**控制代码调试详解**

1. 代码注释详解
   1. mainwindow.cpp注释详解

#include "mainwindow.h"

#include "ui\_mainwindow.h" // 以上两行为QT工程的自带头文件，主要是QT界面的初始化头文件

#include <QDebug> // qDebug() 输出数据信息到调试栏，相当于c++的 std::cout

#include <windows.h> // 貌似现在没啥用，最初为了使用windows系统自带接口读取串口，后来发现QSerialPort库更方便

#include <QTcpServer> // UDP网络通讯的必要头文件

#include <QHostAddress> // UDP通讯

#include <iostream> // C以及C++的标准输入输出库，在此项目中主要用作文件的读写操作，例如将GPS数据保存到txt文件

#define model\_switch // 控制界面开环与闭环切换使能，如果不定义这个，也就不能实现直线航行的开环闭环的切换

//#include <math.h> // math.h暂时没有使用，貌似C++的std包含充分的数学运算库，例如反正切函数atan()

// #include <QMutex> // 地址锁暂时没有想好要不要用，地址锁已经被纯C++地址锁替代，不再使用QMutex

/\*\*

\* @brief MainWindow::MainWindow

\* @param parent

\* 此函数是QT界面初始化的自带函数，函数名称以及后方自带的参数不可更改

\*/

MainWindow::**MainWindow**(QWidget \*parent) :

QMainWindow(parent),

ui(new Ui::MainWindow)

{

ui->setupUi(this); // 将界面的指针引入此工程的mainwindow类之下，QT自带操作，不可更改

\_main\_flash\_timer = new QTimer(); // 新建QT界面的显示刷新定时器，此定时器在主线程中使用，可以直接定义

connect(\_main\_flash\_timer, &QTimer::timeout, this, &MainWindow::flash\_data); // 定时器溢出槽函数绑定flash\_data函数

\_main\_secur\_timer = new QTimer(); // 主线程是否被卡死的判断定时器

connect(\_main\_secur\_timer, &QTimer::timeout, this, &MainWindow::secur\_flash\_time); // 同上，绑定溢出槽函数

\_main\_secur\_timer->setInterval(10000); // 设置溢出触发时间为10000ms

\_main\_secur\_timer->start(); // 开始定时器事件后，大概每隔10S会输出当前时间，以判断控制程序是否还正常运行

\_main\_secur\_GPS\_timer = new QTimer();// 新建惯导系统的保障定时器，相当于看门狗的功能

// 此定时器，如若惯导数据长时间不产生变化，则时间溢出进行错误处理，此功能在最初的推进器与主控系统未做电磁隔离时候非常有用

// 在2022年3月份更换了推进器之后，惯导受到的干扰次数明显减少，现在可以作为检测无人船控制系统硬件是否链接良好的手段

// 现在（2022年3月到2022年9月）一般出现错误就是因为硬件没有链接好

connect(\_main\_secur\_GPS\_timer, &QTimer::timeout, this, &MainWindow::process\_for\_GPS);

\_main\_secur\_GPS\_timer->setInterval(10);

\_main\_secur\_GPS\_timer->start();

// 摇杆以及数字输入框的规则设定，Slider为滑条，box为数字输入框，下边的操作基本都是在定义滑条的范围以及初始值为0

ui->x\_vel\_Slider->setRange(-1000, 1000);

ui->x\_vel\_Slider->setSliderPosition(0);

ui->y\_vel\_Slider->setRange(-1000, 1000);

ui->y\_vel\_Slider->setSliderPosition(0);

ui->w\_vel\_Slider->setRange(-1000, 1000);

ui->w\_vel\_Slider->setSliderPosition(0);

ui->x\_vel\_spin\_Box->setRange(-1000, 1000);

ui->x\_vel\_spin\_Box->setValue(0);

ui->x\_vel\_spin\_Box->setSingleStep(10);

ui->y\_vel\_spin\_Box->setRange(-1000, 1000);

ui->y\_vel\_spin\_Box->setValue(0);

ui->y\_vel\_spin\_Box->setSingleStep(10);

ui->w\_vel\_spin\_Box->setRange(-1000, 1000);

ui->w\_vel\_spin\_Box->setValue(0);

ui->w\_vel\_spin\_Box->setSingleStep(10);

ui->remoter\_radioButton->setChecked(true); // 控制程序开始即设置为遥控模式

ui->open\_loop\_checkBox->setChecked(true); // 开始即设置为遥控模式下的开环遥控

init\_var(); // 初始化主线程中用到的重要变量，以及开启相对应的子线程

}

/\*\*

\* @brief MainWindow::~MainWindow

\* mainwindow析构函数，点击界面的×按钮之后执行

\*/

MainWindow::~***MainWindow***()

{

write\_serial\_data(\_serial\_pwrpak, "UNLOGALL\r\n"); // 主程序退出之后，直接取消订阅惯导中所有信息，避免下次打开积攒数据

