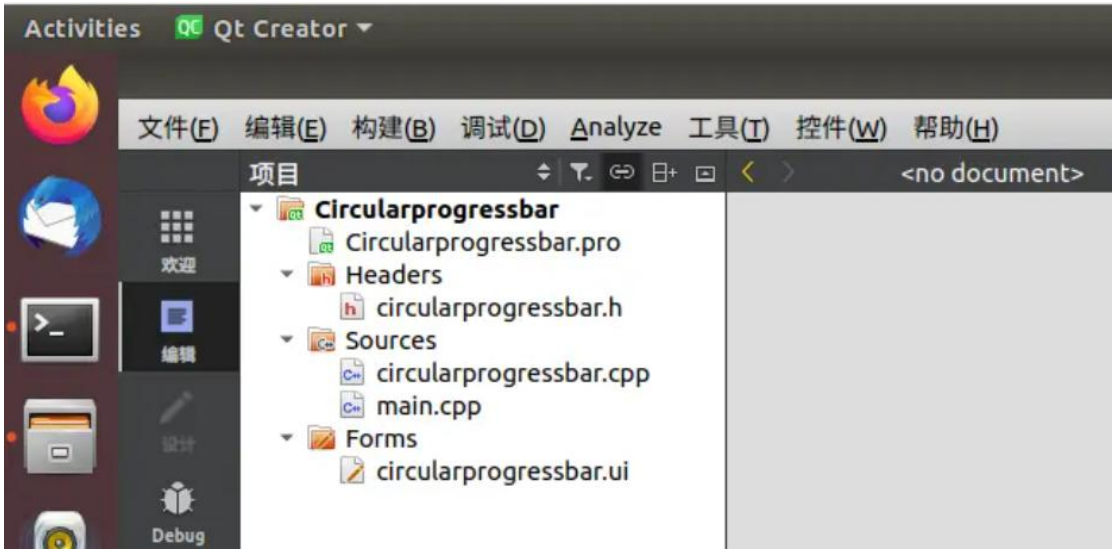
《嵌入式系统》第三次作业

基于 Qt（Quick time）两次实验项目，参考 Qt 指导手册，完成下述任务：

1. 针对第一次 Qt 实验中的下述工程：

   a) 自定义控件

   代码的项目结构如下图所示



   各个文件的类型和核心作用如下

   Circularprogressbar.pro：项目配置文件，定义工程依赖、模块（如 widgets）、文件路径，是 Qt 编译的入口配置；

   headers/circularprogressbar.h：头文件，声明自定义控件类 Circularprogressbar、成员函数、成员变量、信号槽；

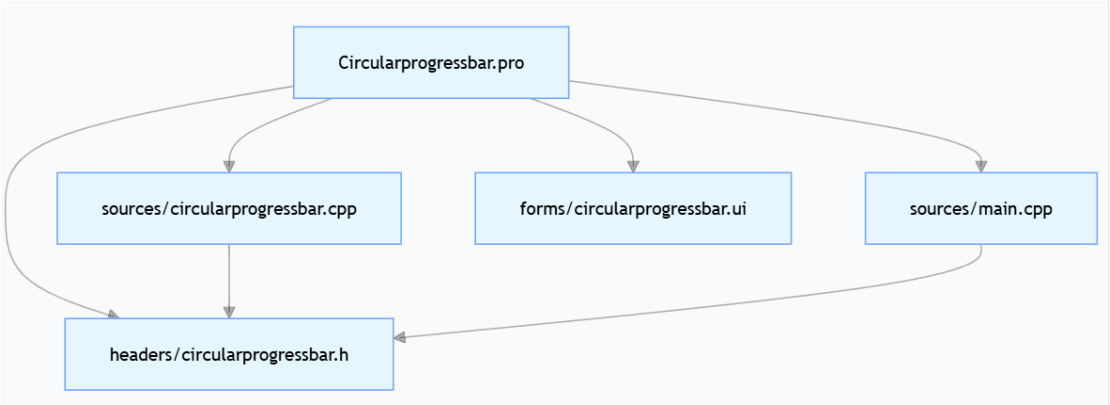
   sources/circularprogressbar.cpp：实现文件，实现头文件声明的所有类方法、绘制逻辑、事件处理、槽函数；

   sources/main.cpp：程序入口文件，创建 QApplication 实例（Qt 应用程序核心）和自定义控件实例，启动应用事件循环；

   forms/circularprogressbar.ui：UI 设计文件，本项目未实际使用（控件通过代码纯绘

制实现），仅为项目结构默认文件

文件依赖关系图：



.pro 文件是工程核心配置，指定所有参与编译的头文件、源文件和 UI 文件，同时依赖 Qt 的 widgets 模块（支持桌面端图形界面）。

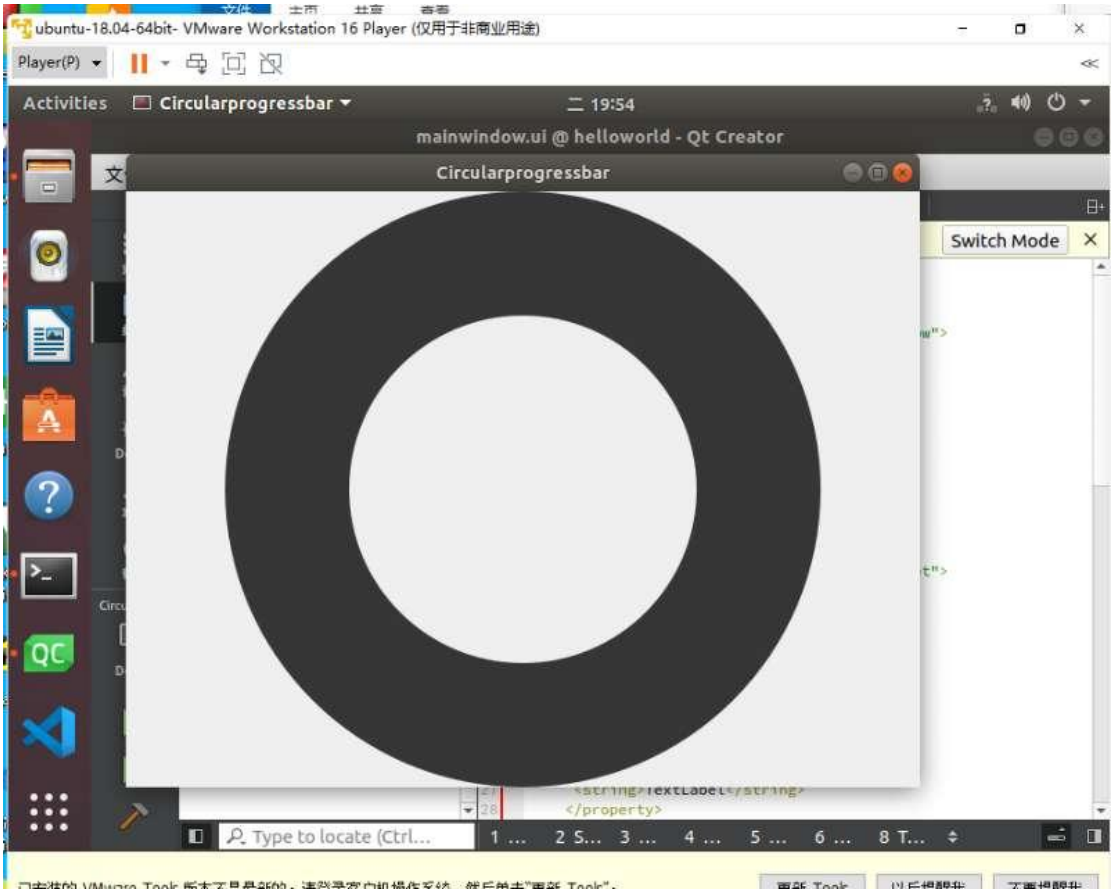
circularprogressbar.cpp 必须包含 circularprogressbar.h，才能实现头文件中声明的类和函数。

main.cpp 必须包含 circularprogressbar.h，才能创建自定义控件 Circularprogressbar 的实例。

circularprogressbar.ui 未被代码引用（本控件通过 paintEvent 纯代码绘制，未使用 Qt Designer 拖拽界面），仅为项目结构占位文件。

工作原理：

实验现象



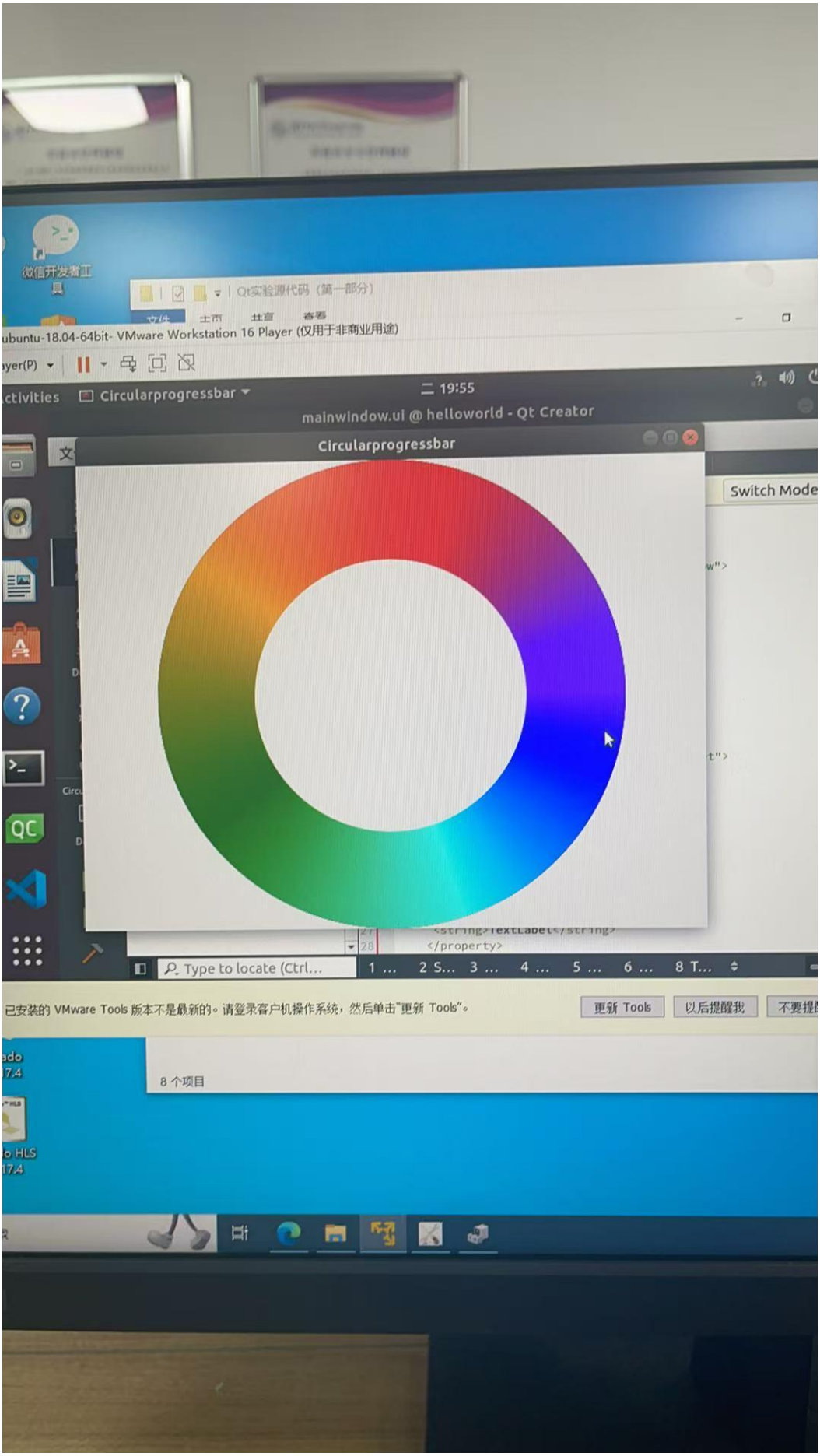
运行程序后，将显示一个中心对齐的圆形控件

初始状态：控件由 “暗灰色外圈大圆”“透明中间环”“窗口背景色内圈小圆” 组成，无彩色进度显示。

按下空格键：中间环开始以顺时针方向填充彩色渐变 (深紫→红→橙→浅绿→青→蓝→深紫)，进度每 10ms 增加 3.6 度（1 秒填满 360 度)。

松开空格键：彩色进度以每 ms 1 度的速度减少，直至进度为 0 时，定时器停止，彩色环完全消失，回归初始状态。

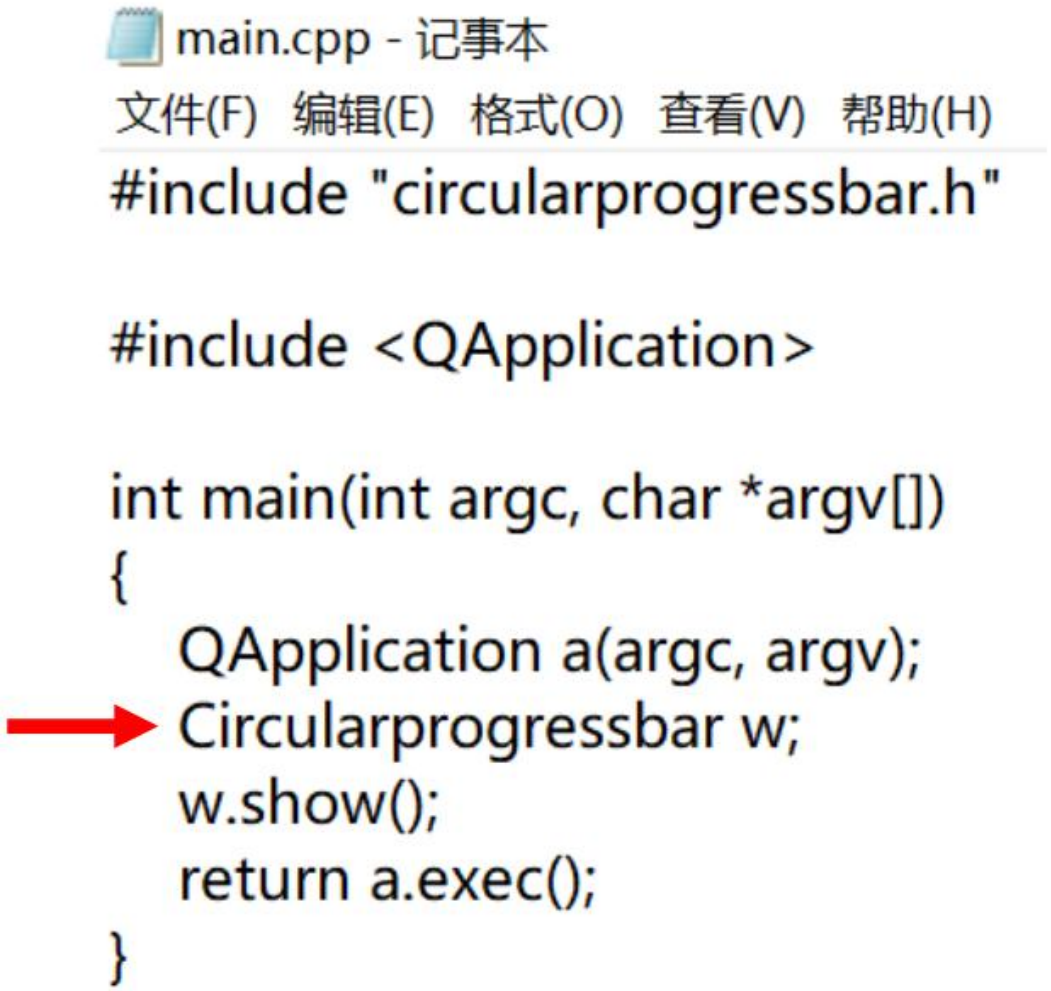
进度上限：彩色环最多填充 360 度 (整圈)，达到后不再增加；下限为 0 度 (无填充)。


微信开发者工具
Qt实验源代码（第一部分）
ubuntu-18.04-64bit- VMware Workstation 16 Player (仅用于非商业用途)
Circularprogressbar
19:55
mainwindow.ui @ helloworld - Qt Creator
Circularprogressbar
Switch Mode
Type to locate (Ctrl...
已安装的 VMware Tools 版本不是最新的。请登录客户机操作系统，然后单击"更新 Tools"。
更新 Tools
以后提醒我
8 个项目

核心工作流程：

（1）初始化流程

程序启动：main.cpp 中创建 QApplication（Qt 应用程序必备核心对象，负责事件循环）和 Circularprogressbar 实例 w，调用 w.show()显示控件。



控件初始化：Circularprogressbar 构造函数中创建 QTimer（定时器），并连接定时器的 timeout 信号到槽函数 decreaseColorProgress（定时器触发时更新进度）。

```
public slots:
    void decreaseColorProgress();

private:
    QTimer *myTimer;
    bool direction = false;
    int currentColorProgress = 0;
};
```

