};
```

在线状态，用于标识执行器是否处于连接状态：

```
enum OnlineStatus{
    Status_Online=0x00,
    Status_Offline=0x01,
};
```

开关状态，标识执行器的开关机状态：

```
enum SwitchStatus{
    ACTUATOR_SWITCH_OFF=0,
    ACTUATOR_SWITCH_ON=1,
};
```

图表开关，用于标识执行器图表功能的开启或关闭：

```
enum ChartSwitchStatus
{
    CHART_SWITCH_OFF=0,
    CHART_SWITCH_ON=1,
};
```

电流环图表索引，用于标识电流图表是 IQ 值还是 ID 值

```
enum CurrnetChart
{
    IQ_CHART=0,
    ID_CHART=1,
};
```

归零模式，分为手动和自动两种

```
enum HomingOperationMode{
    Homing_Auto=0,
    Homing_Manual,
};
```

通信方式，可通过以太网或者串口两种方式与执行器通信，初始化执行器控制器时候要指定方式，默认为以太网通信：

```
enum CommunicationType{
    Via_Ethernet,
    Via_Serialport,
};
```

操作标识，标识操作完成，可用于判断执行器控制器的指令执行状态：

```
enum OperationFlags{
    //自动识别完成
    Recognize_Finished,
    //执行器启动完成（如果连接的是多个执行器，会触发多次启动完成信号）
    Launch_Finished,
    //执行器关闭完成（如果连接的是多个执行器，会触发多次关闭完成信号）
    Close_Finished,
    //执行器参数保存完成（如果连接的是多个执行器，会触发多次参数保存完成信号）
    Save_Params_Finished,
    //执行器参数保存失败
    Save_Params_Failed,
    //暂未实现
    Attribute_Change_Finished,
};
```

执行器模式，标识当前执行器的模式：

```
enum ActuatorMode{
    Mode_None,
    Mode_Cur,//电流模式
    Mode_Vel,//速度模式
    Mode_Pos,//位置模式
    Mode_Teaching,//暂未实现
    Mode_Profile_Pos=6,//profile 位置模式，比较于位置模式，该模式有加速减速过程
    Mode_Profile_Vel, //profile 速度模式，比较于速度模式，该模式有加速减速过程

    Mode_Homing,//归零模式
};
```

执行器属性，标识了执行器所有相关属性：

```
enum ActuatorAttribute{
    CUR_IQ_SETTING,//电流 IQ 值
    CUR_PROPORTIONAL,//电流比例
    CUR_INTEGRAL,//电流积分
    CUR_ID_SETTING,//电流 ID 值
    CUR_MINIMUM,//预留
```