\_serial\_arduino->write("$3000,3000,3000,3000\n"); // 恢复推进器零位

delete \_thread\_pwrpak\_reader; // 删除子线程，一般不用删除，主线程结束，子线程自销毁

delete \_thread\_aduino\_reader; // 严谨一点删除最好

delete \_thread\_ethernet\_reader;

delete \_thread\_ethernet\_sender;

delete ui; // 删除ui

}

void MainWindow::**on\_start\_Button\_clicked**() // start按钮点击响应槽函数，一切控制代码的初始！

{

if(first\_start\_all\_flag){ // 对于各种类的新建，只能新建一次。

init\_all();

first\_start\_all\_flag = 0; // 防止多次点击start按钮导致bug

}

ui->start\_Button->setEnabled(false); // 双保险，开启程序之后start按钮禁用变灰，不可再次点击

}

/\*\*

\* @brief MainWindow::on\_stop\_Button\_clicked

\*

\* stop按钮的点击事件槽函数，当时作为一个测试，现在不再使用，以后做新代码的测试可以暂时借用此函数

\*/

void MainWindow::**on\_stop\_Button\_clicked**()

{

//\_serial\_arduino->write("$045,045,045,045\n");

// 停止推进器

}

/\*\*

\* @brief MainWindow::on\_send\_Button\_clicked

\* send按钮的点击事件槽函数，当时作为声呐数据发送代码的测试程序入口，现在声呐数据发送已经融合到子线程中，现可留作测试接口

\*/

void MainWindow::**on\_send\_Button\_clicked**()

{

}

/\*\*

\* @brief MainWindow::on\_test\_Button\_clicked

\* test按钮的点击事件槽函数，当时作为割草机模式的代码测试入口，现在割草机融入fsm\_states子线程，可留作测试接口，一般不要点击它

\*/

void MainWindow::**on\_test\_Button\_clicked**()

{

//\_thread\_ethernet\_sender->help\_net\_send\_flag = true;

//\_serial\_arduino->write("$3000,3000,3000,3000\n"); // 恢复推进器零位

qDebug() << "[DYZ test]";

}

/\*\*

\* @brief MainWindow::ConnectToClient

\* 废物函数，没用

\*/

void MainWindow::**ConnectToClient**()

{

}

/\*\*

\* @brief MainWindow::init\_all

\* 整合初始化程序，完成除自主控制任务外的所有任务开启

\*/

void MainWindow::**init\_all**()

{

//if(open\_serials()) // 要完全开启所有串口才可继续

open\_serials(); // 平时测试可能不会有所有串口都在，所以能开几个就算几个，正式上船测试要换成上边的，算是硬件操作保险

{

qDebug() << "[SERIALS] all serials have been opened";

write\_serial\_data(\_serial\_pwrpak, "UNLOGALL\r\n"); // 重新连接设置之后解除所有输出

qDebug() << "[IMU] UNLOGALL "; // IMU相关的提示什么的，都是惯导的意思，省事用的IMU代指惯导

write\_serial\_data(\_serial\_pwrpak, "CONNECTIMU SPI EPSON\_G320\r\n"); // 设置惯导系统中IMU的通讯对象，这个是官方推荐

write\_serial\_data(\_serial\_pwrpak, "serialconfig 115200 n 8 1 n on\r\n");// 设置惯导发送时候串口波特率

\_serial\_pwrpak->setBaudRate(QSerialPort::Baud115200); // 重设QT软件读写时候的波特率，算是个保险

write\_serial\_data(\_serial\_pwrpak, "SETINSPROFILE LAND\_PLUS\r\n"); // 开启LAND\_PLUS功能，失锁情况的长时间精度保持

write\_serial\_data(\_serial\_pwrpak, "GENERATERTKCORRECTIONS rtcmv3 com2\r\n");

// write\_serial\_data(\_serial\_pwrpak, "SETINITAZIMUTH 72 5\r\n"); // 航向角修正命令，一般手动输入，不自动设置

// SETINITAZIMUTH 72 5

// 设置采集为100HZ 0.01S间隔，主控制程序是20HZ

// 设置采集为20HZ 0.05S间隔已经极限输出

write\_serial\_data(\_serial\_pwrpak,

"SETINSTRANSLATION ANT1 0.14 -0.81 0.05 0.05 0.05 0.05\r\n"); // 天线1 位置设置

write\_serial\_data(\_serial\_pwrpak,

"SETINSTRANSLATION ANT2 0.14 0.84 0.05 0.05 0.05 0.05\r\n"); // 天线2 位置设置

// 以上天线位置的设置，需要手动测量一下，蘑菇头与信号接收器之间的相对距离，接收器上有对应的方向指示

// 按照接收器的坐标测量两个蘑菇头位置，然后按照 [x偏置] [y偏置] [z偏置] [x测量误差] [y测量误差] [z测量误差] 编写上述命令

// 例如上述ANT1，X测量误差就是感觉它在14±5厘米范围内

\_serial\_pwrpak->write("LOG INSPVAXA ONTIME 0.05\r\n"); // 设置INSPVAXA数据每0.1S发送一次

// INSPVAXA数据是诺瓦泰的惯导融合数据，官方推荐的，也是在代码中使用的数据，但是有个问题，速度量会有严重的延时

// 或许有别的数据格式会更好，具体参考诺瓦泰的说明书，诺瓦泰对于北斗信号的支持不是很好（某自动化院老师如是说）

//\_serial\_pwrpak->write("LOG BESTXYZA ONTIME 0.05\r\n"); // 设置INSPVAXA数据每0.1S发送一次

\_serial\_pwrpak->write("LOG GPGGA ONTIME 0.05\r\n"); // 声呐所需的GGA也可从惯导中获取

\_thread\_pwrpak\_reader->\_save\_flag = &save\_flag; // 数据保存的标志位，主要用到data\_saver子线程中