（2）按键事件处理流程

按键按下（keyPressEvent）：触发事件 按下空格键（Qt::Key_Space），1. 启动定时器（默认按系统最小间隔触发，约 10ms / 次）；

2. 设置 direction = true（标记进度 “增加” 模式）；

3. 调用 update()触发 paintEvent 重绘控件。

```
void Circularprogressbar::keyPressEvent(QKeyEvent *event)
{
    if(event->key() == Qt::Key_Space)
    {
        myTimer->start();
        direction = true;
        // 更新控件的颜色
        update();
    }
}
```

按键释放（keyReleaseEvent）：触发事件：松开空格键，1. 设置 direction = false（标记进度 “减少” 模式）；

2. 调用 update()触发 paintEvent 重绘控件。

```cpp
void Circularprogressbar::keyReleaseEvent(QKeyEvent *event)
{
    if(event->key() == Qt::Key_Space)
    {
        direction = false;
        // 更新控件的颜色
        update();
    }
}
```

（3）进度更新流程（槽函数 decreaseColorProgress）

定时器每次触发时执行，核心逻辑：

```cpp
void Circularprogressbar::decreaseColorProgress()
{
    if(direction){
        // 增加颜色进度
        currentColorProgress += 3.6;
        if(currentColorProgress > 360){
            currentColorProgress = 360;
        }
    }else{
        // 减少颜色进度
        currentColorProgress -= 1;
        if(currentColorProgress < 0){
            currentColorProgress = 0;
            myTimer->stop();
        }
    }
    // 更新控件的颜色
    update();
}
```

（4）绘制流程（paintEvent 核心函数）

控件重绘时自动调用，流程如下：

画家初始化：创建 QPainter（Qt 绘图工具），将绘图原点平移到窗口中心（确保圆形居中），开启抗锯齿（Antialiasing），设置无画笔（Qt::NoPen，仅填充颜色，无轮廓

线)。

计算半径：取窗口宽高的最小值的一半作为外圈半径（确保圆形不拉伸），内圈半径 = 外圈半径 - 50（形成环形间隙)。

分层绘制：

第一层：调用 drawBiggestCircle 绘制外圈大圆（颜色为暗灰色#363636)。

第二层：调用 drawColor 绘制彩色进度环（核心层)：

使用 QConicalGradient (锥形渐变) 定义颜色分布 (从 0 度到 360 度的渐变序列)。

用 drawPie 绘制扇形进度：rect 为外圈圆形的外接矩形，startAngle = -180*16（起始角度为顺时针 180 度，即 9 点钟方向），spanAngle = -(currentColorProgress*16) (顺时针跨越 currentColorProgress 度，Qt 中 drawPie 角度单位为 1/16 度)。

第三层：调用 drawLittleCircle 绘制内圈小圆（颜色为窗口背景色)，遮挡中间区域，形成 “环形进度条” 效果。

```
void Circularprogressbar::paintEvent(QPaintEvent *event) {
    Q_UNUSED(event);//忽略event
    QPainter painter(this);
    int width = this->width();
    int height = this->height();
    painter.translate(width / 2, height / 2); // 将画布中心移动到窗口中心
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.setPen(Qt::NoPen);

    int outerRadius = std::min(width, height) / 2; // 外圈半径
    int innerRadius = outerRadius - 50; // 内圈半径

    // 绘制外圈大圆
    drawBiggestCircle(painter, outerRadius);

    drawColor(painter, outerRadius);

    // 绘制内圈小圆
    drawLittleCircle(painter, innerRadius);
}
```

（5）颜色渐变逻辑

drawColor 中使用 QConicalGradient (锥形渐变，以原点为中心，按角度分布颜色)，

```
void Circularprogressbar::drawColor(QPainter &painter, int radius)
{
    QRect rect(-radius,-radius,2*radius,2*radius);
    QConicalGradient Conical(0, 0, 0);

    // 在0点处设置与0.05点相似的颜色
    Conical.setColorAt(0, QColor(128, 0, 255)); // 深紫色
    Conical.setColorAt(0.05, QColor(128, 0, 255)); // 深紫色

    // 其他颜色保持不变
    Conical.setColorAt(0.2, QColor(255, 0, 0));     // 红色
    Conical.setColorAt(0.4, QColor(255, 165, 0));   // 橙色
    Conical.setColorAt(0.6, QColor(0, 128, 0));     // 浅绿色
    Conical.setColorAt(0.8, QColor(0, 255, 255));   // 青色

    // 在接近1的位置复制0点的颜色以完成闭环
    Conical.setColorAt(0.95, QColor(0, 0, 255));    // 蓝色
    Conical.setColorAt(1.0, QColor(128, 0, 255));   // 深紫色


    painter.setBrush(Conical);
    painter.drawPie(rect, -180 * 16, -(currentColorProgress * 16));
}
```

核心.cpp 文件代码注解如下

circularprogressbar.cpp 文件

```
#include "circularprogressbar.h"

// 构造函数：初始化控件核心资源
Circularprogressbar::Circularprogressbar(QWidget* parent)
    : QWidget(parent)  // 继承QWidget，指定父控件（默认nullptr，无父控件）
{
    myTimer = new QTimer(this);  // 创建定时器，父控件设为this（自动随控件销毁，避免内存泄漏）
    // 连接定时器的timeout信号到槽函数decreaseColorProgress：定时器触发时更新进度
    connect(myTimer, &QTimer::timeout, this, &Circularprogressbar::decreaseColorProgress);
}

// 绘制外圈大圆（暗灰色背景环）
void Circularprogressbar::drawBiggestCircle(QPainter& painter, int radius) {
    painter.save();  // 保存当前画家状态（如平移、画笔设置），后续可通过restore()恢复
```

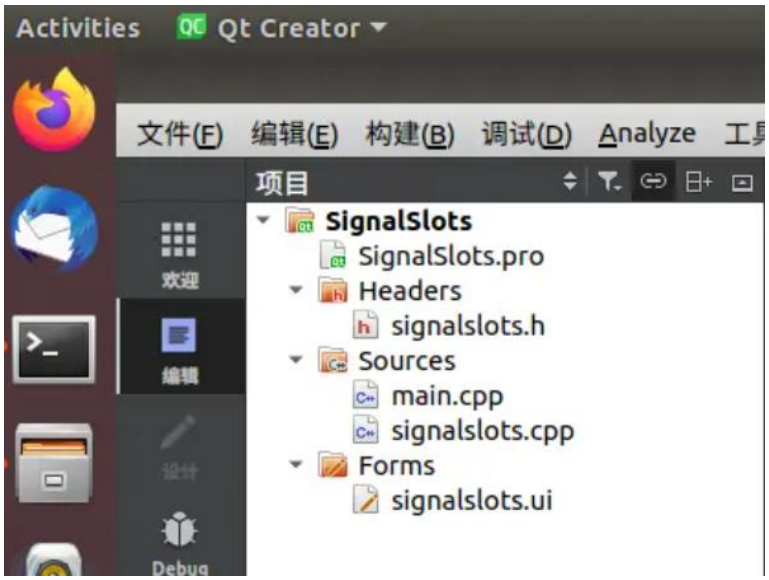
```
    QPainterPath path;  // 路径对象：用于绘制复杂图形（此处绘制圆形）
    // 添加椭圆（圆形是椭圆的特殊情况）：左上角(-radius,-radius)，宽高2*radius（以原点为中心）
    path.addEllipse(-radius, -radius, 2 * radius, 2 * radius);
    painter.setBrush(QColor(54, 54, 54));  // 设置填充色为暗灰色（RGB：54,54,54）
    painter.drawPath(path);  // 绘制路径（填充暗灰色圆形）
    painter.restore();  // 恢复画家之前的状态，避免影响后续绘制
}

// 绘制内圈小圆（窗口背景色，用于形成环形间隙）
void Circularprogressbar::drawLittleCircle(QPainter& painter, int radius) {
    painter.save();
    QPainterPath path;
    // 内圈半径=传入的radius（已在外层计算为外圈半径-50），绘制中心对齐的小圆
    path.addEllipse(-radius, -radius, 2 * radius, 2 * radius);
    // 获取控件的窗口背景色（与父控件背景色一致，确保视觉统一）
    QColor ringColor = palette().color(QPalette::Window);
    painter.setBrush(ringColor);  // 设置填充色为窗口背景色
    painter.drawPath(path);  // 绘制内圈小圆，遮挡中间区域
    painter.restore();
}

// 绘制彩色进度环（核心绘制函数）
void Circularprogressbar::drawColor(QPainter& painter, int radius)
{
    // 定义彩色环的外接矩形：以原点为中心，宽高2*radius（与外圈大圆尺寸一致）
    QRect rect(-radius, -radius, 2 * radius, 2 * radius);
    // 创建锥形渐变对象：中心(0,0)，起始角度0度（3点钟方向）
    QConicalGradient Conical(0, 0, 0);

    // 配置渐变颜色节点（按角度占比设置，0→1对应0度→360度）
    Conical.setColorAt(0, QColor(128, 0, 255));    // 0度：深紫色
    Conical.setColorAt(0.05, QColor(128, 0, 255)); // 18度：深紫色（过渡，避免突变）
    Conical.setColorAt(0.2, QColor(255, 0, 0));    // 72度：红色
    Conical.setColorAt(0.4, QColor(255, 165, 0));  // 144度：橙色
    Conical.setColorAt(0.6, QColor(0, 128, 0));    // 216度：浅绿色
    Conical.setColorAt(0.8, QColor(0, 255, 255));  // 288度：青色
    Conical.setColorAt(0.95, QColor(0, 0, 255));   // 342度：蓝色
    Conical.setColorAt(1.0, QColor(128, 0, 255));  // 360度：深紫色（闭环，与起始色一致）

    painter.setBrush(Conical);  // 设置填充色为锥形渐变
    // 绘制扇形进度环：
    // startAngle：-180*16（顺时针180度，即9点钟方向，作为进度起始点）
```

```
    // spanAngle：-(currentColorProgress*16)（顺时针跨越currentColorProgress度，负号表示顺时针）
    painter.drawPie(rect, -180 * 16, -(currentColorProgress * 16));
}

// 重写paintEvent：控件重绘时自动调用，定义绘制流程
void Circularprogressbar::paintEvent(QPaintEvent* event) {
    Q_UNUSED(event);  // 标记event未使用，避免编译器警告
    QPainter painter(this);  // 创建画家对象，绘图设备为当前控件
    int width = this->width();  // 获取控件当前宽度
    int height = this->height();// 获取控件当前高度

    // 关键：将画家原点平移到控件中心（后续绘图以中心为原点，方便圆形居中）
    painter.translate(width / 2, height / 2);
    painter.setRenderHint(QPainter::Antialiasing, true);  // 开启抗锯齿，使图形边缘平滑
    painter.setPen(Qt::NoPen);  // 取消画笔（仅填充颜色，不绘制轮廓线）

    int outerRadius = std::min(width, height) / 2;  // 外圈半径：取宽高最小值的一半（避免圆形拉伸）
    int innerRadius = outerRadius - 50;  // 内圈半径：外圈半径减50，形成环形间隙

    drawBiggestCircle(painter, outerRadius);  // 第一层：绘制外圈暗灰色大圆
    drawColor(painter, outerRadius);          // 第二层：绘制彩色进度环（覆盖在外圈内侧）
    drawLittleCircle(painter, innerRadius);   // 第三层：绘制内圈背景色小圆（遮挡中间）
}

// 重写keyPressEvent：处理键盘按下事件
void Circularprogressbar::keyPressEvent(QKeyEvent* event)
{
    // 判断是否按下空格键
    if (event->key() == Qt::Key_Space)
    {
        myTimer->start();  // 启动定时器，开始触发进度更新
        direction = true;  // 标记为“进度增加”模式
        update();  // 触发paintEvent重绘控件（立即更新进度显示）
    }
}

// 重写keyReleaseEvent：处理键盘释放事件
void Circularprogressbar::keyReleaseEvent(QKeyEvent* event)
{
    // 判断是否松开空格键
    if (event->key() == Qt::Key_Space)
```

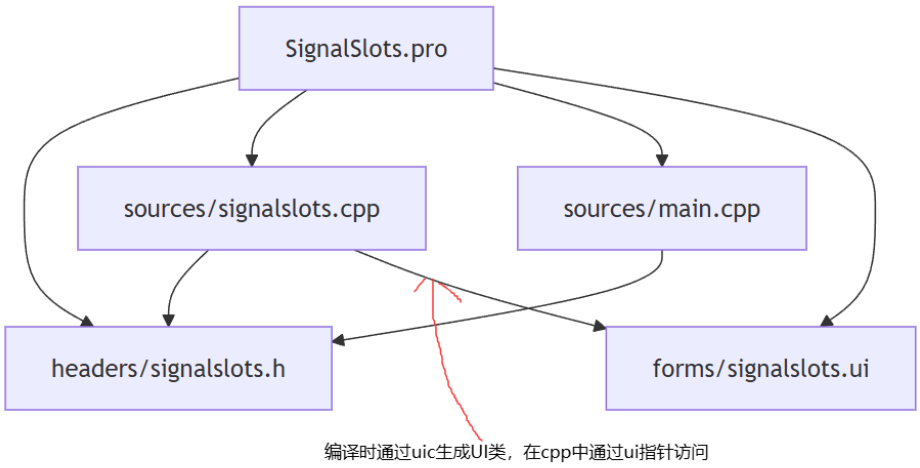
```
    {
        direction = false;  // 标记为“进度减少”模式
        update();  // 触发paintEvent重绘控件（立即更新进度显示）
    }
}

// 槽函数：定时器触发时更新进度（核心进度控制逻辑）
void Circularprogressbar::decreaseColorProgress()
{
    if (direction) {  // 进度增加模式（按下空格键）
        currentColorProgress += 3.6;  // 每次触发增加3.6度（约每秒36次触发，实际每秒增加3.6度）
        if (currentColorProgress > 360) {  // 进度上限：360度（整圈）
            currentColorProgress = 360;
        }
    }
    else {  // 进度减少模式（松开空格键）
        currentColorProgress -= 1;  // 每次触发减少1度（每秒约100次触发，实际每秒减少1度）
        if (currentColorProgress < 0) {  // 进度下限：0度（无填充）
            currentColorProgress = 0;
            myTimer->stop();  // 进度为0时停止定时器，避免无效触发
        }
    }
    update();  // 每次进度更新后，触发重绘，显示最新进度
}

// 析构函数：销毁控件时释放资源（本项目中myTimer父控件为this，会自动销毁，无需手动释放）
Circularprogressbar::~Circularprogressbar()
{
    }
```

b) 信号与槽的应用（秒表）

代码的项目结构如下图所示



文件依赖关系图



依赖逻辑说明：

.pro 文件统一管理工程文件，指定 QT += widgets（依赖 Qt 桌面图形控件模块），确保编译时能找到 QDialog、QPushButton 等控件类。

signalslots.cpp 必须包含 signalslots.h，才能访问类声明的 UI 指针和槽函数；同时依赖 signalslots.ui 生成的 UI 类 (编译时自动生成 ui_signalslots.h，内含界面元素的访问接口)。

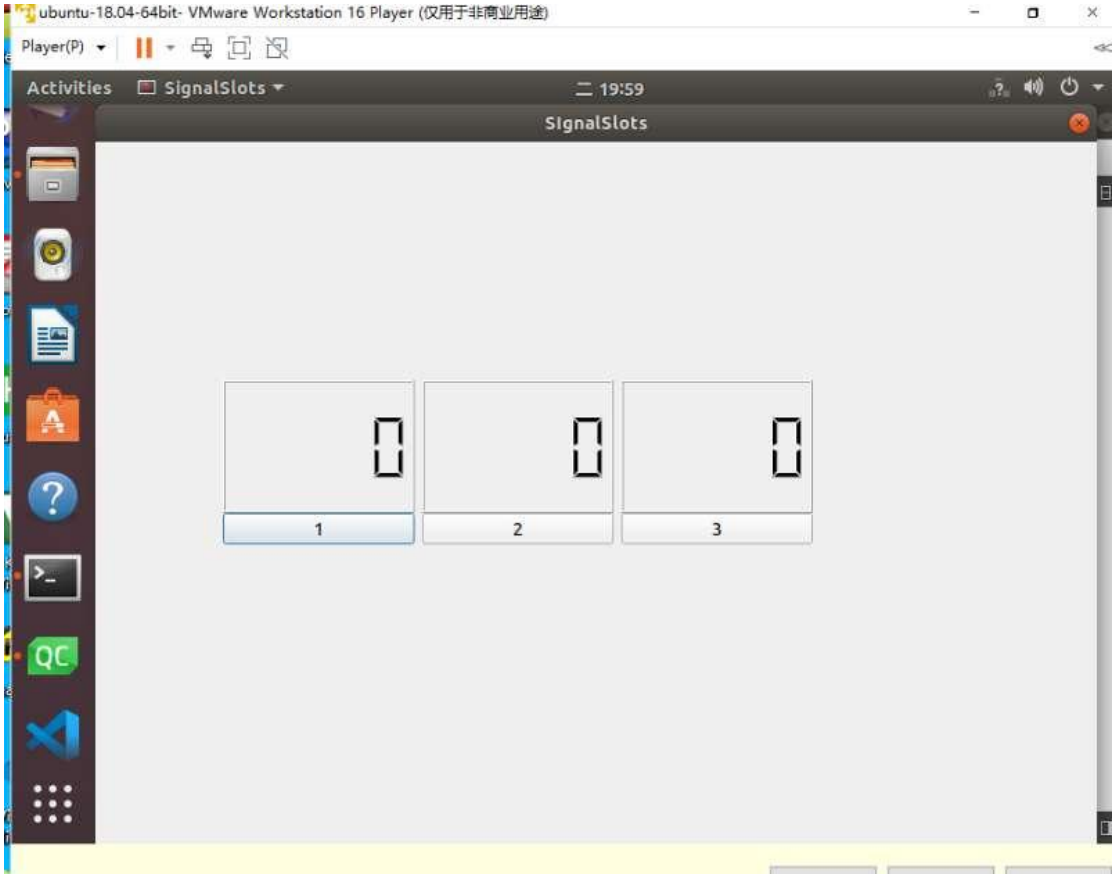
main.cpp 需包含 signalslots.h，才能创建 SIgnalSlots 窗口实例并显示。