```
CUR_MAXIMUM,//预留
CUR_NOMINAL,//预留
CUR_OUTPUT,//预留
CUR_MAXSPEED,//电流环最大速度
ACTUAL_CURRENT,//当前电流值
VEL_SETTING,//速度设置
VEL_PROPORTIONAL,//速度比例
VEL_INTEGRAL,//速度积分
VEL_OUTPUT_LIMITATION_MINIMUM,//速度环输出最小电流比例
VEL_OUTPUT_LIMITATION_MAXIMUM,//速度环输出最大电流比例
ACTUAL_VELOCITY,//速度值
POS_SETTING,//位置设置
POS_PROPORTIONAL,//位置比例
POS_INTEGRAL,//位置积分
POS_DIFFERENTIAL,//位置微分
POS_OUTPUT_LIMITATION_MINIMUM,//位置环输出最小速度比例
POS_OUTPUT_LIMITATION_MAXIMUM,//位置环输出最大速度比例
POS_LIMITATION_MINIMUM,//最小位置限制
POS_LIMITATION_MAXIMUM,//最大位置限制
HOMING_POSITION,//归零位置
ACTUAL_POSITION,//当前位置
PROFILE_POS_MAX_SPEED,//profile position 模式最大速度
PROFILE_POS_ACC,//profile position 模式加速度
PROFILE_POS_DEC,//profile position 模式减速速度
PROFILE_VEL_MAX_SPEED,//profile velocity 模式最大速度
PROFILE_VEL_ACC,//profile velocity 模式加速度
PROFILE_VEL_DEC,//profile velocity 模式减速速度
CHART_FREQUENCY,//图像频率
CHART_THRESHOLD,//图像阈值
CHART_SWITCH,//图像开关
POS_OFFSET,//位置偏移
VOLTAGE,//电压
POS_LIMITATION_SWITCH,//开启或关闭位置限制
HOMING_CUR_MAXIMUM,//归零最大电流
HOMING_CUR_MINIMUM,//归零最小小电流
CURRENT_SCALE,//物理最大电流值
VELOCITY_SCALE,//速度最大电流值
FILTER_C_STATUS,//电流环滤波是否开启
FILTER_C_VALUE,//电流环滤波值
FILTER_V_STATUS,//速度环滤波是否开启
FILTER_V_VALUE,//速度环滤波值
FILTER_P_STATUS,//位置环滤波是否开启
FILTER_P_VALUE,//位置环滤波值
INERTIA,//惯量
LOCK_ENERGY,//堵转保护能量
ACTUATOR_TEMPERATURE,//执行器温度
INVERTER_TEMPERATURE,//逆变器温度
ACTUATOR_PROTECT_TEMPERATURE,//执行器保护温度
ACTUATOR_RECOVERY_TEMPERATURE,//执行器恢复温度
INVERTER_PROTECT_TEMPERATURE,//逆变器保护温度
INVERTER_RECOVERY_TEMPERATURE,//逆变器恢复温度
CALIBRATION_SWITCH,//预留
CALIBRATION_ANGLE,//预留
ACTUATOR_SWITCH,//执行器开关机
FIRMWARE_VERSION,//执行器固件版本
ONLINE_STATUS,//执行器是否在线
DEVICE_ID,//执行器 Id
SN_ID,//执行器 SN 号
MODE_ID,//执行器当前模式
ERROR_ID,//错误代码
RESERVE_0, //预留
RESERVE_1, //预留
RESERVE_2, //预留
RESERVE_3, //预留
DATA_CNT,
DATA_CHART,//预留
DATA_INVALID,
```