\_thread\_pwrpak\_reader->start(); // IMU线程开

\_thread\_aduino\_reader->start(); // arduino线程开

\_fsm\_states\_controller->start(); // 控制主模块fsm\_states子线程开

\_thread\_data\_saver = new data\_saver(); // data\_saver子线程创建

\_thread\_data\_saver->\_save\_flag = &save\_flag;

\_thread\_data\_saver->\_IMU\_data = \_IMU\_data; // 获取需要保存的数据指针

\_thread\_data\_saver->start(); // data\_saver子线程开

}

/\*else // 上边“if（串口全部打开）”的对应else

{

qDebug() << "[SERIALS] have something wrong";

}\*/

// 信息刷新定时器设置

\_main\_flash\_timer->setInterval(100); // 主界面的数据显示刷新，一秒10刷，10HZ，可适当降低刷新率

\_main\_flash\_timer->start(); // 刷新定时器开，对应槽函数按时执行

//qDebug() << "start tets print";

// 上边几个线程的开启，都有一个很重要的属性，他们直接或者间接依赖于串口数据，串口打开之后才能执行子线程

// 网络收发子线程就没这个问题，下边就是网络收发的ethernet\_reader和ethernet\_sender子线程的创建与开启

\_thread\_ethernet\_reader = new ethernet\_reader(&return\_flag, nullptr);

\_fsm\_states\_controller->\_fsm\_data->\_remoter\_set = &\_thread\_ethernet\_reader->remoter\_set\_read;

\_thread\_ethernet\_reader->\_fsm\_data = \_fsm\_states\_controller->\_fsm\_data;

// ethernet\_reader子线程中有个很重要的指针是\_fsm\_data，里边基本包含了控制所需的所有必要数据，

// 例如：惯导数据，PWM数据，位置跟踪误差，航速等等。。。。。

\_remoter\_flag = &\_thread\_ethernet\_reader->remoter\_flag; // 一下四个是王老师要求的岸基与控制界面融合

\_stance\_flag = &\_thread\_ethernet\_reader->stance\_flag; // 融合方式是用的网络消息传递命令，自然是要在ethernet\_reader子线程

\_coverage\_flag = &\_thread\_ethernet\_reader->coverage\_flag; // 但是现在这个融合框架不是很好，不应该像这样用标志位判断

\_tracking\_flag = &\_thread\_ethernet\_reader->tracking\_flag; // 只能说这个马马虎虎够用

\_thread\_ethernet\_reader->start(); // 网络UDP reader线程开

\_thread\_ethernet\_sender = new ethernet\_sender(\_IMU\_data,&return\_flag, nullptr);

//\_thread\_ethernet\_sender->\_base\_udpsocket = \_thread\_ethernet\_reader->\_base\_udpsocket;

// 接收与发送可以用同一个udp，但是多线程有限制，容易出BUG，所以分开两个子线程使用不同的端口

\_thread\_ethernet\_sender->start(); // 网络发送线程开

}

/\*\*

\* @brief MainWindow::init\_var

\* 代码所需的类全部初始化，为后边的多线程开启做准备

\*/

void MainWindow::**init\_var**() // 所有的类初始化都是必要的,start后必执行

{

\_serial\_sonar = new QSerialPort(); // 声呐串口，主要向声呐发送GGA（GPS标准格式）数据

\_serial\_pwrpak = new QSerialPort(); // 惯导串口

\_thread\_pwrpak\_reader = new pwrpak\_reader(\_serial\_pwrpak, &running, nullptr); // 惯导数据读取及处理子线程

\_IMU\_data = &\_thread\_pwrpak\_reader->IMU\_data; // IMU数据存储指针，指向存储位置，传给fsm\_states子线程

\_serial\_arduino = new QSerialPort(); // arduino串口，控制推进器，现在是32单片机解算，不再是arduino

\_thread\_aduino\_reader = new arduino\_reader(\_serial\_arduino, &running, nullptr); // arduino线程

\_pwm\_info = &\_thread\_aduino\_reader->pwm\_info; // pwm信息指针，要传给fsm\_states子线程

\_fsm\_states\_controller = new FSM\_States(&running, \_IMU\_data, \_pwm\_info, nullptr);

\_fsm\_data = \_fsm\_states\_controller->\_fsm\_data; // fsm\_states子线程在，主窗口留一个指针

\_thread\_aduino\_reader->\_serial\_sonar\_sender = \_serial\_sonar; // 声呐串口操作放在arduino线程，不再单开子线程给声呐，要不然贼麻烦

\_thread\_aduino\_reader->\_IMU\_data = \_IMU\_data; // arduino线程，通过\_serial\_sonar\_sender串口指针发送惯导子线程的惯导数据

}

/\*\*

\* @brief MainWindow::open\_serials

\* @return 1表示全部成功打开

\* 是否arduino与GPS串口完全打开以及是否成功打开

\*/

bool MainWindow::**open\_serials**()

{

print\_info = QSerialPortInfo::availablePorts(); // 查看所有可用串口，并将所有串口的串口号和设备描述输出到下方

qDebug() << "[Serial] have found " << print\_info.size() << "serial(s) :";

for(auto &info : print\_info){

qDebug() <<info.portName().append(':').append(info.description()).toStdString().c\_str();

}

for(auto &info : print\_info)

{

//if(strcmp(info.description().toStdString().c\_str(),"MOXA USB Serial Port") == 0)

//if(strcmp(info.description().toStdString().c\_str(),"NovAtel USB Virtual Serial Port") == 0)

if(strcmp(info.description().toStdString().c\_str(),"NovAtel USB Virtual Serial Port") == 0 &&

serial\_pwrpak\_open\_flag == 0 &&

strcmp(info.portName().toStdString().c\_str(),"COM19") == 0)