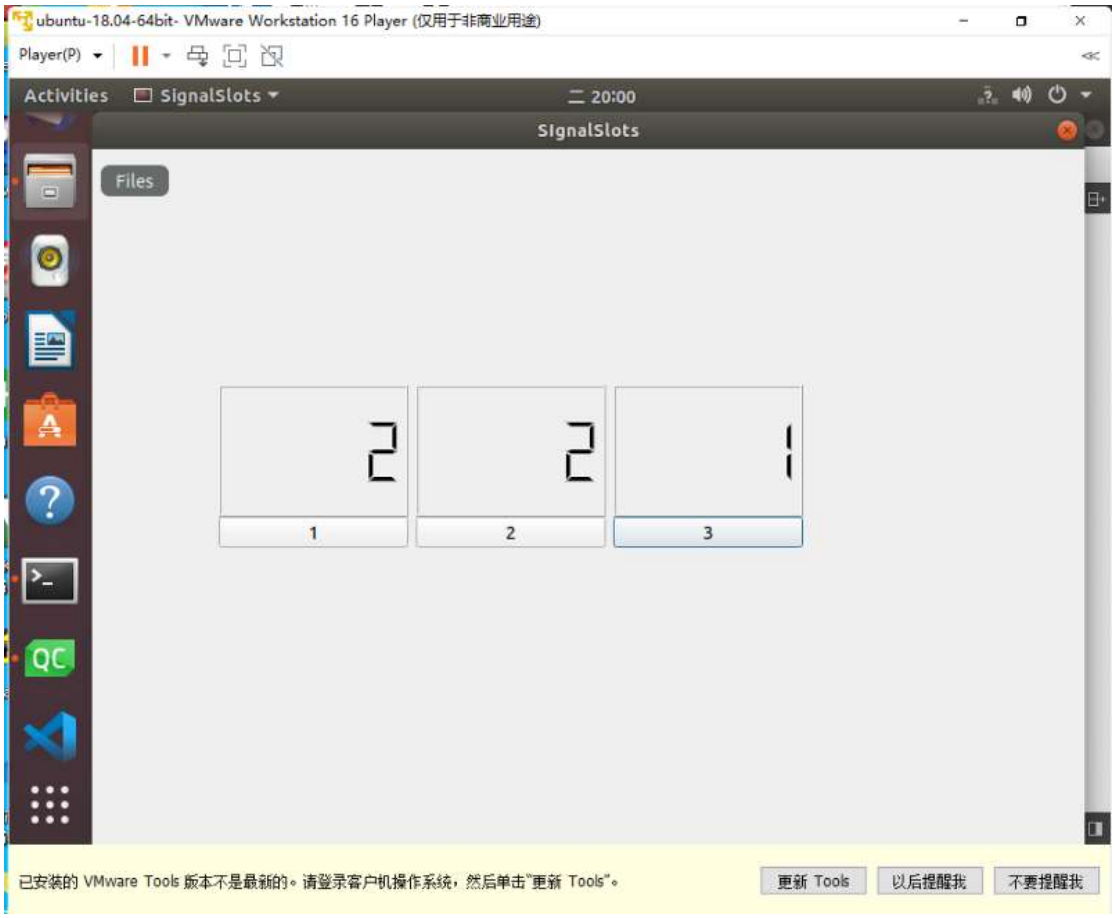
signalslots.ui 是可视化设计文件，无需手动修改代码，编译时由 Qt 工具链自动转换

为 C++ 代码，供 signalslots.cpp 通过 ui->访问界面控件。

工作原理分析

实验现象如下





点击按钮 “1”：仅 LCD1 的数字加 1（从 0→1→2… 循环递增）。

点击按钮 “2”：仅 LCD2 的数字加 1（从 0→1→2… 循环递增）。

点击按钮 “3”：LCD1、LCD2、LCD3 的数字同时加 1（三个显示器同步递增）。

核心工作流程拆解

（1）初始化流程

程序启动：main.cpp 中创建 QApplication 实例（负责管理 Qt 应用的事件循环、资源分配），创建 SIgnalSlots 窗口实例 w，调用 w.show()显示窗口。

```cpp
#include "signalslots.h"

#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    SIgnalSlots w;
    w.show();
    return a.exec();
}
```

窗口初始化：SIgnalSlots 构造函数执行核心操作：

ui->setupUi(this)：初始化 UI 界面 (加载 signalslots.ui 设计的控件，创建按钮、LCD 显示器并布局)。

信号槽连接：通过 connect 函数绑定按钮的 clicked 信号（按钮被点击时发射）与对应的槽函数（处理计数逻辑)。

```cpp
SIgnalSlots::SIgnalSlots(QWidget *parent)
    : QDialog(parent)
    , ui(new Ui::SIgnalSlots)
{
    ui->setupUi(this);
    //按钮1和LcdNumber1建立信号槽连接
    connect(ui->pushButton_1,SIGNAL(clicked()),this,SLOT(pushbutten_1_Slot()));
    //按钮2和LcdNumber2建立信号槽连接
    connect(ui->pushButton_2,SIGNAL(clicked()),this,SLOT(pushbutten_2_Slot()));
    //按钮3和LcdNumber1.2.3建立信号槽连接
    connect(ui->pushButton_3,SIGNAL(clicked()),this,SLOT(pushbutten_1_Slot()));
    connect(ui->pushButton_3,SIGNAL(clicked()),this,SLOT(pushbutten_2_Slot()));
    connect(ui->pushButton_3,SIGNAL(clicked()),this,SLOT(pushbutten_3_Slot()));
}
```

（2）信号槽连接逻辑（核心设计）

Qt 的信号与槽是事件驱动的核心机制，本工程的连接关系如下：

信号发送者（Sender）　信号（Signal）　信号接收者（Receiver）　槽函数 (Slot)

功能

ui->pushButton_1（按钮 1） clicked()（点击信号）this （当前窗口实例）

pushbutten_1_Slot() 触发 LCD1 计数 + 1

ui->pushButton_2（按钮 2） clicked()（点击信号）this （当前窗口实例）

pushbutten_2_Slot() 触发 LCD2 计数 + 1

ui->pushButton_3（按钮 3） clicked()（点击信号）this （当前窗口实例）

pushbutten_1_Slot() 触发 LCD1 计数 + 1

ui->pushButton_3（按钮 3） clicked()（点击信号）this （当前窗口实例）

pushbutten_2_Slot() 触发 LCD2 计数 + 1

ui->pushButton_3（按钮 3） clicked()（点击信号）this （当前窗口实例）

pushbutten_3_Slot() 触发 LCD3 计数 + 1

```
    //按钮1和LcdNumber1建立信号槽连接
    connect(ui->pushButton_1,SIGNAL(clicked()),this,SLOT(pushbutten_1_Slot()));
    //按钮2和LcdNumber2建立信号槽连接
    connect(ui->pushButton_2,SIGNAL(clicked()),this,SLOT(pushbutten_2_Slot()));
    //按钮3和LcdNumber1.2.3建立信号槽连接
    connect(ui->pushButton_3,SIGNAL(clicked()),this,SLOT(pushbutten_1_Slot()));
    connect(ui->pushButton_3,SIGNAL(clicked()),this,SLOT(pushbutten_2_Slot()));
    connect(ui->pushButton_3,SIGNAL(clicked()),this,SLOT(pushbutten_3_Slot()));
}
```

关键特性：

“一对多”：按钮 3 的 1 个 clicked 信号，绑定 3 个槽函数，实现 “点击一次，三个 LCD 同时更新”。

“多对一”：按钮 1 和按钮 3 的信号，均可绑定 pushbutten_1_Slot()，实现 “两个按钮控制同一个 LCD”。

（3）事件触发与槽函数执行流程

以 “点击按钮 3” 为例，完整流程如下：

1. 用户点击按钮 3 → 按钮 3 发射`clicked()`信号；

2. Qt 事件循环捕获该信号，查找所有绑定的槽函数；

3. 按绑定顺序依次执行三个槽函数：

   a. `pushbutten_1_Slot()`：获取`LCD1`当前值（`ui->lcdNumber_1->value()`），加 1 后通过`display()`显示；

   b. `pushbutten_2_Slot()`：获取`LCD2`当前值，加 1 后显示；

   c. `pushbutten_3_Slot()`：获取`LCD3`当前值，加 1 后显示；

4. 槽函数执行完毕，LCD 控件自动刷新，界面更新为最新计数。

```
SIgnalSlots::SIgnalSlots(QWidget *parent)
    : QDialog(parent)
    , ui(new Ui::SIgnalSlots)
{
    ui->setupUi(this);
    //按钮1和LcdNumber1建立信号槽连接
    connect(ui->pushButton_1,SIGNAL(clicked()),this,SLOT(pushbutten_1_Slot()));
    //按钮2和LcdNumber2建立信号槽连接
    connect(ui->pushButton_2,SIGNAL(clicked()),this,SLOT(pushbutten_2_Slot()));
    //按钮3和LcdNumber1.2.3建立信号槽连接
    connect(ui->pushButton_3,SIGNAL(clicked()),this,SLOT(pushbutten_1_Slot()));
    connect(ui->pushButton_3,SIGNAL(clicked()),this,SLOT(pushbutten_2_Slot()));
    connect(ui->pushButton_3,SIGNAL(clicked()),this,SLOT(pushbutten_3_Slot()));
}

SIgnalSlots::~SIgnalSlots()
{
    delete ui;
}
void SIgnalSlots::pushbutten_1_Slot()
{
    //在lcdNumber_1中数字加1
    ui->lcdNumber_1->display(ui->lcdNumber_1->value() + 1);
}
void SIgnalSlots::pushbutten_2_Slot()
{
    //在lcdNumber_2中数字加1
    ui->lcdNumber_2->display(ui->lcdNumber_2->value() + 1);
}
void SIgnalSlots::pushbutten_3_Slot()
{
    //在lcdNumber_3中数字加1
    ui->lcdNumber_3->display(ui->lcdNumber_3->value() + 1);

}
```

（4）LCD 显示控件核心逻辑

本工程使用 QLCDNumber 控件显示计数，核心 API 说明：

value()：获取当前显示的数字（默认初始值为 0）；

display(double num)：设置显示的数字，自动格式化显示为整数（因计数逻辑为整数递增）；

特性：QLCDNumber 默认支持循环递增，数值无上限（持续加 1 不会溢出，仅显示位数范围内的数字）。

核心.cpp 文件代码注解如下

signalslots.cpp

```
#include "signalslots.h"
#include "ui_signalslots.h"  // 编译时由signalslots.ui生成，包含UI控件的访问接口

// 构造函数：初始化窗口和信号槽连接
SIgnalSlots::SIgnalSlots(QWidget *parent)
    : QDialog(parent)  // 继承QDialog（对话框类），指定父控件（默认nullptr，独立窗口）
    , ui(new Ui::SIgnalSlots)  // 创建UI对象（封装界面控件）
{
    ui->setupUi(this);  // 初始化UI：加载signalslots.ui设计的控件布局、样式、初始状态

    // 1. 按钮1与LCD1绑定：点击按钮1 → 触发pushbutten_1_Slot（LCD1加1）
    // 注：此处使用Qt旧版信号槽语法（SIGNAL/SLOT宏），需确保信号和槽函数签名完全匹配
    connect(ui->pushButton_1, SIGNAL(clicked()), this, SLOT(pushbutten_1_Slot()));

    // 2. 按钮2与LCD2绑定：点击按钮2 → 触发pushbutten_2_Slot（LCD2加1）
    connect(ui->pushButton_2, SIGNAL(clicked()), this, SLOT(pushbutten_2_Slot()));

    // 3. 按钮3与LCD1、LCD2、LCD3绑定：点击按钮3 → 触发三个槽函数（三个LCD同时加1）
    connect(ui->pushButton_3, SIGNAL(clicked()), this, SLOT(pushbutten_1_Slot()));  //
绑定LCD1
    connect(ui->pushButton_3, SIGNAL(clicked()), this, SLOT(pushbutten_2_Slot()));  //
绑定LCD2
    connect(ui->pushButton_3, SIGNAL(clicked()), this, SLOT(pushbutten_3_Slot()));  //
绑定LCD3
}

// 析构函数：释放UI资源，避免内存泄漏
SIgnalSlots::~SIgnalSlots()
{
    delete ui;  // UI对象由new创建，需手动释放（父控件不会自动回收）
```

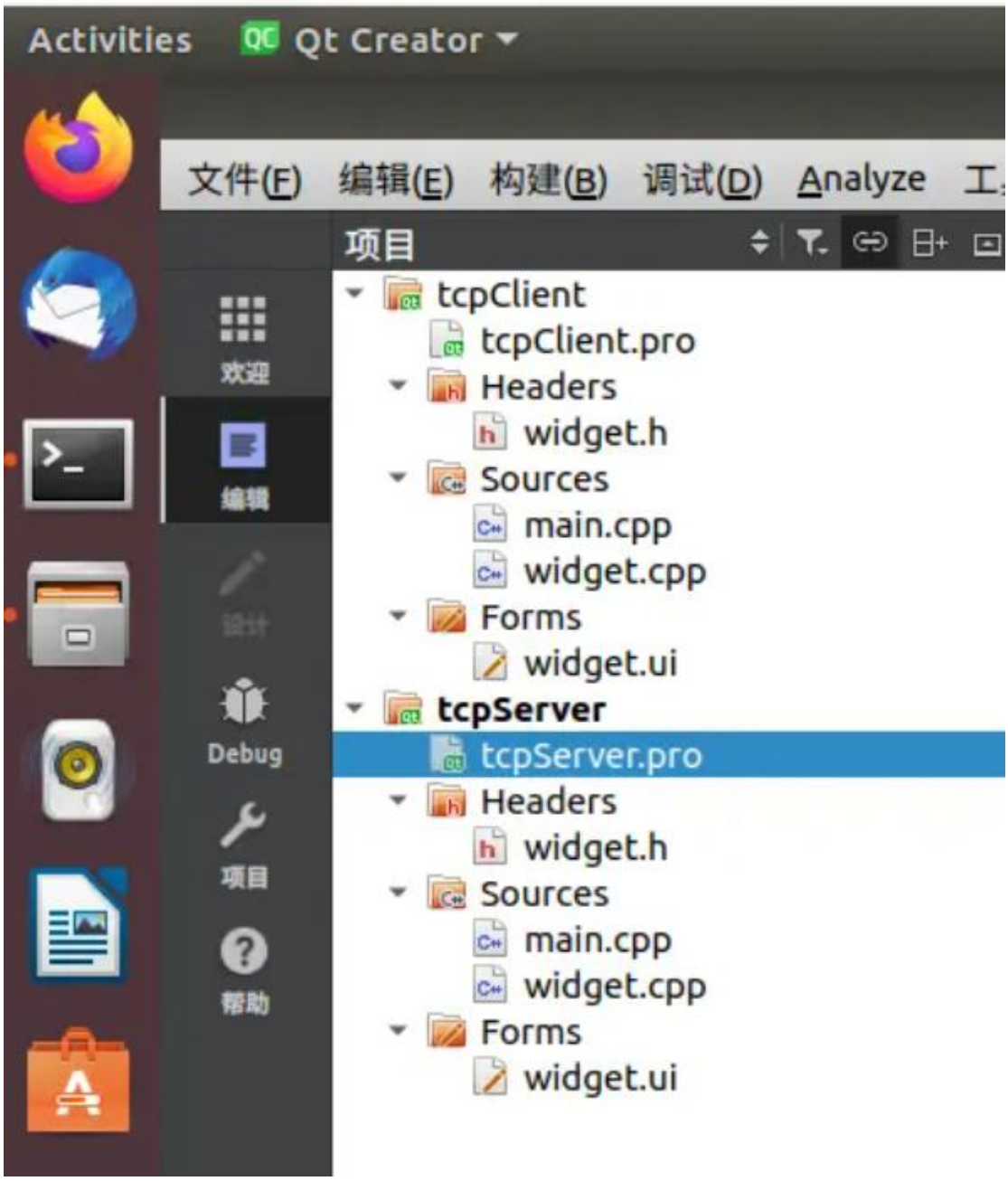
```
}

// 槽函数：按钮1或按钮3触发，LCD1计数加1
void SIgnalSlots::pushbutten_1_Slot()
{
    // 步骤：1.获取LCD1当前值 → 2.加1 → 3.显示更新后的值
    ui->lcdNumber_1->display(ui->lcdNumber_1->value() + 1);
}

// 槽函数：按钮2或按钮3触发，LCD2计数加1
void SIgnalSlots::pushbutten_2_Slot()
{
    // 同理：获取LCD2当前值，加1后显示
    ui->lcdNumber_2->display(ui->lcdNumber_2->value() + 1);
}

// 槽函数：按钮3触发，LCD3计数加1
void SIgnalSlots::pushbutten_3_Slot()
{
    // 同理：获取LCD3当前值，加1后显示
    ui->lcdNumber_3->display(ui->lcdNumber_3->value() + 1);
    }
```