```
};
```

指令集合，用户操作执行器无需关心指令集合：

```
enum Directives
{
    D_HANDSHAKE=0x00,
    D_READ_VERSION=0x01,
    D_READ_ADDRESS=0x02,
    D_READ_CONFIG=0x03,
    D_READ_CUR_CURRENT=0x04,
    D_READ_CUR_VELOCITY=0x05,
    D_READ_CUR_POSITION=0x06,
    D_SET_MODE=0x07,
    D_SET_CURRENT=0x08,//设置当前 q 轴电流
    D_SET_VELOCITY=0x09,
    D_SET_POSITION=0x0a,
    D_SET_PAIRS=0x0b,
    D_SET_CURRENT_ID=0x0c,//设置当前 d 轴电流
    D_SAVE_PARAM=0x0d,//
    D_SET_CURRENT_P=0x0e,//电流环的 p
    D_SET_CURRENT_I=0x0f,
    D_SET_VELOCITY_P=0x10,
    D_SET_VELOCITY_I=0x11,
    D_SET_POSITION_P=0x12,
    D_SET_POSITION_I=0x13,
    D_SET_POSITION_D=0x14,
    D_READ_CUR_P=0x15,
    D_READ_CUR_I=0x16,
    D_READ_VEL_P=0x17,
    D_READ_VEL_I=0x18,
    D_READ_POS_P=0x19,
    D_READ_POS_I=0x1a,
    D_READ_POS_D=0x1b,
    D_READ_PROFILE_POS_MAX_SPEED=0x1c,
    D_READ_PROFILE_POS_ACC=0x1d,
    D_READ_PROFILE_POS_DEC=0x1e,
    D_SET_PROFILE_POS_MAX_SPEED=0x1f,
    D_SET_PROFILE_POS_ACC=0x20,
    D_SET_PROFILE_POS_DEC=0x21,
    D_READ_PROFILE_VEL_MAX_SPEED=0x22,
    D_READ_PROFILE_VEL_ACC=0x23,
    D_READ_PROFILE_VEL_DEC=0x24,
    D_SET_PROFILE_VEL_MAX_SPEED=0x25,
    D_SET_PROFILE_VEL_ACC=0x26,
    D_SET_PROFILE_VEL_DEC=0x27,
    D_READ_CURRENT_MAXSPEED=0x28,
    D_SET_CURRENT_MAXSPEED=0x29,
    D_SET_SWITCH_MOTORS=0x2a,
    D_READ_MOTORS_SWITCH=0x2b,
    D_SET_CURRENT_PID_MIN = 0x2e,//设置电流环的 pid 的上下限
    D_SET_CURRENT_PID_MAX=0x2f,
    D_SET_VELOCITY_PID_MIN=0x30,
    D_SET_VELOCITY_PID_MAX=0x31,
    D_SET_POSITION_PID_MIN=0x32,
    D_SET_POSITION_PID_MAX=0x33,
    D_READ_CURRENT_PID_MIN=0x34,//读取电流环的 pid 的上下限
    D_READ_CURRENT_PID_MAX=0x35,
    D_READ_VELOCITY_PID_MIN=0x36,
    D_READ_VELOCITY_PID_MAX=0x37,
    D_READ_POSITION_PID_MIN=0x38,
    D_READ_POSITION_PID_MAX=0x39,
    D_READ_CHANNEL_2=0x3a,
    D_READ_CHANNEL_3=0x3b,
    D_READ_CHANNEL_4=0x3c,
    D_SET_DEVICE_ID=0x3d,
    D_SOFTWARE_CLOSE=0x3e,
    D_SET_CHART_THRESHOLD=0x3f,
```

```
D_SET_CHART_FREQUENCY=0x40,
D_READ_CHART_THRESHOLD=0x41,
D_READ_CHART_FREQUENCY=0x42,
D_CHART_DATA_STATR=0x43,
D_CAN_CONNECT=0x44,
D_READ_VOLTAGE=0x45,
D_CHART_OPEN=0x46,
D_CHART_CLOSE=0x47,
D_CHANNEL2_OPEN=0x48,//
D_CHANNEL2_CLOSE=0x49,//
D_CHANNEL3_OPEN=0x4a,//
D_CHANNEL3_CLOSE=0x4b,//
D_CHANNEL4_OPEN=0x4c,//
D_CHANNEL4_CLOSE=0x4d,//
D_READ_CHANNEL_1=0x4e,
D_SET_VOLTAGE=0x4f,//
D_CRC_ERROR=0x50,
D_CHANNEL1_OPEN=0x51,
D_CHANNEL1_CLOSE=0x52,
D_READ_CURRENT_SCALE=0x53,
D_SET_CUR_TRIGGER_MODE=0x54,//
D_READ_MOTOR_MODE=0x55,
D_READ_TEMP_MOTOR=0x5f,
D_READ_TEMP_INVERTER=0x60,
D_SET_TEMP_PROTECT=0x6b,
D_READ_TEMP_PROTECT=0x6c,
D_SET_TEMP_RECOVERY=0x6d,
D_READ_TEMP_RECOVERY=0x6e,
D_SET_INVERTER_TEMP_PROTECT=0x61,
D_READ_INVERTER_TEMP_PROTECT=0x62,
D_SET_INVERTER_TEMP_RECOVERY=0x63,
D_READ_INVERTER_TEMP_RECOVERY=0x64,