{// 防止多开串口，usb虚拟串口有3个，打开一个就放弃

qDebug() << "[IMU] open port"

<< info.portName().append(':').append(info.description()).toStdString().c\_str() << "...";

\_serial\_pwrpak->setPort(info);

if(\_serial\_pwrpak->*open*(QIODevice::ReadWrite)) // 读写方式打开串口

{

serial\_pwrpak\_open\_flag = 1;

qDebug() << "[IMU] trying to open IMU at set baud rate" << "115200 ...";

\_serial\_pwrpak->setDataBits(QSerialPort::Data8); // 设置数据位8

\_serial\_pwrpak->setParity(QSerialPort::NoParity);

\_serial\_pwrpak->setStopBits(QSerialPort::OneStop); // 停止位设置为1

\_serial\_pwrpak->setFlowControl(QSerialPort::NoFlowControl); // 设置为无流控制

\_serial\_pwrpak->setDataTerminalReady(true);

\_serial\_pwrpak->setBaudRate(QSerialPort::Baud115200); // 不同波特率下清空接收区

\_serial\_pwrpak->write("UNLOGALL\r\n"); // 发命令关闭输出

\_serial\_pwrpak->setBaudRate(QSerialPort::Baud9600);

\_serial\_pwrpak->write("UNLOGALL\r\n"); // 全方位关输出

\_serial\_pwrpak->setBaudRate(QSerialPort::Baud115200);

std::string read\_buf\_clear;

read\_buf\_clear = read\_serial\_data(\_serial\_pwrpak);

\_serial\_pwrpak->write("send dyz\r\n"); // 测试语句

Sleep(500);

read\_buf\_clear = "";

read\_buf\_clear = read\_serial\_data(\_serial\_pwrpak);

// 惯导的串口会接收到“send dyz”命令，并向控制主机回复“dyz”，之后可以读取回复的数据判断是否成功打开串口

// 但是在后续的测试中，貌似并不是很好用，所以从此处往下直到 FLAG:惯导串口打开完成 处，都作用不是很大

std::string read\_buf = "";

if(read\_buf\_clear.substr(0, 3) == "dyz")

{

qDebug() << "[IMU] have recived "<< read\_buf\_clear.substr(0, 3).c\_str() <<

endl << "[IMU] now buad rate is 115200!";

}

else{

\_serial\_pwrpak->setBaudRate(QSerialPort::Baud9600);

read\_buf\_clear = read\_serial\_data(\_serial\_pwrpak);

//qDebug() << "[dyz]" << read\_buf\_clear.c\_str();

qDebug() << "[IMU] trying to open IMU with setting baud" << "9600 ...";

read\_buf = "";

\_serial\_pwrpak->write("send dyz\r");

Sleep(500);

read\_buf = read\_serial\_data(\_serial\_pwrpak);

if(read\_buf.size() >= 3 && read\_buf.substr(0, 3) == "dyz")

{

qDebug() << "[IMU] baudrate is 9600, have changed to buad 9600";

serial\_pwrpak\_open\_flag = 1;

}

}

}

} // FLAG:惯导串口打开完成

else if (strcmp(info.description().toStdString().c\_str(),"USB-SERIAL CH340") == 0 && // USB-SERIAL CH340

strcmp(info.portName().toStdString().c\_str(),"COM22") == 0) { // 打开 arduino串口（实际现在是32）

qDebug() << "[Arduino] open port"

<< info.portName().append(':').append(info.description()).toStdString().c\_str() << "...";

\_serial\_arduino->setPort(info);

if(\_serial\_arduino->*open*(QIODevice::ReadWrite)) // 读写方式打开串口

{

//\_serial\_arduino->setBaudRate(QSerialPort::Baud115200);

\_serial\_arduino->setBaudRate(QSerialPort::Baud9600);

// 32的串口吞吐量有可能是跟不上，实际测试下来115200波特率有影响，9600刚好

qDebug() << "[Arduino] have opend Arduino serial with setting baud 9600";

\_serial\_arduino->setDataBits(QSerialPort::Data8); //设置数据位8

\_serial\_arduino->setParity(QSerialPort::NoParity);

\_serial\_arduino->setStopBits(QSerialPort::OneStop); //停止位设置为1

\_serial\_arduino->setFlowControl(QSerialPort::NoFlowControl); //设置为无流控制

\_serial\_arduino->setDataTerminalReady(true);

serial\_arduino\_open\_flag = 1;

}

}

else if (strcmp(info.description().toStdString().c\_str(),"Prolific PL2303GT USB Serial COM Port") == 0

&& strcmp(info.portName().toStdString().c\_str(),"COM23") == 0) ///// sonar 串口

{

//qDebug() << "[SERIAL] have a bad serial info";

qDebug() << "[Sonar] open port"

<< info.portName().append(':').append(info.description()).toStdString().c\_str() << "...";

\_serial\_sonar->setPort(info);

if(\_serial\_sonar->*open*(QIODevice::ReadWrite)) // 读写方式打开串口

{

// 声呐的串口用的是9600，因为GGA数据量也是非常短，声呐接收的数据频率也比较低，9600可以说是够用，115200也可以用

//\_serial\_arduino->setBaudRate(QSerialPort::Baud115200);

\_serial\_sonar->setBaudRate(QSerialPort::Baud9600);

qDebug() << "[sonar] have opend sonar serial with setting baud 9600";