c) 文件操作

d) 数据库管理

e) 网络编程（TCP）

项目结构如下



文件关系如下

tcpClient（客户端）部分：

tcpClient.pro 项目配置文件：指定依赖模块（widgets+network）、编译文件列表，是客户端工程编译入口；

headers/widget.h 头文件：声明客户端主窗口类 Widget，包含 QTcpSocket（TCP 通信套接字）、数据存储成员及槽函数声明

sources/main.cpp 程序入口文件：创建 QApplication 实例和客户端窗口实例，启动客户端事件循环；

sources/widget.cpp 实现文件：实现客户端 TCP 连接、数据接收、错误处理等核心逻辑，绑定信号与槽；

forms/widget.ui UI 设计文件：可视化设计客户端界面（主机地址输入框、端口输入框、连接按钮、接收信息标签）；

tcpServer（服务器端）部分：

tcpServer.pro 项目配置文件：指定依赖模块（widgets+network）、编译文件列表，是服务器端工程编译入口；

headers/widget.h 头文件：声明服务器端主窗口类 Widget，包含 QTcpServer（TCP 监听套接字）及槽函数声明；

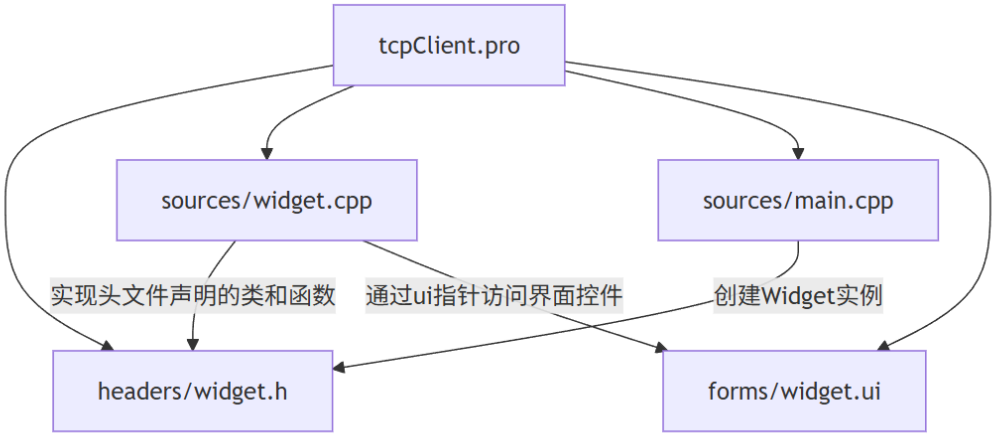
sources/main.cpp 程序入口文件：创建 QApplication 实例和服务器端窗口实例，启动服务器端事件循环；

sources/widget.cpp 实现文件：实现服务器端端口监听、客户端连接接收、数据发送等核心逻辑，绑定信号与槽；

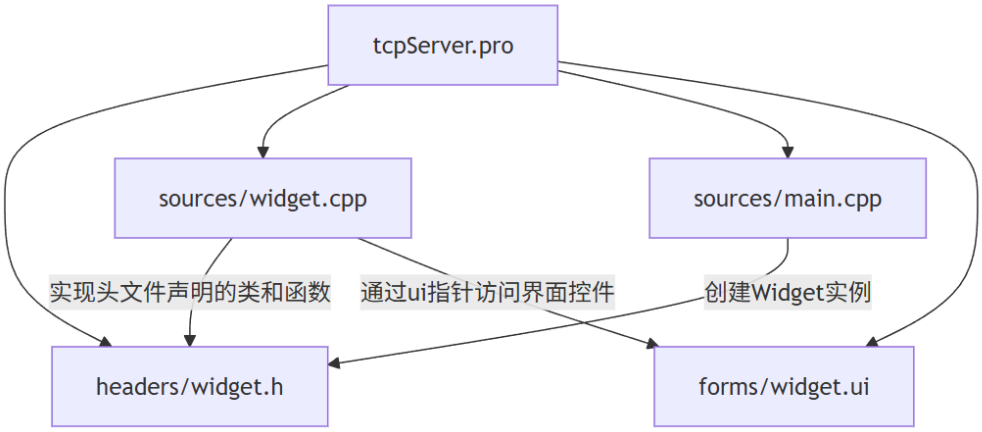
forms/widget.ui UI 设计文件：可视化设计服务器端界面（发送文本框、发送状态标签）；

文件依赖关系图

（1） 客户端（tcpClient）文件依赖



（2）服务器端（tcpServer）文件依赖



（3）客户端与服务器端网络交互关系

依赖逻辑说明：

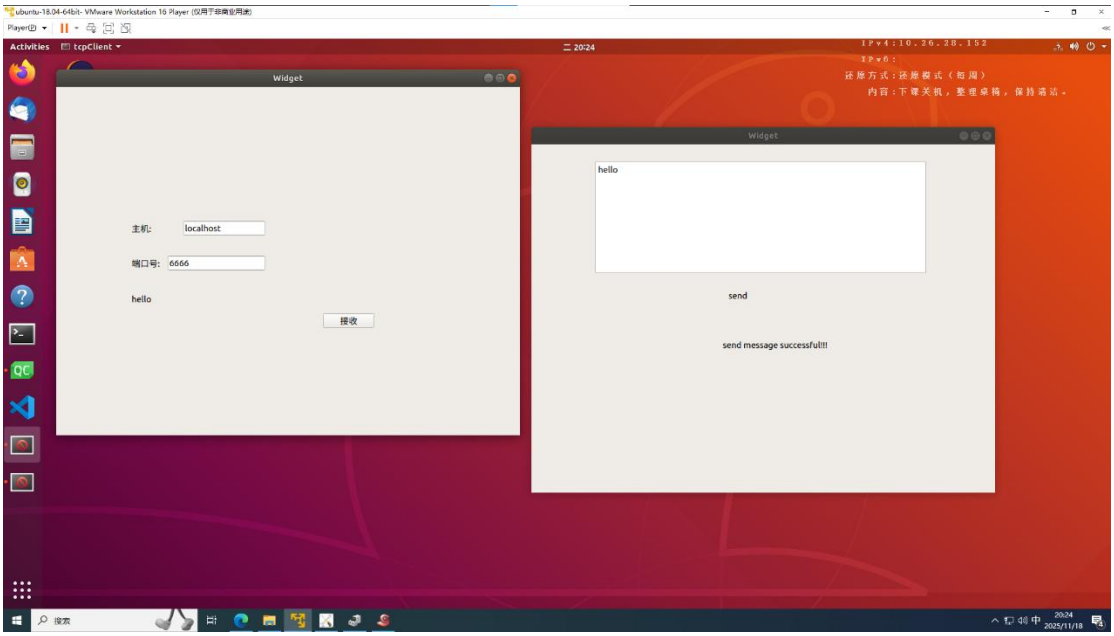
i 两个工程均依赖 Qt 的 network 模块（提供 QTcpSocket、QTcpServer 等网络类）和 widgets 模块 (提供图形界面控件)，需在.pro 文件中添加 QT += widgets network（默认配置)。

ii 客户端的 widget.cpp 依赖 QTcpSocket 实现 TCP 通信，服务器端的 widget.cpp 依赖 QTcpServer 实现端口监听和连接接收。

iii 界面控件通过 ui 指针访问，依赖 ui_widget.h（编译时由.ui 文件自动生成），实现“界面设计” 与 “网络逻辑” 分离。

工作原理分析

实验现象如下



服务器端：启动后自动监听本地主机（localhost）的 6666 端口，在文本框中输入消息（如 “hello world!”），当客户端成功连接时，自动发送该消息，发送完成后显示 “send message successful!!!”。

客户端：启动后，在 “主机” 输入框填写 localhost（同一台机器通信），“端口号” 输入框填写 6666，点击 “连接” 按钮，成功连接服务器后，接收服务器发送的消息

并显示在 “接收” 标签中（如 “hello world!”）。

核心工作流程拆解

（1）服务器端工作流程（核心：监听连接 + 发送数据）

1. 程序启动：main.cpp 创建 QApplication 和 Widget 实例，显示服务器端窗口。

2. 初始化监听：

- Widget 构造函数中创建 QTcpServer 对象（tcpServer）。

- 调用 tcpServer->listen(QHostAddress::LocalHost, 6666)：绑定本地主机地址，监听 6666 端口（端口号固定）。

- 若监听失败（如端口被占用），输出错误信息并关闭窗口；监听成功则等待客户端连接。

3. 绑定信号槽：连接 tcpServer 的 newConnection()信号到 sendMessage()槽函数（有客户端连接时触发）。

4. 接收连接与发送数据：

- 客户端发起连接请求，tcpServer 触发 newConnection()信号，调用 sendMessage()。

- 创建客户端套接字：QTcpSocket *clientConnection = tcpServer->nextPendingConnection()（获取与客户端的连接套接字）。

- 数据打包：使用 QDataStream 将消息长度（quint16 类型，2 字节）和实际消息（字符串）写入字节数组（QByteArray），避免 TCP 粘包。

- 发送数据：clientConnection->write(block)，发送打包后的字节数组。

- 断开连接：发送完成后调用 clientConnection->disconnectFromHost()，并绑定 disconnected()信号到 deleteLater()（自动释放套接字资源，避免内存泄漏）。

5. 状态提示：发送成功后，更新界面标签显示“send message successful!!!”。

（2）客户端工作流程（核心：发起连接 + 接收数据）

1. 程序启动：main.cpp 创建 QApplication 和 Widget 实例，显示客户端窗口。

2. 初始化通信：

- Widget 构造函数中创建 QTcpSocket 对象（tcpSocket）。

- 绑定信号槽：

- readyRead()信号 → readMessage()槽函数（有数据到达时触发）。

- error(QAbstractSocket::SocketError)信号 → displayError()槽函数（网络错误时触发）。

- 连接按钮的 clicked()信号 → pushButton_clicked()槽函数（用户点击连接时触发）。

3. 发起连接：

- 用户点击“连接”按钮，调用 pushButton_clicked()，进而调用 newConnect()。

- newConnect()中：

- tcpSocket->abort()：取消现有连接（避免重复连接）。

- tcpSocket->connectToHost(ui->hostLineEdit->text(), ui->portLineEdit->text().toInt())：根据用户输入的主机地址和端口，发起 TCP 连接请求。

4. 接收数据：

- 连接成功后，服务器端发送数据，tcpSocket 触发 readyRead()信号，调用 readMessage()。

- 数据解析（解决 TCP 粘包问题）：

- 初始时 blockSize=0：先判断是否接收到 2 字节（quint16 类型长度），若未接收到则返回（继续等待数据）；若接收到则读取该长度存入 blockSize。

- 再判断已接收的数据长度是否≥blockSize（实际消息长度），若未达到则返回（继续等待）；若达到则读取实际消息存入 message 变量。

- 界面更新：将 message 显示到 ui->messageLabel（接收标签）。

5. 错误处理：若连接失败或通信异常，触发 error()信号，调用 displayError()，输出错误信息（如 tcpSocket->errorString()）。

（3）TCP 数据打包与解析逻辑（关键：避免粘包）

TCP 是流式协议，数据以字节流形式传输，可能出现 “粘包”（多个消息合并）或 “拆包”（一个消息拆分）问题。本项目通过 “长度 + 数据” 的格式打包数据，确保客户端正确解析，具体格式如下：

数据段	类型	长度（字节）	作用
长度字段	quint16（无符号短整型）	2	存储后续实际消息的字节长度
数据字段	QString（字符串）	可变（由长度字段指定）	实际要传输的消息内容

服务器端打包：

创建 QByteArray（字节数组）作为数据缓冲区。

使用 QDataStream 写入数据：先写入占位符（quint16 (0)），再写入实际消息，最后将文件指针移回开头，更新长度字段为 “实际消息长度”（block.size () - sizeof (quint16)）。

客户端解析：

先读取长度字段（blockSize），再根据 blockSize 读取对应长度的实际消息，确保数据完整性。

核心代码注解如下

widget（client）.cpp

```
#include "widget.h"
#include "ui_widget.h"

// 构造函数：初始化客户端窗口、TCP套接字及信号槽绑定
Widget::Widget(QWidget* parent)
    : QWidget(parent)
    , ui(new Ui::Widget)
{
    ui->setupUi(this);  // 初始化UI界面（加载widget.ui设计的控件：主机输入框、端口输入框、连接按钮、接收标签）

    tcpSocket = new QTcpSocket(this);  // 创建TCP套接字对象（用于与服务器通信）

    // 绑定信号槽1：有数据到达时（readyRead信号），触发readMessage槽函数（接收并解析数据）
    connect(tcpSocket, SIGNAL(readyRead()), this, SLOT(readMessage()));
    // 绑定信号槽2：网络出错时（error信号），触发displayError槽函数（显示错误信息）
    connect(tcpSocket, SIGNAL(error(QAbstractSocket::SocketError)),
        this, SLOT(displayError(QAbstractSocket::SocketError)));
    // 绑定信号槽3：点击连接按钮时（clicked信号），触发pushButton_clicked槽函数（发起连接）
    connect(ui->pushButton, SIGNAL(clicked()), this, SLOT(pushButton_clicked()));
}

// 析构函数：释放UI资源
Widget::~Widget()
{
    delete ui;  // 释放UI对象，避免内存泄漏
}

// 槽函数：发起与服务器的TCP连接
void Widget::newConnect()
{
    blockSize = 0;  // 初始化数据长度变量（每次连接重置，避免旧数据干扰）
```

```
    tcpSocket->abort();  // 取消当前所有已存在的连接（避免重复连接导致的异常）

    // 发起连接：主机地址从hostLineEdit获取，端口号从portLineEdit获取（转为整数）
    tcpSocket->connectToHost(ui->hostLineEdit->text(),
        ui->portLineEdit->text().toInt());
}

// 槽函数：接收并解析服务器发送的数据
void Widget::readMessage()
{
    QDataStream in(tcpSocket);  // 创建数据流，绑定tcpSocket（从套接字读取数据）
    in.setVersion(QDataStream::Qt_4_6);  // 数据流版本与服务器一致（关键：避免解析错
误）

    // 阶段1：读取数据长度（blockSize）
    if (blockSize == 0)  // 尚未读取到数据长度（首次接收数据）
    {
        // 判断已接收的数据是否≥2字节（quint16类型的长度）：若不足则返回，继续等待数据
        if (tcpSocket->bytesAvailable() < (int)sizeof(quint16)) return;
        in >> blockSize;  // 读取2字节的长度信息，存入blockSize
    }

    // 阶段2：读取实际消息
    // 判断已接收的数据是否≥blockSize（实际消息长度）：若不足则返回，继续等待数据
    if (tcpSocket->bytesAvailable() < blockSize) return;
    in >> message;  // 读取blockSize长度的实际消息，存入message变量

    ui->messageLabel->setText(message);  // 在界面标签中显示接收的消息
}

// 槽函数：显示网络错误信息
void Widget::displayError(QAbstractSocket::SocketError)
{
    // 输出错误信息：errorString()返回人类可读的错误描述（如Connection refused）
    qDebug() << "error" << tcpSocket->errorString();
}

// 槽函数：连接按钮点击事件触发，调用newConnect发起连接
void Widget::pushButton_clicked()
{
    newConnect();  // 转发到newConnect函数，发起TCP连接
    }
```