D_SET_FILTER_C_STATUS=0x70,
D_READ_FILTER_C_STATUS=0x71,
D_SET_FILTER_C_VALUE=0x72,
D_READ_FILTER_C_VALUE=0x73,
D_SET_FILTER_V_STATUS=0x74,
D_READ_FILTER_V_STATUS=0x75,
D_SET_FILTER_V_VALUE=0x76,
D_READ_FILTER_V_VALUE=0x77,
D_SET_FILTER_P_STATUS=0x78,
D_READ_FILTER_P_STATUS=0x79,
D_SET_FILTER_P_VALUE=0x7a,
D_READ_FILTER_P_VALUE=0x7b,
D_SET_INERTIA = 0x7c,
D_READ_INERTIA = 0x7d,
D_SET_LOCK_ENERGY=0x7e,
D_READ_LOCK_ENERGY=0x7f,

D_SET_MAX_POS=0x83,//上下限
D_SET_MIN_POS=0x84,
D_READ_MAX_POS=0x85,
D_READ_MIN_POS=0x86,
D_SET_HOMING_POS=0x87,
D_CLEAR_HOMING=0x88,//清除 homing 相关信息
D_SET_POS_OFFSET=0x89,
D_READ_POS_OFFSET=0x8a,
D_READ_HOMING_LIMIT=0x8b,
D_SET_HOMING_LIMIT=0x8c,
D_SET_HOMING_OPERATION=0x8d,
D_SET_HOMING_MIN=0x8e,
D_SET_HOMING_MAX=0x8f,
D_SET_HOMING_CUR_MIN=0x90,
D_SET_HOMING_CUR_MAX=0x91,
D_READ_HOMING_CUR_MIN=0x92,
D_READ_HOMING_CUR_MAX=0x93,
```

```
    D_SWITCH_CALIBRATION=0xa0,
    D_READ_CALIBRATION_SWITCH=0xa1,
    D_START_CALIBRATION=0xa2,
    D_SET_CALIBRATION_ANGLE=0xa3,
    D_READ_CALIBRATION_ANGLE=0xa4,
    D_SWITCH_CALIBRATION_VEL=0xa5,

    D_READ_RESERVE_0=0xd0,
    D_READ_RESERVE_1=0xd1,
    D_READ_RESERVE_2=0xd2,
    D_READ_RESERVE_3=0xd3,
    D_READ_LAST_STATE=0xb0,//读取上一次状态（是否正常关机）

    D_IP_BROADCAST=0xc0,//广播查找 ip 地址
    D_TMP_COMMAND=0xc1,//与中间板通信的协议指令

    D_CLEAR_ERROR=0xfe,//清理错误
    D_CHECK_ERROR=0xff,//错误提示
    DIRECTIVES_INVALID,
};
```

4.主要类说明

1.相关 API

用户与执行器进行的全部交互都在此类中实现。

`static void initController(int &argc, char **argv,int nCommunicationType=Actuator::Via_Ethernet)`
初始化控制器，使用控制器之前必须先初始化，通信方式可分为串口通信和以太网通信两种，默认为以太网通信
示例代码：

```
int main(int argc, char *argv[])
{
    //初始化控制器
    ActuatorController::initController(argc,argv,Actuator::Via_Ethernet);
    ....
}
```

`static ActuatorController * getInstance();`
获取控制器对象实例，用户只能通过此接口获取控制器对象实例，而不应该以 new 的方式获取
示例代码：

```
int main(int argc, char *argv[])
{
    //初始化控制器
    ActuatorController::initController(argc,argv,Actuator::Via_Ethernet);
    ActuatorController * pController = ActuatorController::getInstance();
    ....
}
```

`static void processEvents();`
处理控制器事件，控制器所有的信号通知以及执行器属性刷新都依赖此函数的调用，所以不应该阻塞该函数的调用
示例代码：

```
...
//执行控制器事件循环
while (!bExit)
{
    ActuatorController::processEvents();
}
...
```

```
void autoRecoginze();
```

识别所有可用设备，初始化完成后调用此函数，识别完成会触发触发 m_sOperationFinished 信号，操作类型为 OperationFlags::Recognize_Finished

示例代码：

```
int main(int argc, char *argv[])
{
    //初始化控制器
    ActuatorController::initController(argc,argv,Actuator::Via_Ethernet);
    ActuatorController * pController = ActuatorController::getInstance();