\_serial\_sonar->setDataBits(QSerialPort::Data8); //设置数据位8

\_serial\_sonar->setParity(QSerialPort::NoParity);

\_serial\_sonar->setStopBits(QSerialPort::OneStop); //停止位设置为1

\_serial\_sonar->setFlowControl(QSerialPort::NoFlowControl);//设置为无流控制

\_serial\_sonar->setDataTerminalReady(true);

}

}

else{}

} // 所有的可用串口遍历完成

if(serial\_pwrpak\_open\_flag && serial\_arduino\_open\_flag)

return 1; // 打开了全部串口

else {

return 0; // 未完全打开所有串口，

} // 上方的判断只是判断惯导的串口以及32的串口，后来增加的声呐串口也可以再加一个标志位，但没必要

}

/\*\*

\* @brief MainWindow::secur\_flash\_time

\* 定时输出系统时间的定时器，每过10S自动输出系统时间，主要是方便判断程序什么时候出的错误，好判断问题所在

\*/

void MainWindow::**secur\_flash\_time**()

{

QTime main\_time\_tem\_for\_resecu;

main\_time\_tem\_for\_resecu.currentTime().msecsSinceStartOfDay();

qDebug() << "main qDebug time" << main\_time\_tem\_for\_resecu.currentTime().hour()

<< main\_time\_tem\_for\_resecu.currentTime().minute()

<< main\_time\_tem\_for\_resecu.currentTime().second();

}

/\*\*

\* @brief MainWindow::flash\_data

\* 刷新界面的显示状态，最初只是为了显示我需要注意的变量，后来无人船控制需求增加，现在主要有三个作用

\* 1、 原有的功能，显示惯导上读取到的位置信息等

\* 2、 刷新控制模式，一直读取模式状态是遥控模式、定点跟踪、动力定位或者割草机模式，然后向FSM\_states子线程传递状态，

\* 但是只是比较偷懒的方式，只能说在这个控制框架下勉强可以用

\* 3、 和岸基界面结合，收取岸基摇杆控制命令，收取岸基模式选择的网络命令

\*/

void MainWindow::**flash\_data**()

{

INSPVAXA \*\_IMU\_data\_t;

INSPVAXA show\_data\_struct;

\_IMU\_data\_t = &show\_data\_struct;

show\_data\_struct = \*\_IMU\_data;

/\*

double lat\_deg; // 纬度

double long\_deg; // 经度

double height; // 高度

double undulation; // 起伏

double north\_vel; // 北方向速度

double east\_vel; // 东向速度

double up\_vel; // 竖直速度

double roll; // 横滚角度

double pitch; // 俯仰角

double azimuth; // 航向角，机器狗的偏航角度，yaw\*/

std::string show\_state = "";

show\_state.append("INS\_states:↓\n");

show\_state.append(\_IMU\_data\_t->INS\_state);

show\_state.append("\nPOS\_states:↓\n");

show\_state.append(\_IMU\_data\_t->POS\_state);

ui->stateBrowser->setText(show\_state.c\_str());

show\_state = "roll:\t\t";

show\_state.append(std::to\_string(\_IMU\_data\_t->roll));

show\_state.append("\npitch:\t\t");

show\_state.append(std::to\_string(\_IMU\_data\_t->pitch));

show\_state.append("\nyaw:\t\t");

show\_state.append(std::to\_string(\_IMU\_data\_t->azimuth));

show\_state.append("\nlong\_deg:\t\t");

show\_state.append(std::to\_string(\_IMU\_data\_t->long\_deg));

show\_state.append("\nlat\_deg:\t\t");

show\_state.append(std::to\_string(\_IMU\_data\_t->lat\_deg));

show\_state.append("\nheight:\t\t");

show\_state.append(std::to\_string(\_IMU\_data\_t->height));

//show\_state.append("\nlat\_deg:\t\t");

//show\_state.append(std::to\_string(\_IMU\_data\_t->x\_vel));

//show\_state.append("\nheight:\t\t");

//show\_state.append(std::to\_string(\_IMU\_data\_t->y\_vel));

show\_state.append("\nyaw\_set:\t\t");

show\_state.append(std::to\_string(\_fsm\_data->\_remoter\_set->yaw\_set));

//qDebug() << std::to\_string(\_IMU\_data\_t->roll).c\_str();

//show\_state.append("\neast\_error:\t\t");

//show\_state.append(std::to\_string(\_fsm\_data->));

ui->textBrowser->setText("");

ui->textBrowser->setText(show\_state.c\_str());

\_fsm\_data->\_remoter\_set->state\_set = control\_state\_name::REMOTER;

#ifndef control\_box // 判断是否要按照岸基摇杆的命令实现遥控功能，重要变量，此变量在ethernet\_reader.h中定义

\_fsm\_data->\_remoter\_set->forward\_vel\_set = -ui->x\_vel\_Slider->value();

\_fsm\_data->\_remoter\_set->lateral\_vel\_set = -ui->y\_vel\_Slider->value();

\_fsm\_data->\_remoter\_set->rotor\_vel\_set = ui->w\_vel\_Slider->value(); // \*/

#endif

//pianhang = ui->w\_vel\_Slider->value()/50;

//

if(pianhang > 20)

{

pianhang = 0;

}

double set = \_fsm\_data->\_remoter\_set->yaw\_set ;

set = set + pianhang ;

if(set > 360)

set -= 360;

if(set < 0)

set += 360;