widget（server）.cpp

```cpp
 #include "widget.h"
#include "ui_widget.h"

// 构造函数：初始化服务器端窗口、TCP服务器对象及信号槽绑定
Widget::Widget(QWidget* parent)
    : QWidget(parent)
    , ui(new Ui::Widget)
{
    ui->setupUi(this);  // 初始化UI界面（加载widget.ui设计的控件：文本框、状态标签）

    tcpServer = new QTcpServer(this);  // 创建TCP服务器对象，父控件设为this（自动释放）

    // 监听本地主机（localhost）的6666端口：QHostAddress::LocalHost表示127.0.0.1
    if (!tcpServer->listen(QHostAddress::LocalHost, 6666))
    {
        qDebug() << "error" << tcpServer->errorString();  // 监听失败，输出错误信息（如端口被占用）
        close();  // 关闭窗口，终止程序
    }

    // 绑定信号槽：当有新客户端连接时（newConnection信号），触发sendMessage槽函数（发送数据）
    connect(tcpServer, SIGNAL(newConnection()), this, SLOT(sendMessage()));
}

// 析构函数：释放UI资源
Widget::~Widget()
{
    delete ui;  // 释放UI对象，避免内存泄漏
}

// 槽函数：有新客户端连接时触发，发送数据给客户端
void Widget::sendMessage()
{
    QByteArray block;  // 字节数组：暂存要发送的打包数据（长度+消息）
    // 创建数据流：绑定block，以“只写”模式操作（QIODevice::WriteOnly）
    QDataStream out(&block, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_6);  // 设置数据流版本，必须与客户端一致（避免数据解析错误）

    out << (quint16)0;  // 写入占位符：预留2字节存储消息长度（后续更新）
    qDebug() << ui->textEdit_send->toPlainText();  // 调试输出：打印要发送的消息（文本框输入内容）
    out << ui->textEdit_send->toPlainText();  // 写入实际消息（从文本框获取）
```

```
    out.device()->seek(0);  // 将数据流指针移回开头（覆盖之前的占位符）
    // 计算实际消息长度：block总长度 - 占位符长度（sizeof(quint16)=2字节）
    out << (quint16)(block.size() - sizeof(quint16));

    // 获取与客户端的连接套接字：nextPendingConnection()返回等待连接的客户端套接字
    QTcpSocket* clientConnection = tcpServer->nextPendingConnection();
    // 绑定信号槽：客户端断开连接时，自动释放套接字资源（deleteLater()延迟删除，避免野指针）
    connect(clientConnection, SIGNAL(disconnected()), clientConnection, SLOT(deleteLater()));

    clientConnection->write(block);  // 发送打包后的字节数组（长度+消息）
    clientConnection->disconnectFromHost();  // 发送完成后，主动断开与客户端的连接

    ui->statusLabel->setText("send message successful!!!");  // 界面显示发送成功提示
    qDebug() << "send message successful!!!";  // 调试输出发送成功信息
    }
```

f) 网络编程（UDP）

g) 程序引用

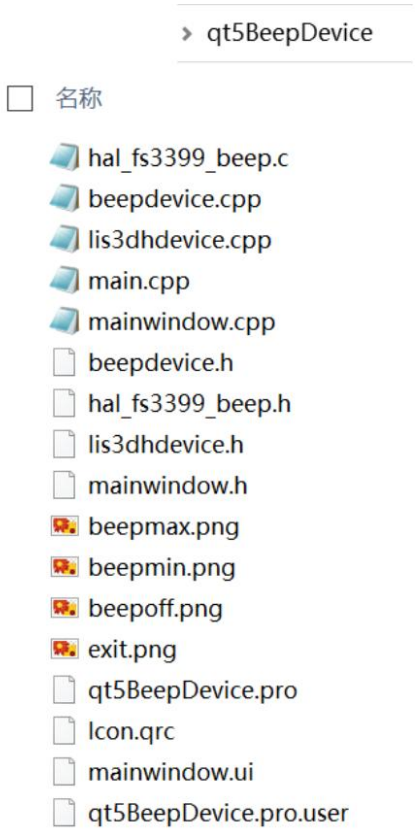
任选 3 个工程文件（其中：网络编程 e 和 f 二择一），进行代码剖析

2. 针对第二次 Qt 实验的 8 个实验样例：

a) LED 灯

b) 蜂鸣器

项目文件如下



文件类别	文件路径 / 名称	文件类型	核心作用	所属层级
项目配置	qt5BeepDevice.pro	配置文件	指定依赖模块（`widgets`）、编译文件列表、资源文件路径，是工程编译入口	-
资源文件	Icon.qrc、beepmax.png 等	资源文件	存储界面图标（蜂鸣器开关状态图标、退出图标），通过 Qt 资源系统引用	应用层

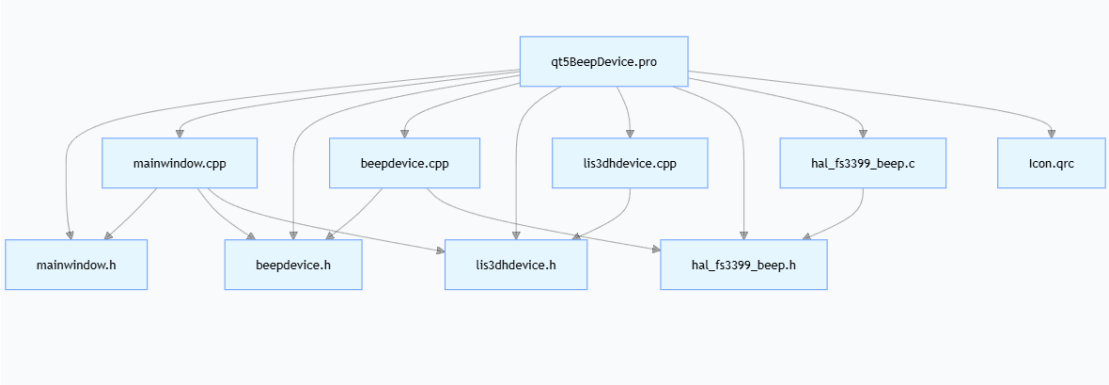
文件类别	文件路径 / 名称	文件类型	核心作用	所属层级
应用层头文件	mainwindow.h	头文件	声明主窗口类 MainWindow、自定义界面类 CustomWidget、顶部标签类 TopLabel，引用设备层类	应用层
应用层源文件	main.cpp	程序入口	创建 QApplication、主窗口实例，设置全屏显示、隐藏光标	应用层
应用层源文件	mainwindow.cpp	实现文件	搭建 UI 界面、创建设备实例、管理线程、绑定信号槽，实现蜂鸣器控制和界面翻转逻辑	应用层
设备层头文件	beepdevice.h	头文件	声明蜂鸣器设备类 beepDevice，封装 HAL 层接口为 Qt 可用的成员函数	设备层

文件类别	文件路径 / 名称	文件类型	核心作用	所属层级
设备层源文件	beepdevice.cpp	实现文件	实现蜂鸣器开关、初始化、销毁接口，调用 HAL 层函数	设备层
设备层头文件	lis3dhdevice.h	头文件	声明加速度传感器类 Lis3dhDevice，包含设备查找、数据读取、线程控制接口	设备层
设备层源文件	lis3dhdevice.cpp	实现文件	实现传感器设备节点查找、三轴数据读取、线程循环检测，发射界面翻转信号	设备层
HAL 层头	hal_fs3399_beep.h	头文件	声明蜂鸣器硬件操作函数、设备路径宏定义（/dev/buzzer_ctl）、控制命令宏（BEEP_ON/OFF）	HAL 层

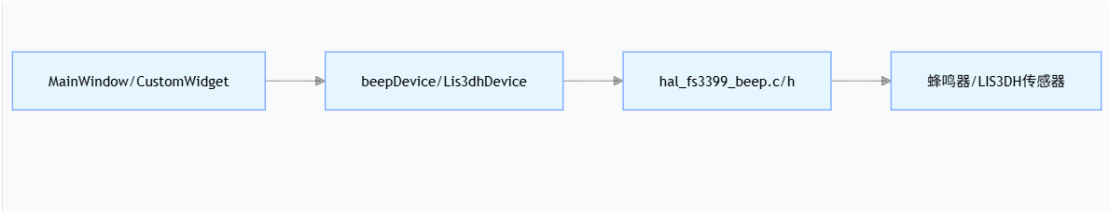
文件类别	文件路径 / 名称	文件类型	核心作用	所属层级
文件				
HAL 层源文件	hal_fs3399_beep.c	实现文件	直接操作硬件：打开 / 关闭蜂鸣器设备文件、通过 ioctl 控制蜂鸣器开关	HAL 层
UI 文件	mainwindow.ui	设计文件	本工程未实际使用（UI 通过代码动态搭建），仅为项目结构默认文件	应用层

文件依赖关系图

（1）文件间依赖关系



（2）分层依赖关系（核心设计）



工作原理分析

实验现象

运行程序后，嵌入式设备全屏显示界面，核心现象如下：

界面组成：顶部显示 “蜂鸣器控制” 标签 + 退出按钮，中间显示大型蜂鸣器控制按钮（初始显示 “关闭” 图标 beepoff.png）。

蜂鸣器控制：点击中间按钮，切换蜂鸣器状态 ——① 关闭→开启：按钮图标变为 beepmax.png，蜂鸣器发声；② 开启→关闭：图标恢复 beepoff.png，蜂鸣器停止。

界面翻转联动：当加速度传感器 (LIS3DH) 的 Y 轴数据 > 700 时，界面自动旋转 180 度；当 Y 轴数据≤700 时，界面恢复原位。

退出功能：点击顶部退出按钮，程序关闭，蜂鸣器停止工作。

核心工作流程拆解

（1）程序初始化流程（从启动到就绪）

1. 程序启动：main.cpp 创建 QApplication 实例，设置全屏显示、隐藏光标，创建 MainWindow 主窗口。

2. 主窗口初始化（mainwindow.cpp）：

- 搭建 UI：创建 CustomWidget（自定义界面），包含顶部标签、蜂鸣器按钮、退出按钮，设置布局为全屏。

- 设备实例化：创建蜂鸣器设备`beepDevice`、加速度传感器设备`Lis3dhDevice`。

- 线程管理：创建 QThread 线程，将`Lis3dhDevice`移至线程（避免传感器数据读取阻塞 UI），绑定线程`started`信号到传感器`run`函数（线程启动时开始读取数据）。

- 信号槽绑定：

  - 传感器`change180()`信号→界面旋转 180 度；

  - 传感器`change0()`信号→界面恢复原位；

  - 传感器`stopthread()`信号→线程安全退出；

  - 蜂鸣器按钮`clicked()`信号→`on_beep_clicked()`槽函数（控制蜂鸣器开关）；

  - 退出按钮`clicked()`信号→`on_exit_clicked()`槽函数（关闭程序）。

3. 硬件初始化：

- 蜂鸣器：`beepDevice`构造函数调用`beep_init()`（HAL 层），打开设备文件`/dev/buzzer_ctl`。

- 传感器：线程启动后，`Lis3dhDevice::run()`开始循环查找传感器设备节点、读取数据。

（2）蜂鸣器控制流程（应用层→硬件）

核心逻辑：通过三层调用实现硬件控制，每层职责明确，无直接耦合。

1. 应用层触发：用户点击蜂鸣器按钮，触发`on_beep_clicked()`槽函数。

2. 状态判断：

- 若当前蜂鸣器关闭(`beepState=false`)：调用`beep->beepOn()`(设备层接口)，切换图标为`beepmax.png`，设置`beepState=true`。

- 若当前蜂鸣器开启(`beepState=true`)：调用`beep->beepOff()`(设备层接口)，切换图标为`beepoff.png`，设置`beepState=false`。

3. 设备层转发：`beepDevice::beepOn()`/`beepOff()`调用 HAL 层的`beep_on()`/`beep_off()`函数。

4. HAL 层硬件操作：

- `beep_on()`：通过`ioctl(fd, BEEP_ON)`向设备文件发送“开启”命令(`BEEP_ON=1`)，蜂鸣器按固定频率发声。

- `beep_off()`：通过`ioctl(fd, BEEP_OFF)`发送“关闭”命令(`BEEP_OFF=0`)，蜂鸣器停止发声。

5. 资源释放：程序退出时，`beepDevice`析构函数调用`beep_close()`(HAL 层)，关闭设备文件描述符`fd`，避免资源泄漏。

(3)传感器数据处理与界面翻转流程

核心逻辑：传感器数据读取在独立线程中执行，通过信号槽通知应用层更新界面，避免 UI 阻塞。

1. 线程启动：MainWindow 初始化时启动 QThread，触发`Lis3dhDevice::run()`。

2. 传感器设备查找：

- 调用`openAccelDeviceXYZ()`，通过`scandir`遍历`/sys/bus/iio/devices/`目录，筛选出包含“iio:”的传感器节点(LIS3DH 属于 IIO 设备)。

- 拼接 X/Y/Z 轴数据文件路径（如`/sys/bus/iio/devices/iio:device0/in_accel_x_raw`），打开设备文件，获取文件描述符`adcxfd_x/y/z`。

3. 数据读取：

- 调用`readAccelValues()`，通过`read()`函数从文件描述符读取三轴原始数据，存入`xValue/yValue/zValue`。

- 关闭文件描述符（避免资源占用）。

4. 条件判断与信号发射：

- 若 Y 轴数据 > 700：发射`change180()`信号。

- 否则：发射`change0()`信号。

5. 界面响应：

- 应用层接收`change180()`信号：调用`view->rotate(180)`，界面旋转 180 度，标记`is180=true`。

- 应用层接收`change0()`信号：再次调用`view->rotate(180)`（旋转 180 度恢复原位），标记`is180=false`。

6. 线程安全退出：若设备查找或数据读取失败，发射`stopthread()`信号，触发线程`quit()`，避免死循环。

（4）退出流程

1. 用户点击顶部退出按钮，触发`on_exit_clicked()`槽函数。

2. 关闭主窗口：`this->close()`。

3. 退出应用：`QApplication::quit()`，触发所有对象析构。

4. 资源释放：`beepDevice`析构函数调用`beep_close()`，关闭蜂鸣器设备文件；线程

自动退出，释放传感器资源。

核心.cpp 文件代码注解见打包的文件

HAL 层核心文件：hal_fs3399_beep.c（直接操作硬件）

```
#include "hal_fs3399_beep.h"

int fd;  // 全局文件描述符：用于关联蜂鸣器设备文件

// 初始化蜂鸣器：打开设备文件
int beep_init()
{
    // 打开蜂鸣器设备节点（/dev/buzzer_ctl是驱动层提供的设备文件）
    fd = open(BEEP_PATH, O_RDWR);  // O_RDWR：可读可写模式
    if (fd < 0) {
        perror("open beep failed");  // 打开失败，输出错误信息（如设备不存在、权限不
足）
    }
    return 0;
}

// 开启蜂鸣器：通过ioctl发送控制命令
int beep_on()
{
    // ioctl：设备控制函数，fd=设备文件描述符，BEEP_ON=控制命令（宏定义为1）
    return ioctl(fd, BEEP_ON);
}

// 关闭蜂鸣器：发送关闭命令
int beep_off()
{
    return ioctl(fd, BEEP_OFF);  // BEEP_OFF=0，设置蜂鸣器频率和占空比为0
}

// 释放资源：关闭设备文件
void beep_close()
{
    close(fd);  // 关闭文件描述符，释放系统资源
    }
```

设备层核心文件：beepdevice.cpp（封装 HAL 层接口）

```
#include "beepdevice.h"
```

```
// 构造函数：初始化蜂鸣器（调用HAL层初始化函数）
beepDevice::beepDevice() {
    beep_init();  // 初始化：打开蜂鸣器设备文件
}

// 开启蜂鸣器：对外提供Qt类接口
int beepDevice::beepOn() {
    int ret;
    ret = beep_on();  // 调用HAL层beep_on()，发送开启命令
    if (ret < 0) {
        perror("open BEEP failed\n");  // 开启失败，输出错误信息
    }
    return ret;  // 返回执行结果（0=成功，-1=失败）
}

// 关闭蜂鸣器：对外提供Qt类接口
int beepDevice::beepOff() {
    int ret;
    ret = beep_off();  // 调用HAL层beep_off()，发送关闭命令
    if (ret < 0) {
        perror("open led failed\n");  // 此处错误提示笔误，应为“close BEEP failed”
    }
    return ret;
}

// 析构函数：释放蜂鸣器资源
beepDevice::~beepDevice() {
    beep_close();  // 调用HAL层beep_close()，关闭设备文件
    }
```