    //关联控制器的操作信号
    int nOperationConnection = pController->m_sOperationFinished.s_Connect([=](uint8_t nDeviceId,uint8_t operationType){
        switch (operationType) {
        case Actuator::Recognize_Finished://自动识别完成
            if(pController->hasAvailableActuator())
            {
                vector<uint8_t> idArray = pController->getActuatorIdArray();
                cout << "Number of connected actuators:" << idArray.size() << endl;
            }
            break;
        default:
            break;
        }
    });
    //自动识别已连接执行器
    pController->autoRecoginze();
    ...
}
```

```
bool hasAvailableActuator()const;
```

当前控制器是否识别到可用执行器

```
vector<uint8_t> getActuatorIdArray()const;
```

获取当前控制器识别到的执行器 id 数组

```
void activeActuatorMode(vector<uint8_t> idArray, const Actuator::ActuatorMode nMode);
```

激活指定执行器的指定模式，激活成功后会触发 m_sActuatorAttrChanged 信号，属性 id 值为 ActuatorAttribute:: MODE_ID

```
void launchAllActuators();
```

启动所有已识别的执行器，每个执行器启动成功后会触发 m_sOperationFinished 信号，操作类型为 OperationFlags::Launch_Finished，如果执行器处于开机状态，则不会触发信号

```
void closeAllActuators();
```

关闭所有已识别的执行器，每个执行器关闭成功后会触发 m_sOperationFinished 信号，操作类型为 OperationFlags::Close_Finished

```
void launchActuator(uint8_t id);
```

启动指定 id 的执行器，执行器启动成功后会触发 m_sOperationFinished 信号，操作类型为 OperationFlags::Launch_Finished，如果执行器处于开机状态，则不会触发信号

```
void closeActuator(uint8_t id);
```

关闭指定 id 的执行器，执行器关闭成功后会触发 m_sOperationFinished 信号，操作类型为 OperationFlags::Close_Finished

```
void switchAutoRefresh(uint8_t id,bool bOpen);
```

开启或关闭指定 id 执行器的自动刷新功能，自动请求设备电流、速度、位置、电压、温度、逆变器温度（默认关闭此功能）

```
void setAutoRefreshInterval(uint8_t id, uint32_t mSec);
```

设置指定 id 执行器的自动刷新时间间隔（默认时间间隔为 1s）

```
void setPosition(uint8_t id,double pos);
```

设置指定 id 执行器的的位置，范围是-128 到 128,单位是 Revolution，为了执行效率，此指令不会触发信号

```
void setVelocity(uint8_t id,double vel);
```
设置指定 id 执行器的的速度，单位是 RPM,为了执行效率，此指令不会触发信号

```
void setCurrent(uint8_t id,double current);
```
设置指定 id 执行器的的电流，单位是 A，为了执行效率，此指令不会触发信号

```
double getPosition(uint8_t id,bool bRefresh=false)const;
```
获取指定 id 执行器的当前位置,单位是 Revolution，因为请求返回存在延时，当前得到的值是上一次请求返回成功后的结果，bRefresh 如果为 true，调用此函数后会自动请求一次执行器当前位置

```
double getVelocity(uint8_t id,bool bRefresh=false)const;
```
获取指定 id 执行器的当前速度,单位是 RPM，因为请求返回存在延时，当前得到的值是上一次请求返回成功后的结果，bRefresh 如果为 true，调用此函数后会自动请求一次执行器当前速度

```
double getCurrent(uint8_t id,bool bRefresh=false)const;
```
获取指定 id 执行器的当前电流,单位是 A，因为请求返回存在延时，当前得到的值是上一次请求返回成功后的结果，bRefresh 如果为 true，调用此函数后会自动请求一次执行器当前电流

```
void setActuatorAttribute(uint8_t id,Actuator::ActuatorAttribute attrId,double value);
```
设置指定 id 执行器的指定属性 attrId 的值 value，成功后会触发 m_sActuatorAttrChanged 信号，信号的执行器 id，属性 id 和属性值对应于设置的值