\_fsm\_data->\_remoter\_set->yaw\_set = set;

control\_state\_name name\_set = control\_state\_name::REMOTER; // 默认的操作模式是遥控模式，然后刷新检测哪个模式按钮被选中

if(ui->remoter\_radioButton->isChecked())

name\_set = control\_state\_name::REMOTER;

if(ui->stance\_radioButton->isChecked())

name\_set = control\_state\_name::STANDCE;

if(ui->track\_radioButton->isChecked())

name\_set = control\_state\_name::TRACKING; // 切换到对应的模式下

// if(ui->coverage\_radioButton->isChecked())

// name\_set = control\_state\_name::COVERAGE; // 割草机模式比较特殊，

// 现在的操作逻辑是割草机模式调用定点跟踪模式，按理说是割草机模式凌驾于其他模式之上的，所以割草机的选中之后的操作不再这边

\_fsm\_data->\_remoter\_set->state\_set = name\_set; // 将读取到的控制模式更新到fsm\_states子线程中

// 更新按钮选择，选择控制状态以及控制代码

// \_fsm\_data->\_remoter\_set->forward\_vel\_set

// 以下是关于岸基发送网络命令，解析Ethernet\_reader子线程解析之后，对其对应的模式标志位进行更新，然后在主线程中将标志位转换为控制模式

// 随后控制模式由主线程更新到fsm\_states子线程中，完成控制模式的切换，逻辑对，但是在代码框架上这样使用不是很好，直接去掉下边的注释就可用

/\*if(\*\_remoter\_flag){

this->ui->remoter\_radioButton->setChecked(true);

}

if(\*\_stance\_flag){

this->ui->stance\_radioButton->setChecked(true);

}

if(\*\_tracking\_flag){

this->ui->track\_radioButton->setChecked(true);

}

if(\*\_coverage\_flag){

this->on\_coverage\_radioButton\_clicked();

}\*/

}

/\*\*

\* @brief MainWindow::read\_serial\_data

\* @param \_serial\_for\_wait

\* @return 返回值为一个串口的完整反馈

\* 只适用于在设置GPS连续输出之前

\*/

std::string MainWindow::**read\_serial\_data**(QSerialPort \*\_serial\_for\_wait)

{

bool flag;

std::string read\_buf\_clear = "";

do {

flag = \_serial\_for\_wait->*waitForReadyRead*(300);

if(flag)

{

read\_buf\_clear.append(\_serial\_for\_wait->readAll().toStdString());

//qDebug() << read\_buf\_clear.c\_str();

}

}while(flag); // 清空

return read\_buf\_clear;

}

/\*\*

\* @brief MainWindow::write\_serial\_data

\* @param \_serial\_for\_write

\* @param str\_cmd

\* @return 判断反馈信息是否包含OK，确保在设置命令正确，不判断也问题不大（后期试验得到经验）

\*/

bool MainWindow::**write\_serial\_data**(QSerialPort \*\_serial\_for\_write, std::string str\_cmd)

{

\_serial\_for\_write->write(str\_cmd.c\_str());

if(read\_serial\_data(\_serial\_for\_write).find("OK") != std::string::npos)

return true;

else{

return false;

}

}

/\*\*

\* @brief MainWindow::on\_start\_key\_Edit\_editingFinished

\* 打算开发一个键盘快捷键遥控来的，已放弃，滑条操作也挺方便的

\*/

void MainWindow::**on\_start\_key\_Edit\_editingFinished**()

{

qDebug() << ui->start\_key\_Edit->keySequence();

}

/\*\*

\* @brief MainWindow::on\_x\_vel\_spin\_Box\_editingFinished

\* 以下几个相似的函数都是在给滑条限位，以及滑条与输入框的操作同步，

\* 此函数就是：输入完成之后，点击回车键，然后读取输入框的数据，看数据是不是很离谱，对数据限位，最后再把输入的数据同步到与之绑定的滑条上

\*/

void MainWindow::**on\_x\_vel\_spin\_Box\_editingFinished**()

{

int x\_vel\_data = 0;

x\_vel\_data = ui->x\_vel\_spin\_Box->value();

if(x\_vel\_data > 1000)

x\_vel\_data = 1000;

if(x\_vel\_data < -1000)

x\_vel\_data = -1000;

ui->x\_vel\_Slider->setValue(x\_vel\_data);

}

/\*\*

\* @brief MainWindow::on\_x\_vel\_Slider\_sliderMoved

\* @param position

\* 滑条拖动之后产生的事件触发的槽函数，读取滑条位置，然后将数值同步到与之对应的输入框中

\*/

void MainWindow::**on\_x\_vel\_Slider\_sliderMoved**(int position)

{

int x\_vel\_data = 0;

x\_vel\_data = position;

ui->x\_vel\_spin\_Box->setValue(x\_vel\_data);

}

void MainWindow::**on\_y\_vel\_spin\_Box\_editingFinished**()

{

int y\_vel\_data = 0;

y\_vel\_data = ui->y\_vel\_spin\_Box->value();

if(y\_vel\_data > 1000)

y\_vel\_data = 1000;

if(y\_vel\_data < -1000)

y\_vel\_data = -1000;

ui->y\_vel\_Slider->setValue(y\_vel\_data);

}

void MainWindow::**on\_y\_vel\_Slider\_sliderMoved**(int position)

{

int y\_vel\_data = 0;

y\_vel\_data = position;

ui->y\_vel\_spin\_Box->setValue(y\_vel\_data);