设备层核心文件：lis3dhdevice.cpp（传感器数据处理）

```
#include "lis3dhdevice.h"

// 构造函数：空实现（初始化在run()中执行）
Lis3dhDevice::Lis3dhDevice()
{
}

// 析构函数：空实现（资源释放由线程退出时完成）
Lis3dhDevice::~Lis3dhDevice()
{
}

// 筛选函数：scandir遍历目录时，只保留IIO设备节点（d_name以"iio:"开头的符号链接）
```

```
int Lis3dhDevice::filter(const dirent* entry)
{
    return entry->d_type == DT_LNK && strncmp(entry->d_name, "iio:", 4) == 0;
}

// 打开传感器X/Y/Z轴设备文件，获取文件描述符
int Lis3dhDevice::openAccelDeviceXYZ(int& adcxfd_x, int& adcxfd_y, int& adcxfd_z,
QString function) {
    struct dirent** namelist;
    // 遍历/sys/bus/iio/devices/目录，用filter筛选节点，alphasort排序
    int n = scandir("/sys/bus/iio/devices/", &namelist, filter, alphasort);
    if (n < 0) {
        perror("scandir");  // 遍历失败（如目录不存在）
        return -1;
    }
    bool x_opened = false, y_opened = false, z_opened = false;
    for (int i = 0; i < n; i++) {
        char device_path[1024];
        // 根据function参数，拼接对应轴的设备文件路径
        if (function.compare("X") == 0 && !x_opened) {
            snprintf(device_path, sizeof(device_path),
"/sys/bus/iio/devices/%s/in_accel_x_raw", namelist[i]->d_name);
            x_opened = true;
        }
        else if (function.compare("Y") == 0 && !y_opened) {
            snprintf(device_path, sizeof(device_path),
"/sys/bus/iio/devices/%s/in_accel_y_raw", namelist[i]->d_name);
            y_opened = true;
        }
        else if (function.compare("Z") == 0 && !z_opened) {
            snprintf(device_path, sizeof(device_path),
"/sys/bus/iio/devices/%s/in_accel_z_raw", namelist[i]->d_name);
            z_opened = true;
        }
        else {
            continue;  // 无需打开当前轴，跳过
        }
        // 检查设备文件是否存在
        struct stat device_file_stat;
        if (stat(device_path, &device_file_stat) == 0) {
            // 打开设备文件（可读可写模式）
            int adcfd = open(device_path, O_RDWR);
            if (adcfd < 0) {
                perror("open device");  // 打开失败
```

```
                return -1;
            }
            else {
                // 将文件描述符赋值给对应轴的变量
                if (function.compare("X") == 0) adcxfd_x = adcfd;
                else if (function.compare("Y") == 0) adcxfd_y = adcfd;
                else if (function.compare("Z") == 0) adcxfd_z = adcfd;
            }
        }
        else {
            return -1;  // 文件不存在，返回失败
        }
        free(namelist[i]);  // 释放目录项内存
    }
    free(namelist);  // 释放目录列表内存
    return 0;
}

// 从设备文件读取原始数据，存入value
int Lis3dhDevice::readDeviceValue(int fd, QString& value)
{
    char buffer[64];
    // 读取数据：最多读取63字节（留1字节存结束符）
    ssize_t bytesRead = read(fd, buffer, sizeof(buffer) - 1);
    if (bytesRead > 0) {
        buffer[bytesRead] = '\0';  // 手动添加字符串结束符，避免乱码
        value = QString(buffer);   // 转换为Qt字符串，存入value
        return 0;  // 读取成功
    }
    else {
        return -1;  // 读取失败（如设备断开）
    }
}

// 读取X/Y/Z轴数据，存入成员变量
int Lis3dhDevice::readAccelValues(int adcxfd_x, int adcxfd_y, int adcxfd_z)
{
    QString value_x, value_y, value_z;
    // 读取X轴数据
    if (readDeviceValue(adcxfd_x, value_x) == 0) xValue = value_x;
    else { perror("Failed to read X-axis value"); return -1; }
    // 读取Y轴数据（核心判断轴）
    if (readDeviceValue(adcxfd_y, value_y) == 0) yValue = value_y;
    else { perror("Failed to read Y-axis value"); return -1; }
```

```
    // 读取Z轴数据
    if (readDeviceValue(adcxfd_z, value_z) == 0) zValue = value_z;
    else { perror("Failed to read Z-axis value"); return -1; }
    // 关闭文件描述符，避免资源泄漏
    close(adcxfd_x);
    close(adcxfd_y);
    close(adcxfd_z);
    return 0;
}

// 线程执行函数：循环读取传感器数据，发射翻转信号
void Lis3dhDevice::run()
{
    runningState = true;  // 标记线程运行状态
    while (runningState) {  // 循环读取，直到runningState为false
        int adcxfd_x, adcxfd_y, adcxfd_z;
        // 分别打开X/Y/Z轴设备文件
        int xret = openAccelDeviceXYZ(adcxfd_x, adcxfd_y, adcxfd_z, "X");
        int yret = openAccelDeviceXYZ(adcxfd_x, adcxfd_y, adcxfd_z, "Y");
        int zret = openAccelDeviceXYZ(adcxfd_x, adcxfd_y, adcxfd_z, "Z");
        if (xret < 0 || yret < 0 || zret < 0) {  // 任一轴打开失败
            emit stopthread();  // 发射线程停止信号
            return;
        }
        else {
            readAccelValues(adcxfd_x, adcxfd_y, adcxfd_z);  // 读取三轴数据
            // 根据Y轴数据发射翻转信号
            if (yValue.toInt() > 700) {
                emit change180();  // Y轴数据＞700，发射180度翻转信号
            }
            else {
                emit change0();    // 否则发射恢复0度信号
            }
        }
    }
}

// 切换线程运行状态（用于安全退出）
void Lis3dhDevice::changeRunningState(bool state)
{
    runningState = state;
    }
```

应用层核心文件：mainwindow.cpp（界面与逻辑控制）

```cpp
#include "mainwindow.h"
#include "ui_mainwindow.h"

// 主窗口构造函数：搭建界面、初始化设备与线程、绑定信号槽
MainWindow::MainWindow(QWidget* parent)
    : QMainWindow(parent)
{
    // 创建图形场景和视图（用于界面旋转）
    QGraphicsScene* scene = new QGraphicsScene(this);
    view = new QGraphicsView(scene, this);

    // 创建自定义界面（CustomWidget）
    customWidget = new CustomWidget();
    QScreen* screen = QGuiApplication::primaryScreen();  // 获取屏幕信息
    QRect screenGeometry = screen->geometry();
    int screenWidth = screenGeometry.width();
    int screenHeight = screenGeometry.height();
    customWidget->resize(screenWidth, screenHeight);  // 设置界面大小为屏幕尺寸
    customWidget->showFullScreen();  // 全屏显示
    customWidget->topLabel->raise();  // 将顶部标签置于顶层（避免被遮挡）

    // 禁用滚动条，将自定义界面添加到场景
    view->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    view->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    proxyWidget = scene->addWidget(customWidget);
    proxyWidget->setTransformOriginPoint(proxyWidget->boundingRect().center());  // 旋转原点设为界面中心
    proxyWidget->setZValue(100);  // 设置层级，确保界面可见

    // 初始化传感器设备和线程
    lis3dh = new Lis3dhDevice();
    thread3dh = new QThread();
    lis3dh->moveToThread(thread3dh);  // 传感器对象移至独立线程（避免阻塞UI）

    // 绑定线程启动信号：线程启动时执行传感器的run()函数
    connect(thread3dh, &QThread::started, lis3dh, &Lis3dhDevice::run);

    // 绑定线程停止信号：传感器发射stopthread时，安全退出线程
    connect(lis3dh, &Lis3dhDevice::stopthread, [&]() {
        if (thread3dh->isRunning()) {
            lis3dh->changeRunningState(false);  // 标记线程停止
            thread3dh->quit();  // 退出线程事件循环
        }
        });
```

```
    // 绑定界面翻转信号：传感器发射change180时，界面旋转180度
    connect(lis3dh, &Lis3dhDevice::change180, [&]() {
        if (!customWidget->is180) {  // 未旋转时才执行
            qDebug() << "turn 180";
            this->view->rotate(180);  // 旋转180度
            update();  // 刷新界面
            customWidget->is180 = true;
            customWidget->is0 = true;
        }
        });

    // 绑定界面恢复信号：传感器发射change0时，界面恢复原位
    connect(lis3dh, &Lis3dhDevice::change0, [&]() {
        if (customWidget->is0) {  // 已旋转时才执行
            qDebug() << "turn 0";
            this->view->rotate(180);  // 再次旋转180度，恢复0度
            update();
            customWidget->is180 = false;
            customWidget->is0 = false;
        }
        });

    // 绑定按钮信号槽：蜂鸣器按钮→控制槽函数；退出按钮→退出槽函数
    connect(customWidget->m_beepButton, SIGNAL(clicked()), this,
SLOT(on_beep_clicked()));
    connect(customWidget->m_beepButton1, SIGNAL(clicked()), this,
SLOT(on_exit_clicked()));

    // 设置视图为中心控件
    setCentralWidget(view);
}

// 蜂鸣器控制槽函数：响应按钮点击，切换蜂鸣器状态
void MainWindow::on_beep_clicked()
{
    if (customWidget->beepState == false)  // 当前关闭→开启
    {
        customWidget->beep->beepOn();  // 调用设备层开启接口
        customWidget->beepState = true;  // 更新状态标记
        customWidget->m_beepButton->setIcon(QIcon(":/icon/beepmax.png"));  // 切换图标
    }
    else  // 当前开启→关闭
    {
```

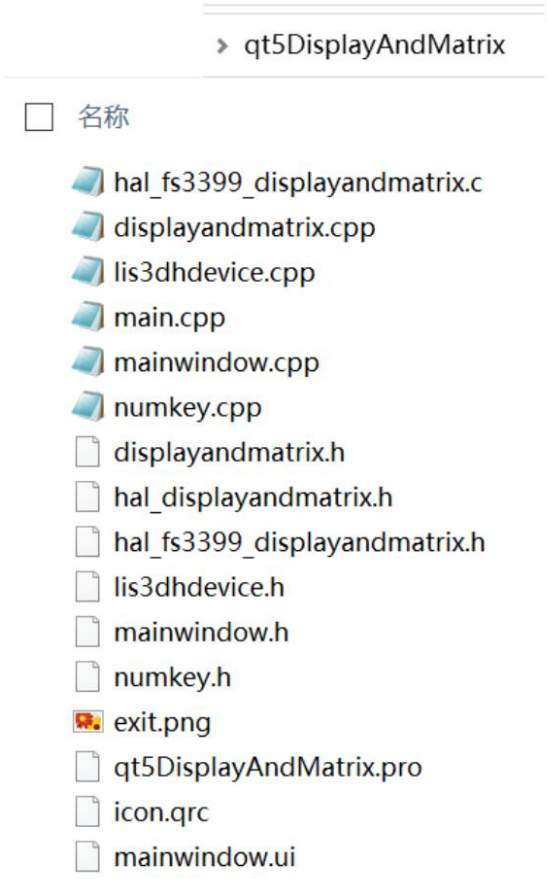
```
        customWidget->beep->beepOff();  // 调用设备层关闭接口
        customWidget->beepState = false;  // 更新状态标记
        customWidget->m_beepButton->setIcon(QIcon(":/icon/beepoff.png"));  // 切换图标
    }
}

// 退出槽函数：关闭程序
void MainWindow::on_exit_clicked()
{
    this->close();  // 关闭主窗口
    QApplication::quit();  // 退出应用程序
}

// 析构函数：释放资源
MainWindow::~MainWindow()
{
    delete customWidget->m_beepButton;  // 释放蜂鸣器按钮
    delete customWidget->m_beepButton1; // 释放退出按钮
    delete customWidget->beep;  // 释放蜂鸣器设备
    }
```

c) 按键

d) 电位检测（ADC）

e) 小键盘/数码管

项目文件如下



文件类别	文件路径 / 名称	文件类型	核心作用	所属层级
项目配置	qt5DisplayAndMatrix.pro	配置文件	指定依赖模块（`widgets`）、编译文件列表、资源路径，是工程编译入口	-
资源文件	Icon.qrc、exit.png	资源文件	存储界面图标（退出按钮图标），通过 Qt 资源系统引用	应用层

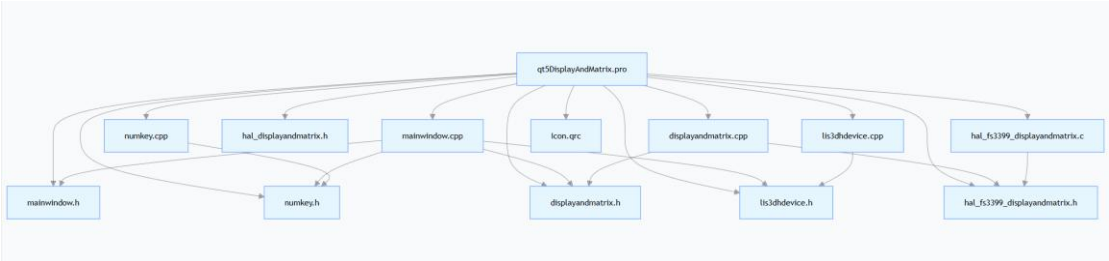
文件类别	文件路径 / 名称	文件类型	核心作用	所属层级
应用层头文件	mainwindow.h	头文件	声明主窗口类 MainWindow、自定义界面类 CustomWidget、值获取类 ValueGetter 等，引用设备层和传感器类	应用层
应用层源文件	main.cpp	程序入口	创建 QApplication、主窗口实例，设置全屏显示、隐藏光标	应用层
应用层源文件	mainwindow.cpp	实现文件	搭建 UI 界面、管理线程（传感器线程、值获取线程）、绑定信号槽，实现按键处理和界面翻转逻辑	应用层
应用层头文件	numkey.h	头文件	声明小键盘类 numKey，封装 0-9 数字按钮的创建和获取接口	应用层

文件类别	文件路径 / 名称	文件类型	核心作用	所属层级
应用层源文件	numkey.cpp	实现文件	实现数字按钮的初始化（样式、字体、布局）和 `getButton` 接口	应用层
设备层头文件	displayandmatrix.h	头文件	声明设备类 `DisplayandMatrix`，封装 HAL 层小键盘 / 数码管操作接口为 Qt 类	设备层
设备层源文件	displayandmatrix.cpp	实现文件	实现设备初始化、按键值获取、缓冲区更新、设备关闭等接口，调用 HAL 层函数	设备层
HAL 层头	hal_displayandmatrix.h	头文件	中转引用 `hal_fs3399_displayandmatrix.h`，简化上层引用	HAL 层

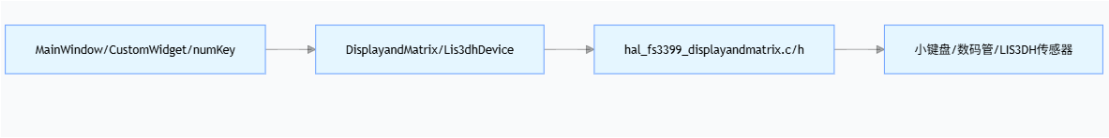
文件类别	文件路径 / 名称	文件类型	核心作用	所属层级
文件				
HAL层头文件	hal_fs3399_displayandmatrix.h	头文件	声明小键盘 / 数码管硬件操作函数、设备路径（/dev/zlg72xx）、IOCTL 命令宏（SET_VAL/GET_KEY）	HAL层
HAL层源文件	hal_fs3399_displayandmatrix.c	实现文件	直接操作硬件：打开设备文件、通过 ioctl 获取按键值、更新数码管缓冲区、关闭设备	HAL层
传感器相关	lis3dhdevice.h/cpp	头文件 / 源文件	加速度传感器设备类，实现三轴数据读取、线程循环、界面翻转信号发射	设备层

文件类别	文件路径 / 名称	文件类型	核心作用	所属层级
UI 文件	mainwindow.ui	设计文件	未实际使用（UI 通过代码动态搭建），仅为项目结构默认文件	应用层

文件依赖关系图



分层逻辑



工作原理分析

实验现象

此时，触摸液晶屏上的键，或者按实验箱上的小键盘，液晶屏、实验箱的数码管上都会显示相应的值

界面组成：顶部显示 “数码管 & 矩阵按键” 标签 + 退出按钮；中间是红色显示的 LCD 数码管（8 位，初始为空）；下方是小键盘（0-9 数字按钮 + A/B/C/D/*/# 功能按钮，采用网格布局）。