```
double getActuatorAttribute(uint8_t id, Actuator::ActuatorAttribute attrId)const;
```
获取指定 id 执行器的指定属性 attrId 的值，因为请求返回存在延时，当前得到的值是上一次请求该属性返回成功后的结果

```
void saveAllParams(uint8_t id);
```
保存指定 id 执行器的当前设置参数，以便参数下次开机依然生效，保存成功后会触发 m_sOperationFinished 信号，操作类型为 OperationFlags::Save_Params_Finished，保存失败的操作类型则为 OperationFlags::Save_Params_Failed

```
void clearHomingInfo(uint8_t id);
```
清除指定 id 执行器的归零和左右位置限制等信息，清除以后，如果想开启位置限制功能，必须先设置好适当的零位和左右位置限制

```
void setHomingOperationMode(uint8_t id,uint8_t nMode);
```
目前只支持手动模式设置零位和左右位置限制，所以该函数暂未实现功能

```
void setMinPosLimit(uint8_t id);
```
将指定 id 执行器的当前位置设置执行器的最小位置限制，设置成功后会触发 m_sActuatorAttrChanged 信号，属性 id 值为 ActuatorAttribute::POS_LIMITATION_MINIMUM

```
void setMinPosLimit(uint8_t id,double posValue);
```
设置指定 id 执行器的最小位置限制，其值为 posValue，设置成功后会触发 m_sActuatorAttrChanged 信号，属性 id 值为 ActuatorAttribute::POS_LIMITATION_MINIMUM

```
void setMaxPosLimit(uint8_t id);
```
将指定 id 执行器的当前位置设置执行器的最大位置限制，设置成功后会触发 m_sActuatorAttrChanged 信号，属性 id 值为 ActuatorAttribute::POS_LIMITATION_MAXIMUM

```
void setMaxPosLimit(uint8_t id,double posValue);
```

设置指定 id 执行器的最大位置限制，其值为 posValue，设置成功后会触发 m_sActuatorAttrChanged 信号，属性 id 值为 ActuatorAttribute::POS_LIMITATION_MAXIMUM

```
void setHomingPosition(uint8_t id,double posValue);
```
设置指定 id 执行器的 posValue 为零位，设置成功后会触发 m_sActuatorAttrChanged 信号，属性 id 值为 ActuatorAttribute::HOMING_POSITION

```
void openChartChannel(uint8_t id,uint8_t  nChannelId);
```
开启指定 id 执行器的指定通道为 nChannelId（Channel_ID::channel_1 到 Channel_ID::channel_4）的图表通道，开启后如果触发了图表数据生成条件，会触发 m_sNewChartStart 信号，提示新的图表数据已经生成，然后会触发图表数据信号 m_sChartValueChange 信号

```
void closeChartChannel(uint8_t id, uint8_t nChannelId);
```
关闭指定 id 执行器的指定通道为 nChannelId（Channel_ID::channel_1 到 Channel_ID::channel_4）的图表通道，关闭后将不会触发新的图表数据

```
void switchChartAllChannel(uint8_t id,bool bOn);
```
开启或关闭指定 id 执行器的所有图表通道

```
void setCurrentChartMode(uint8_t id, uint8_t mode);
```
设置指定 id 的电流图表数据的模式为 ID 模式或者 IQ 模式（CurrnetChart::ID_CHART、CurrnetChart::IQ_CHART）默认是 CurrnetChart::ID_CHART

```
void regainAttrbute(uint8_t id,uint8_t attrId);
```
请求刷新指定 id 执行器的指定属性 attrId，请求后成功返回会触发 m_sActuatorAttrChanged 信号，属性 id 值为 attrId

```
vector<uint16_t> getErrorHistory(uint8_t id);
```
获取指定 id 执行器的错误历史记录，记录为错误代码，具体含义可参考 Actuator 命名空间中的 ErrorsDefine