}

void MainWindow::**on\_w\_vel\_Slider\_sliderMoved**(int position)

{

int w\_vel\_data = 0;

w\_vel\_data = position;

ui->w\_vel\_spin\_Box->setValue(w\_vel\_data);

}

void MainWindow::**on\_w\_vel\_spin\_Box\_editingFinished**()

{

int w\_vel\_data = 0;

w\_vel\_data = ui->w\_vel\_spin\_Box->value();

if(w\_vel\_data > 1000)

w\_vel\_data = 1000;

if(w\_vel\_data < -1000)

w\_vel\_data = -1000;

ui->w\_vel\_Slider->setValue(w\_vel\_data);

}

/\*\*

\* @brief MainWindow::on\_remoter\_radioButton\_clicked

\* @param checked

\* radioButton按钮组的选中事件触发槽函数，发现按钮自动成组（其中一个选中，其余自动设置为非选中状态），所以此机制为冗余的，

\*/

void MainWindow::**on\_remoter\_radioButton\_clicked**(bool checked)

{

// 点击事件不在这里，在定时器刷新程序中实现

}

void MainWindow::**on\_stance\_radioButton\_clicked**(bool checked)

{

}

void MainWindow::**on\_track\_radioButton\_clicked**(bool checked)

{

}

/\*\*

\* @brief MainWindow::on\_w\_vel\_Slider\_sliderReleased

\* 此函数主要功能是：拐弯的滑条释放之后实现自动归零位，避免拐弯与直行需要同时操作而产生的不便

\* 拐弯的时候，为了操作的方便，等到滑条Released，代表释放滑动，滑条自动归零，保持开环直线航行

\*/

void MainWindow::**on\_w\_vel\_Slider\_sliderReleased**()

{

int w\_vel\_data = 0;

ui->w\_vel\_spin\_Box->setValue(w\_vel\_data);

ui->w\_vel\_Slider->setValue(0);

}

/\*\*

\* @brief MainWindow::process\_for\_GPS

\* 此函数为GPS信息不准确时候，进行重新打开串口的操作

\* 主要流程是：

\* 1、 关闭原有串口

\* 2、 打开惯导的3个虚拟串口中，除了正在使用的串口的另外两个，只要能打开其中一个就可以

\* 3、 打开新串口之后，注销所有订阅消息，并关闭新串口

\* 4、 重新打开之前所用的老串口

\* 5、 如若再次出现错误重复以上过程

\* 例如：惯导虚拟串口有COM18/COM19/COM20，原来使用的是COM19，出错之后，关闭COM19，打开COM20，发送命令“UNLOGALL”，

\* 关闭COM20，重新打开COM19

\*/

void MainWindow::**process\_for\_GPS**()

{

if(\_thread\_pwrpak\_reader->GPS\_broken\_flag)

{

\_serial\_pwrpak->*close*();

QList<QSerialPortInfo> print\_info\_re; // serial\_info中间变量

print\_info\_re = QSerialPortInfo::availablePorts(); // 查看所有串口

for(auto &info : print\_info\_re)

{

if(strcmp(info.description().toStdString().c\_str(),"NovAtel USB Virtual Serial Port") == 0

&& strcmp(info.portName().toStdString().c\_str(),"COM20") == 0) // 防止多开串口，usb虚拟串口有3个，打开一个就放弃

{

\_serial\_pwrpak->setPort(info);

if(\_serial\_pwrpak->*open*(QIODevice::ReadWrite)) // 读写方式打开串口

{

qDebug() << "[IMU] trying to open IMU with setting baud" << "115200 ...";

\_serial\_pwrpak->setDataBits(QSerialPort::Data8); //设置数据位8

\_serial\_pwrpak->setParity(QSerialPort::NoParity);

\_serial\_pwrpak->setStopBits(QSerialPort::OneStop); //停止位设置为1

\_serial\_pwrpak->setFlowControl(QSerialPort::NoFlowControl);//设置为无流控制

\_serial\_pwrpak->setDataTerminalReady(true);

\_serial\_pwrpak->setBaudRate(QSerialPort::Baud115200); // 不同波特率下清空接收区

\_serial\_pwrpak->write("UNLOGALL\r\n"); // 发命令关闭输出

\_serial\_pwrpak->setBaudRate(QSerialPort::Baud9600);

\_serial\_pwrpak->write("UNLOGALL\r\n");

\_serial\_pwrpak->setBaudRate(QSerialPort::Baud115200);

\_serial\_pwrpak->*close*();

}

}

}

print\_info\_re = QSerialPortInfo::availablePorts(); // 查看所有串口

for(auto &info : print\_info\_re)

{

if(strcmp(info.description().toStdString().c\_str(),"NovAtel USB Virtual Serial Port") == 0

&& strcmp(info.portName().toStdString().c\_str(),"COM19") == 0)

{

\_serial\_pwrpak->setPort(info);

if(\_serial\_pwrpak->*open*(QIODevice::ReadWrite)) // 读写方式打开串口

{

qDebug() << "[IMU] trying to open IMU with setting baud" << "115200 ...";

\_serial\_pwrpak->setDataBits(QSerialPort::Data8); //设置数据位8

\_serial\_pwrpak->setParity(QSerialPort::NoParity);

\_serial\_pwrpak->setStopBits(QSerialPort::OneStop); //停止位设置为1

\_serial\_pwrpak->setFlowControl(QSerialPort::NoFlowControl); //设置为无流控制

\_serial\_pwrpak->setDataTerminalReady(true);