输入与显示：点击数字按钮（0-9）或功能按钮（A/B/C/D/*/#），LCD 数码管实时追加显示对应字符（最多显示 8 位，超出部分向左滚动）。

小键盘硬件按键 (若连接实体小键盘) 输入时，LCD 和硬件数码管同步显示输入字符，终端打印按键信息（如 “put is '1'”）。

界面翻转联动：当加速度传感器 Y 轴数据 > 700 时，界面自动旋转 180 度；Y 轴数据≤700 时，界面恢复原位。

退出功能：点击顶部退出按钮，程序关闭，释放设备资源，强制终止相关进程。

核心工作流程拆解

（1）程序初始化流程（从启动到就绪）

1. 程序启动：main.cpp 创建 QApplication 实例，设置全屏显示、隐藏光标，创建 MainWindow 主窗口。

2. 主窗口初始化（mainwindow.cpp）：

- 搭建 UI：创建 CustomWidget（自定义界面），包含顶部标签、LCD 数码管、小键盘（数字+功能按钮），设置深色背景和按钮样式。

- 设备实例化：创建小键盘/数码管设备`DisplayandMatrix`、加速度传感器设备`Lis3dhDevice`。

- 线程管理：

  - 传感器线程：创建 QThread，将`Lis3dhDevice`移至线程，绑定`started`信号到`run`函数（启动传感器数据读取）。

- 值获取线程：创建`ValueGetter`类（用于循环获取按键值），移至独立线程，绑定`started`信号到`run`函数，绑定`valueReceived`信号到`getValues`函数（更新 LCD 显示）。

- 信号槽绑定：

- 数字按钮`clicked`信号→`numberButtonClicked`槽函数（处理数字输入）。

- 功能按钮`clicked`信号→`letterButtonClicked`槽函数（处理 A/B/C/D/*/#输入）。

- 传感器`change180`/`change0`信号→界面旋转/恢复。

- 退出按钮`clicked`信号→`buttonexit`槽函数（关闭程序）。

3. 硬件初始化：

- 小键盘/数码管：`DisplayandMatrix`构造函数调用`dis_mat_init()`（HAL 层），打开设备文件`/dev/zlg72xx`。

- 传感器：线程启动后，`Lis3dhDevice::run()`开始循环查找传感器节点、读取三轴数据。

（2）小键盘输入与显示流程（软件按钮 + 硬件按键双支持）

① 软件按钮输入流程（界面按钮点击）

1. 用户点击界面数字/功能按钮，触发`numberButtonClicked`/`letterButtonClicked`槽函数。

2. 字符转换：获取按钮文本（如“1” “A” “*”），转换为 C 字符类型。

3. 缓冲区更新：调用`dm->trunNewBuf(*c_str)`（设备层），将字符写入 HAL 层缓冲区`buf`（缓冲区共 8 位，新字符追加到末尾，旧字符向左滚动）。

4. LCD 显示：`accumulatedStr`字符串追加字符，调用`Lcd->display()`实时更新 LCD

显示。

5. 数码管同步：设备层调用 HAL 层`ioctl(fd, SET_VAL, buf)`，将缓冲区数据发送到硬件数码管，实现同步显示。

② 硬件按键输入流程（实体小键盘操作）

1. 用户按下实体小键盘按键，硬件触发按键事件。

2. 按键值获取：`ValueGetter`线程循环调用`dm->getValue()`（设备层），设备层调用`Run()`（HAL 层）。

3. HAL 层处理：

- 调用`ioctl(fd, GET_KEY, &key)`获取硬件按键原始值（如 28 对应数字 1）。
- 通过`switch`语句将原始值映射为对应字符（如 28→'1'），终端打印按键信息。

4. 缓冲区更新：将映射后的字符写入缓冲区`buf`（向左滚动，保留最新 8 位），调用`ioctl(fd, SET_VAL, buf)`更新数码管。

5. LCD 同步显示：`ValueGetter`发射`valueReceived`信号，触发`getValues`槽函数，`accumulatedStr`追加字符，LCD 更新显示。

（3）传感器数据处理与界面翻转流程

与蜂鸣器工程一致，核心逻辑：

1. 传感器线程启动后，循环查找 IIO 设备节点，打开 X/Y/Z 轴数据文件，读取原始数据。

2. 条件判断：Y 轴数据 > 700→发射`change180`信号；否则→发射`change0`信号。

3. 界面响应：接收信号后，调用`view->rotate(180)`实现界面旋转/恢复，更新状态标记（`is180`/`is0`）。

4. 线程安全退出：设备查找或数据读取失败时，发射`stopthread`信号，触发线程

`quit()`。

（4）退出流程

1. 用户点击退出按钮，触发`buttonexit`槽函数。

2. 线程停止：设置`getter->isrunning = false`，退出值获取线程并等待结束；退出传感器线程。

3. 设备关闭：调用`dm->close()`（设备层），进而调用 HAL 层`close_dis()`。

4. 资源释放：清空缓冲区`buf`，关闭设备文件描述符`fd`，强制终止`qt5DisplayAndMatrix`进程。

5. 程序退出：关闭主窗口，调用`QApplication::quit()`终止应用。

核心.cpp 文件代码注解如下

HAL 层核心文件：hal_fs3399_displayandmatrix.c（直接操作硬件）

```
#include "hal_fs3399_displayandmatrix.h"

int fd = 0;  // 全局文件描述符：关联小键盘/数码管设备文件
char buf[10] = { 0 };  // 8位显示缓冲区（预留2位避免越界）

// 初始化设备：打开设备文件
int dis_mat_init()
{
    // 打开设备节点（/dev/zlg72xx是小键盘/数码管驱动提供的设备文件）
    fd = open("/dev/zlg72xx", O_RDWR);  // O_RDWR：可读可写模式
    if (fd < 0) {
        perror("open");  // 打开失败（如设备不存在、权限不足），输出错误信息
    }
    return 0;
}

// 运行设备：获取硬件按键值，映射为字符，更新缓冲区
int Run()
{
    int key = 0;  // 存储硬件按键原始值
    char value;  // 存储映射后的字符
```

```
    ioctl(fd, SET_VAL, buf);  // 发送缓冲区数据到数码管（更新显示）
    ioctl(fd, GET_KEY, &key);  // 获取硬件按键原始值（通过IOCTL命令）

    // 按键值映射：将原始key值转换为对应字符（如28→'1'）
    switch (key)
    {
    case 28: printf("put is '1'\n"); value = '1'; break;
    case 27: printf("put is '2'\n"); value = '2'; break;
    case 26: printf("put is '3'\n"); value = '3'; break;
    case 25: printf("put is 'A'\n"); value = 'A'; break;
    case 20: printf("put is '4'\n"); value = '4'; break;
    case 19: printf("put is '5'\n"); value = '5'; break;
    case 18: printf("put is '6'\n"); value = '6'; break;
    case 17: printf("put is 'B'\n"); value = 'B'; break;
    case 12: printf("put is '7'\n"); value = '7'; break;
    case 11: printf("put is '8'\n"); value = '8'; break;
    case 10: printf("put is '9'\n"); value = '9'; break;
    case 9: printf("put is 'C'\n"); value = 'C'; break;
    case 4: printf("put is '*'\n"); value = '*'; break;
    case 3: printf("put is '0'\n"); value = '0'; break;
    case 2: printf("put is '#'\n"); value = '#'; break;
    case 1: printf("put is 'D'\n"); value = 'D'; break;
    default: value = 0x00; break;  // 无按键时返回空字符
    }
    puts("==========================");

    // 缓冲区滚动：字符向左移动1位，新字符写入末尾（保留最新8位）
    for (int i = 0; i < 7; i++)
    {
        buf[i] = buf[i + 1];
    }
    buf[7] = value;

    ioctl(fd, SET_VAL, buf);  // 发送更新后的缓冲区到数码管，同步显示
    return value;  // 返回映射后的字符，供上层使用
}

// 更新缓冲区：软件输入（界面按钮）时调用，直接写入新字符
void turn_new_buf(char value) {
    // 与Run()中缓冲区逻辑一致，实现滚动效果
    for (int i = 0; i < 7; i++)
    {
        buf[i] = buf[i + 1];
    }
```

```
    buf[7] = value;
    ioctl(fd, SET_VAL, buf);  // 同步更新数码管
}

// 关闭设备：释放资源
void close_dis()
{
    memset(buf, 0, sizeof(buf));  // 清空缓冲区
    close(fd);  // 关闭设备文件描述符
    system("killall -9 qt5DisplayAndMatrix");  // 强制终止进程，避免资源泄漏
    }
```

设备层核心文件：displayandmatrix.cpp（封装 HAL 层接口）

```
#include "displayandmatrix.h"

// 构造函数：初始化设备（调用HAL层初始化函数）
DisplayandMatrix::DisplayandMatrix()
{
    dis_mat_init();  // 调用HAL层dis_mat_init()，打开设备文件
}

// 获取按键值：供上层调用，获取硬件按键映射后的字符
char DisplayandMatrix::getValue()
{
    char value = Run();  // 调用HAL层Run()，获取按键字符
    return value;
}

// 更新缓冲区：软件输入（界面按钮）时，更新硬件数码管缓冲区
void DisplayandMatrix::trunNewBuf(char value)
{
    turn_new_buf(value);  // 调用HAL层turn_new_buf()，写入新字符并滚动
}

// 关闭设备：释放硬件资源
void DisplayandMatrix::close()
{
    close_dis();  // 调用HAL层close_dis()，清空缓冲区、关闭文件
    }
```

应用层核心文件：numkey.cpp（小键盘数字按钮封装）

```
#include "numkey.h"

// 构造函数：初始化0-9数字按钮
```

```
numKey::numKey(QWidget* parent) : QWidget(parent)
{
    // 反向循环（9→0）创建按钮，便于后续布局
    for (int i = 9; i >= 0; i--) {
        // 创建按钮，显示文本为数字i
        buttons[i] = new QPushButton(QString::number(i), this);
        // 设置按钮样式：深色背景、无边框、圆角、hover/pressed状态变色
        buttons[i]->setStyleSheet("QPushButton {"
            "background-color: #444444;"
            "border: none;"
            "color: white;"
            "padding: 10px;"
            "border-radius: 5px;"
            "outline: none;"
            "}"
            "QPushButton:hover {"
            "background-color: #555555;"
            "outline: none;"
            "}"
            "QPushButton:pressed {"
            "background-color: #666666;"
            "outline: none;"
            "}");
        buttons[i]->setFixedHeight(80);  // 固定按钮高度
        QFont buttonsfont("Noto Sans CJK SC Regular", 20);  // 设置字体大小
        buttons[i]->setFont(buttonsfont);
    }
}

// 获取指定索引的数字按钮（0-9）
QPushButton* numKey::getButton(int index) {
    if (index >= 0 && index < 10) {  // 检查索引有效性（0-9）
        return buttons[index];  // 返回对应按钮对象
    }
    return nullptr;  // 索引无效时返回空指针
    }
```

应用层核心文件：mainwindow.cpp（界面与逻辑控制）

```
#include "mainwindow.h"

// 主窗口构造函数：搭建界面、初始化设备与线程、绑定信号槽
MainWindow::MainWindow(QWidget* parent)
    : QMainWindow(parent)
{
```

```
    // 创建图形场景和视图（用于界面旋转）
    QGraphicsScene* scene = new QGraphicsScene(this);
    view = new QGraphicsView(scene, this);

    // 创建自定义界面（CustomWidget）
    customWidget = new CustomWidget();
    QScreen* screen = QGuiApplication::primaryScreen();
    QRect screenGeometry = screen->geometry();
    int screenWidth = screenGeometry.width();
    int screenHeight = screenGeometry.height();
    customWidget->resize(screenWidth, screenHeight);
    customWidget->showFullScreen();  // 全屏显示

    // 禁用滚动条，将自定义界面添加到场景
    view->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    view->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    proxyWidget = scene->addWidget(customWidget);
    proxyWidget->setTransformOriginPoint(proxyWidget->boundingRect().center());  // 旋
转原点为界面中心
    proxyWidget->setZValue(100);  // 确保界面层级置顶

    // 初始化传感器设备和线程（界面翻转用）
    lis3dh = new Lis3dhDevice();
    thread3dh = new QThread();
    lis3dh->moveToThread(thread3dh);
    thread3dh->start();  // 启动传感器线程
    // 线程启动时，执行传感器run()函数（读取三轴数据）
    connect(thread3dh, &QThread::started, lis3dh, &Lis3dhDevice::run);
    // 传感器读取失败时，安全退出线程
    connect(lis3dh, &Lis3dhDevice::stopthread, [&]() {
        if (thread3dh->isRunning()) {
            lis3dh->changeRunningState(false);
            thread3dh->quit();
        }
        });
    // 接收change180信号，界面旋转180度
    connect(lis3dh, &Lis3dhDevice::change180, [&]() {
        if (!customWidget->is180) {
            qDebug() << "turn 180";
            this->view->rotate(180);
            update();  // 刷新界面
            customWidget->is180 = true;
            customWidget->is0 = true;
        }
```

```
        });
    // 接收change0信号，界面恢复原位
    connect(lis3dh, &Lis3dhDevice::change0, [&]() {
        if (customWidget->is0) {
            qDebug() << "turn 0";
            this->view->rotate(180);  // 再次旋转180度，抵消之前的旋转
            update();
            customWidget->is180 = false;
            customWidget->is0 = false;
        }
        });

    // 绑定数字按钮信号槽（0-9）
    for (int i = 0; i < 10; i++) {
        QPushButton* button = customWidget->numKeys->getButton(i);
        connect(button, &QPushButton::clicked, this, [this, i]() {
            numberButtonClicked(i);  // 点击时触发数字处理槽函数
            });
    }

    // 绑定功能按钮信号槽（A/B/C/D/*/#）
    connect(customWidget->AButton, &QPushButton::clicked, this,
&MainWindow::letterButtonClicked);
    connect(customWidget->BButton, &QPushButton::clicked, this,
&MainWindow::letterButtonClicked);
    connect(customWidget->CButton, &QPushButton::clicked, this,
&MainWindow::letterButtonClicked);
    connect(customWidget->DButton, &QPushButton::clicked, this,
&MainWindow::letterButtonClicked);
    connect(customWidget->EButton, &QPushButton::clicked, this,
&MainWindow::letterButtonClicked);
    connect(customWidget->FButton, &QPushButton::clicked, this,
&MainWindow::letterButtonClicked);

    // 绑定退出按钮信号槽
    connect(customWidget->m_exitButton1, &QPushButton::clicked, this,
&MainWindow::buttonexit);

    // 设置视图为中心控件
    setCentralWidget(view);
}

// 析构函数：释放资源
MainWindow::~MainWindow()
```

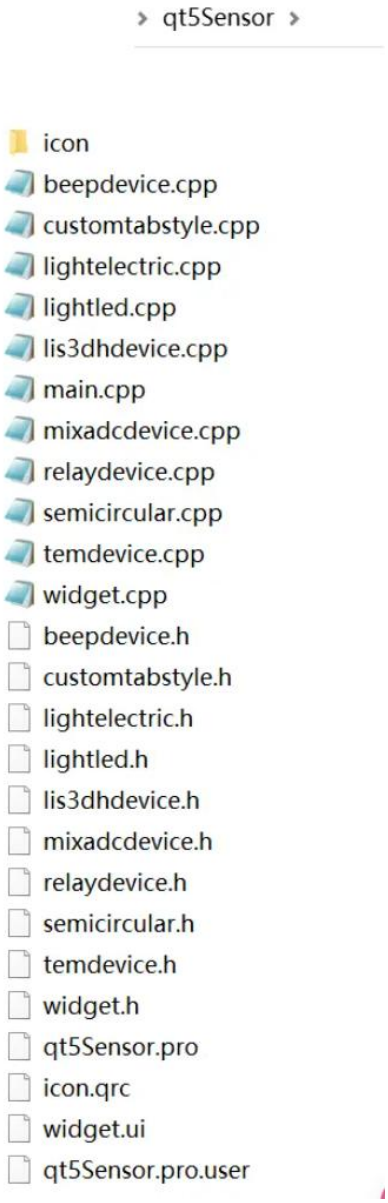
```
{
    stopGettingValues();  // 停止值获取线程
    delete customWidget->Lcd;  // 释放LCD控件
    delete customWidget->dm;  // 释放小键盘/数码管设备
}

// 停止值获取线程（安全退出）
void MainWindow::stopGettingValues() {
    if (customWidget->thread->isRunning()) {
        customWidget->getter->isrunning = false;  // 设置运行标记为false
        customWidget->thread->quit();  // 退出线程事件循环
        customWidget->thread->wait();  // 等待线程结束
    }
}

// 功能按钮点击槽函数（A/B/C/D/*/#）
void MainWindow::letterButtonClicked() {
    // 获取发送信号的按钮对象
    QPushButton* button = qobject_cast<QPushButton*>(sender());
    char* c_str;
    if (button) {
        QString numberStr = button->text();  // 获取按钮显示文本（如"A" "*"）
        QByteArray ba = numberStr.toLatin1();  // 转换为Latin1编码（兼容ASCII字符）
        c_str = ba.data();  // 转换为C字符指针

        customWidget->accumulatedStr += c_str;  // 追加到累计字符串
        customWidget->dm->trunNewBuf(*c_str);  // 更新硬件数码管缓冲区
        customWidget->Lcd->display(customWidget->accumulatedStr);  // 更新LCD显示
    }
}

// 数字按钮点击槽函数（0-9）
void MainWindow::numberButtonClicked(int num) {
    QPushButton* button = customWidget->numKeys->getButton(num);
    char* c_str;
    if (button) {
        QString numberStr = button->text();  // 获取按钮文本（如"1" "0"）
        QByteArray ba = numberStr.toLatin1();
        c_str = ba.data();

        customWidget->accumulatedStr += c_str;  // 追加字符串
        customWidget->dm->trunNewBuf(*c_str);  // 更新硬件数码管
        customWidget->Lcd->display(customWidget->accumulatedStr);  // 更新LCD
    }
```

```
}

// 退出按钮槽函数：关闭程序
void MainWindow::buttonexit()
{
    customWidget->getter->isrunning = false;  // 停止值获取线程
    if (customWidget->thread->isRunning())
    {
        qDebug() << "exit";
        customWidget->dm->close();  // 关闭小键盘/数码管设备
        customWidget->thread->quit();  // 退出线程
    }
    this->close();  // 关闭主窗口
    QApplication::quit();  // 退出应用
    }
```

f) 电机综合（3 个设备）

g) 传感器综合（8 个设备）

项目文件如下



文件功能情况如下

文件类别	文件路径 / 名称	文件类型	核心作用	所属层级
项目配置	qt5Sensor.pro	配置文件	指定依赖模块（widgets、network）、编译文件列表、	-