```
void reconnect(uint8_t id);
```
重新连接指定 id 执行器，重连成功后会触发发 m_sActuatorAttrChanged 信号，属性 id 值为 ActuatorAttribute::ONLINE_STATUS，属性值为 OnlineStatus:: Status_Online

```
void clearError(uint8_t id);
```
清除指定 id 执行器的错误，清除成功后会触发 m_sActuatorAttrChanged 信号，属性 id 值为 ActuatorAttribute:: ERROR_ID，属性值为 ErrorsDefine:: ERR_NONE

```
string versionString()const;
```
获取指定 sdk 的版本号字符串，格式为：主版本号.次版本号.发布版本号

2.信号

```
CSignal<uint8_t,uint8_t> m_sOperationFinished;
```
操作完成信号,第一个 uint8_t 代表执行器 id，如果是 0 不代表特定执行器，第二个 uint8_t 代表操作类型
示例代码：

```
...
//关联控制器的操作信号
    int nOperationConnection = pController->m_sOperationFinished.s_Connect([&](uint8_t nDeviceId,uint8_t operationType){
        switch (operationType) {
        case Actuator::Recognize_Finished://自动识别完成
            if(pController->hasAvailableActuator())
            {
                vector<uint8_t> idArray = pController->getActuatorIdArray();
                cout << "Number of connected actuators:" << idArray.size() << endl;
                foreach (uint8_t id, idArray) {
                    if(pController->getActuatorAttribute(id,Actuator::ACTUATOR_SWITCH)!=Actuator::ACTUATOR_SWITCH_OFF)
                    {
                        ++ nLaunchedActuatorCnt;
                        if(nLaunchedActuatorCnt == pController->getActuatorIdArray().size())//所有执行器都已启动完成
                        {
                            cout << "All actuators have launched!"  << endl;
                        }
                    }
                }

                break;
            }
            break;
        case Actuator::Launch_Finished:
            if(++nLaunchedActuatorCnt == pController->getActuatorIdArray().size())//所有执行器都已启动完成
            {
                cout << "All actuators have launched!"  << endl;
            }
            break;
        default:
            break;
```

```
        }
    });
...
```

CSignal<uint8_t,uint8_t,double> m_sRequestBack;
请求返回信号，保留信号，暂未实现

CSignal<uint8_t,uint16_t,std::string> m_sError;
错误信号：uint8_t 代表执行器 id，如果是 0 不代表特定执行器，uint16_t 代表错误代码，std::string 代表错误信息字符串
示例代码：

```
...
    //关联错误信号
    int nErrorConnection = pController->m_sError.s_Connect([=](uint8_t nDeviceId,uint16_t nErrorType,string errorInfo){
        if(nDeviceId==0)
        {
            cout << "Error: " << (int)nErrorType << " " << errorInfo << endl;
        }
        else
        {
            cout << "Actuator " << (int)nDeviceId << " " <<"error " << (int)nErrorType << " " << errorInfo << endl;
        }
    });
...
```

CSignal<uint8_t,uint8_t,double> m_sActuatorAttrChanged;
执行器属性变化信号：第一个 uint8_t 代表执行器 id，第二个 uint8_t 代表执行器属性 id，double 代表执行器该属性的值，示例代码：

```
...
    //关联控制器控制的执行器属性变化信号
    int nAttrConnection =pController->m_sActuatorAttrChanged.s_Connect([=](uint8_t nDeviceId,uint8_t nAttrId,double value){
        cout << "Actuator ID: " << (int)nDeviceId << endl;
        cout << "atribute ID: " << (int)nAttrId << endl;
        cout << "atribute value: " << value << endl;
        cout << "----------------------------"<<endl;
    });
...
```

CSignal<> m_sNewChartStart;
图表新周期开始信号，当执行器触发新的图表数据时，会触发此信号，代表会开始发送新周期的图表数据

CSignal<uint8_t,double> m_sChartValueChange;
图表数据信号，uint8_t 代表图表通道 id，double 代表图表数据，一个周期会有 200 个数据点