\_serial\_pwrpak->setBaudRate(QSerialPort::Baud115200); // 不同波特率下清空接收区

\_serial\_pwrpak->write("LOG INSPVAXA ONTIME 0.05\r\n"); // 重新订阅控制使用的消息

\_serial\_pwrpak->write("LOG GPGGA ONTIME 0.05\r\n"); // 重新订阅GGA给雷达，容易忘记！！！

\_serial\_pwrpak->write("GENERATERTKCORRECTIONS rtcmv3 com2\r\n");

}

}

}

QTime time\_tem\_for\_resecu;

\_thread\_pwrpak\_reader->IMU\_save\_data.last\_read\_time =

time\_tem\_for\_resecu.currentTime().msecsSinceStartOfDay(); // （“看门狗定时器”貌似以弃用，貌似）

\_thread\_pwrpak\_reader->GPS\_broken\_flag = 0; // 清除报错标志位

}

}

/\*\*

\* @brief MainWindow::on\_open\_loop\_checkBox\_clicked

\* @param checked

\* 遥控直线航行的开环/闭环切换框，选中状态下checked为TRUE，代表开环行驶

\*/

void MainWindow::**on\_open\_loop\_checkBox\_clicked**(bool checked)

{

#ifdef model\_switch // 此宏定义控制着能否实现开闭环，主要是留给岸基的接口，岸基的摇杆遥控下，就不用开闭环切换，直接注销掉宏定义即可，方便

static pthread\_mutex\_t read\_IMU\_data\_lock = PTHREAD\_MUTEX\_INITIALIZER; // 线程地址锁，防止变量传递时候，各线程同时传递产生冲突，有一说一，不要也罢

pthread\_mutex\_lock(&read\_IMU\_data\_lock);

\_fsm\_states\_controller->\_fsm\_data->\_remoter\_set->yaw\_set = \_IMU\_data->azimuth;

\_fsm\_states\_controller->open\_loop\_flag = checked;

pthread\_mutex\_unlock(&read\_IMU\_data\_lock);

qDebug() << "[REMOTER OPEN LOOP FLAG] open loop flag : " << checked;

#else

qDebug() << "fff";

#endif

}

/\*\*

\* @brief MainWindow::on\_motor\_start\_clicked

\* motor\_strat 和 motort\_stop 按钮的槽函数

\* 推进器上电之后的软件保险，点击 motor\_strat 之后打开软件锁，船才能动，双保险

\*/

void MainWindow::**on\_motor\_start\_clicked**()

{

\_pwm\_info->start\_flag = true;

}

/\*\*

\* @brief MainWindow::on\_motor\_stop\_clicked

\* motort\_stop 点击之后船立马停动力，紧急刹车的手段，之前船经常失控加的

\*/

void MainWindow::**on\_motor\_stop\_clicked**()

{

\_pwm\_info->start\_flag = false;

}

/\*\*

\* @brief MainWindow::on\_sonar\_Button\_clicked

\* 声呐的测试按钮 sonar 按钮槽函数，最开始测试传递信息给声呐的，发送程序已经融合进去 arduino\_reader 线程中，现在这个函数没用了

\*/

void MainWindow::**on\_sonar\_Button\_clicked**()

{

//\_serial\_sonar->write(\_IMU\_data->GGA\_string\_reader.c\_str());

\_thread\_aduino\_reader->\_serial\_sonar\_sender = \_serial\_sonar;

\_thread\_aduino\_reader->\_IMU\_data = \_IMU\_data;

}

/\*\*

\* @brief MainWindow::on\_coverage\_radioButton\_clicked

\* 割草机模式的 coverage\_radioButton 点击事件槽函数

\*/

void MainWindow::**on\_coverage\_radioButton\_clicked**()

{

// 解析输入框中的数据，文本转成数字，然后计算多个点位，组成航行轨迹

QString str = ui->viewpoint\_edit->toPlainText();

QStringList list = str.split('\n');

if(list.size() == 5) // 如果数据输入格式不正确的话，非常容易产生BUG，正常我定义的输入应该是5行的

{

\_fsm\_states\_controller->\_fsm\_data->coverage\_point[0][0] = list[0].toDouble();

\_fsm\_states\_controller->\_fsm\_data->coverage\_point[0][1] = list[1].toDouble();

\_fsm\_states\_controller->\_fsm\_data->coverage\_point[1][0] = list[2].toDouble();

\_fsm\_states\_controller->\_fsm\_data->coverage\_point[1][1] = list[3].toDouble();

\_fsm\_states\_controller->\_fsm\_data->coverage\_num = list[4].toInt();

\_fsm\_states\_controller->\_fsm\_data->coverage\_first\_run\_flag = true;

qDebug() << "read double data:" << QString::number(

\_fsm\_states\_controller->\_fsm\_data->coverage\_point[0][0],

10,7);

printf("%10f", \_fsm\_states\_controller->\_fsm\_data->coverage\_point[0][0]);

\_fsm\_states\_controller->\_fsm\_data->\_remoter\_set->state\_set = control\_state\_name::COVERAGE;

\_fsm\_data->\_remoter\_set->state\_set = control\_state\_name::COVERAGE;

// on\_test\_Button\_clicked();

this->ui->track\_radioButton->setChecked(true);

\*\_coverage\_flag = false;

}else {

qDebug() << "[MAINWINDOW COVERAGE] input data error";

}

}