文件类别	文件路径 / 名称	文件类型	核心作用	所属层级
			资源路径，工程编译入口	
资源文件	icon.qrc、各类图标.png	资源文件	存储按钮、设备状态图标（如蜂鸣器开关、继电器状态图标），通过 Qt 资源系统引用	应用层
UI 组件类	lightled.h/cpp	头文件 / 源文件	自定义 LED 指示灯组件，支持颜色、位置、大小设置，用于状态指示	应用层（UI 组件）
UI 组件类	semicircular.h/cpp	头文件 / 源	自定义半圆仪表盘组件，支持温度等数值的渐变显示、刻度与指针绘制	应用层（UI 组

文件类别	文件路径 / 名称	文件类型	核心作用	所属层级
		文件		件）
UI 组件类	customtabstyle.h/cpp	头文件/源文件	自定义 QTabWidget 样式，实现垂直标签页、选中状态高亮、图标 + 文字布局	应用层（UI 组件）
传感器设备类	mixadcdevice.h/cpp	头文件/源文件	ADC 接口传感器类，支持酒精（A）、燃气（S）、火焰（F）、光强（L）数据读取	设备层（传感器）
传感器设	temdevice.h/cpp	头文件/源	I2C 接口温度传感器类（LM75），实	设备层（传

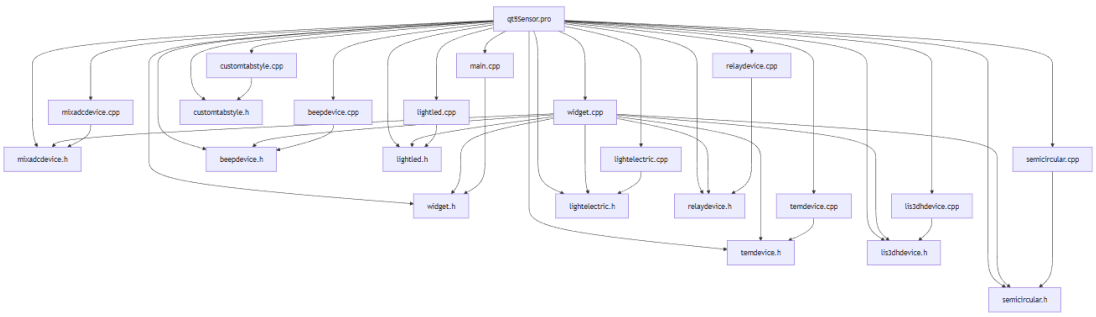
文件类别	文件路径 / 名称	文件类型	核心作用	所属层级
备类		文件	现温度数据采集与发送	感器）
传感器设备类	lightelectric.h/cpp	头文件/源文件	光电开关设备类，读取遮挡状态并发送信号	设备层（传感器）
控制设备类	beepdevice.h/cpp	头文件/源文件	蜂鸣器控制类，通过设备文件 /dev/buzzer 控制开关	设备层（控制设备）
控制设	relaydevice.h/cpp	头文件/	继电器控制类，通过设备文件	设备层（

文件类别	文件路径 / 名称	文件类型	核心作用	所属层级
备类		源文件	/dev/relay 控制开关	控制设备）
主程序文件	main.cpp	程序入口	创建 QApplication、主窗口实例，设置全屏显示、隐藏光标	应用层（主程序）
主程序文件	widget.h/cpp	头文件 / 源文件	主窗口类，负责页面管理（QTabWidget）、线程池调度、信号槽绑定、数据处理	应用层（主程序）
辅助任	ReadAdcDataTask/ReadTemDataTask 等	内部类	继承 QRunnable，封装传感器数据	应用层

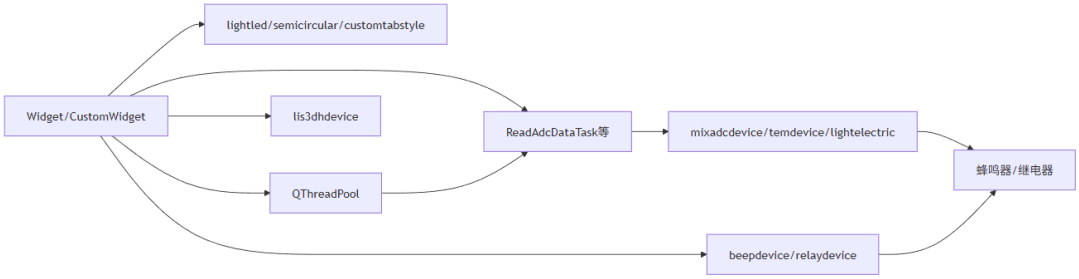
文件类别	文件路径 / 名称	文件类型	核心作用	所属层级
务类			读取任务，供线程池执行	（任务封装）
UI 设计文件	widget.ui	设计文件	未实际使用（UI 通过代码动态搭建），仅为项目结构默认文件	应用层

文件依赖关系图

（1）文件间依赖关系

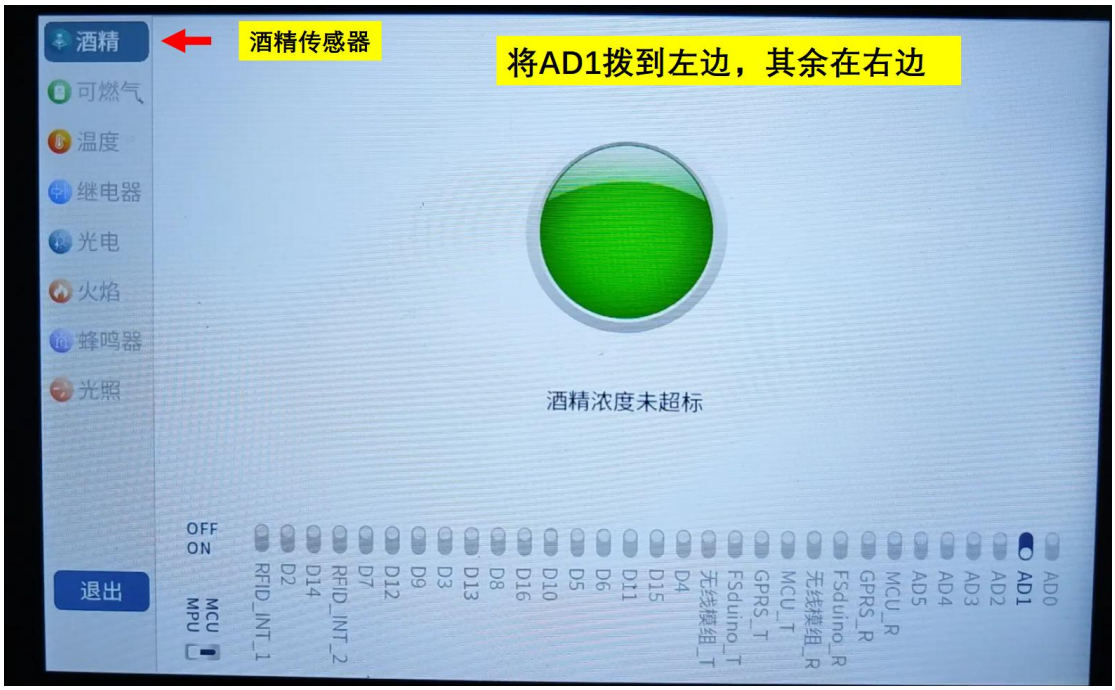


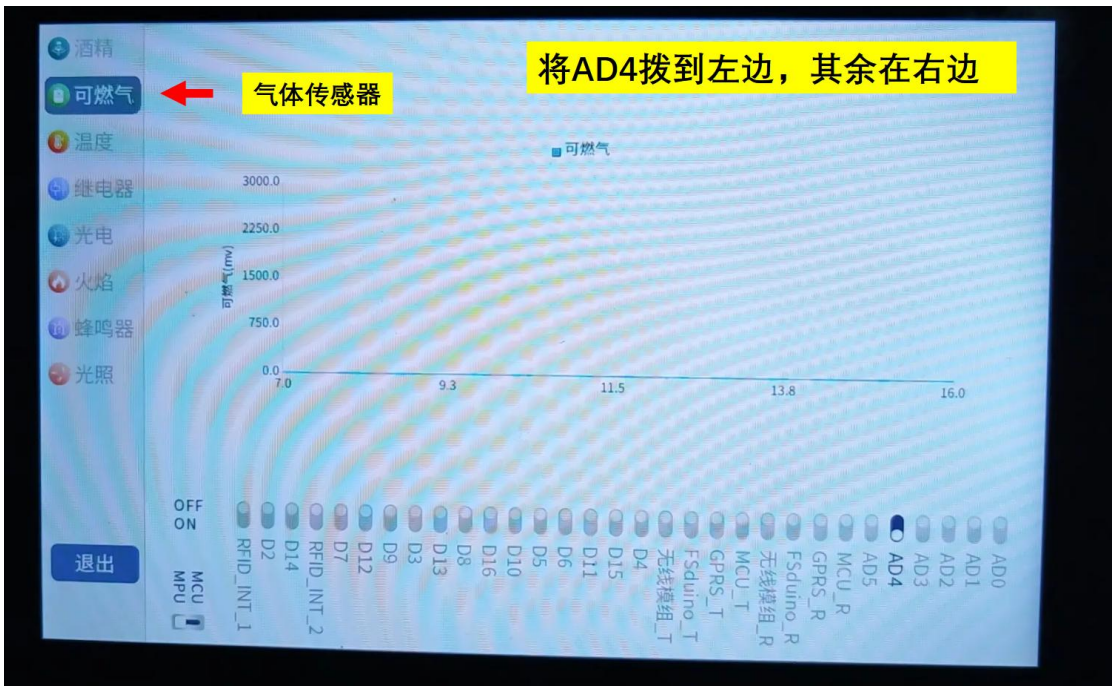
（2）功能模块依赖关系

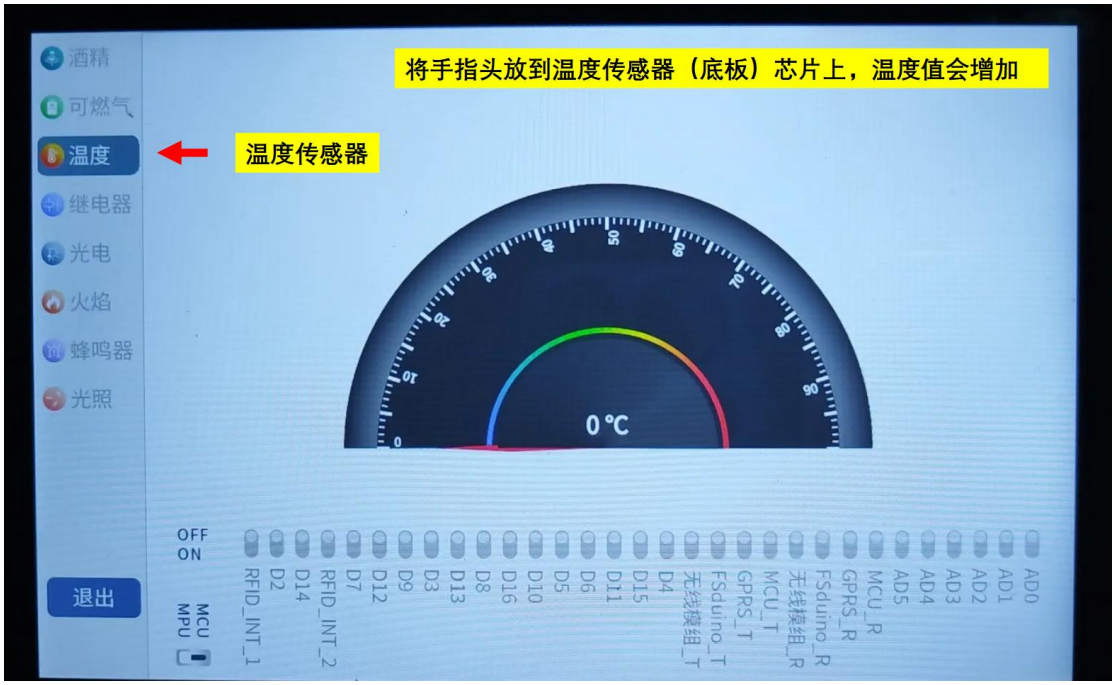

lightled/semicircular/customtabstyle
Widget/CustomWidget
lis3dhdevice
ReadAdcDataTask等
mixadcdevice/temdevice/lightelectric
蜂鸣器/继电器
QThreadPool
beepdevice/relaydevice

工作原理分析

实验现象


酒精
可燃气
温度
继电器
光电
火焰
蜂鸣器
光照
退出
酒精传感器
将AD1拨到左边，其余在右边
酒精浓度未超标
OFF
ON
MCU
MPU
RFID_INT_1
D2
D14
RFID_INT_2
D7
D12
D9
D3
D13
D8
D16
D10
D5
D6
D11
D15
D4
无线模组_T
FSduino_T
GPRS_T
MCU_T
无线模组_R
FSduino_R
GPRS_R
MCU_R
AD5
AD4
AD3
AD2
AD1
AD0





- 界面组成：采用垂直标签页（QTabWidget），包含 8 个功能页面（酒精检测、燃气检测、温度检测、光电开关、火焰检测、光强检测、继电器控制、蜂鸣器控制），顶部有退出按钮。
- 传感器数据显示：
    - 温度检测页：半圆仪表盘实时显示温度（单位℃），指针随温度变化，支持

0-100℃范围显示。

- 燃气检测页：折线图实时绘制燃气浓度数据，最多保留 10 个数据点，超出后滚动显示。
- 酒精检测页：LED 指示灯通过颜色变化指示浓度（绿色 = 未超标，红色 = 超标），搭配文字提示。
- 火焰检测页：动态图标显示（检测到火焰时播放动图，否则显示静态图标）。
- 光强检测页：LCD 数字显示器实时显示光强数值。
- 光电开关页：图标和文字指示是否检测到遮挡（遮挡 = 显示 “检测遮挡” 图标，否则 = “未检测遮挡”）。

- 设备控制：
  - 继电器控制页：按钮切换继电器状态（打开 / 关闭），图标同步更新。
  - 蜂鸣器控制页：按钮切换蜂鸣器状态（开启 = 红色图标，关闭 = 灰色图标），发声同步。
- 界面翻转：加速度传感器（LIS3DH）Y 轴数据 > 700 时，界面旋转 180 度；≤700 时恢复原位。
- 线程安全：页面切换时自动停止之前页面的传感器线程，避免资源冲突。

核心工作流程拆解

（1）初始化流程（启动到就绪）

1. 程序启动：main.cpp 创建 QApplication 实例，设置全屏显示、隐藏光标，创建 Widget 主窗口。

2. 主窗口初始化（widget.cpp）：

- 场景与UI创建：创建QGraphicsScene和QGraphicsView，添加CustomWidget

（自定义多页面组件），设置界面旋转原点。

- 设备实例化：创建 ADC 传感器类（mixadcdevice）、温度传感器类（temdevice）、光电开关类（lightelectric）、蜂鸣器类（beepdevice）、继电器类（relaydevice）、加速度传感器类（lis3dhdevice）。

- 线程初始化：创建 QThreadPool（线程池），用于执行传感器数据读取任务；创建加速度传感器线程，移至独立线程并启动。

- UI 组件初始化：创建半圆仪表盘（semicircular）、LED 指示灯（lightled）、折线图（QChart）、LCD 显示器等，绑定到对应页面。

- 信号槽绑定：

  - 页面切换信号（currentChanged）→ handleIndexChange（切换传感器线程）；

  - 传感器数据信号（sendData）→ 数据处理槽函数（如 updateCircular、updateSeries）；

  - 设备控制按钮信号（clicked）→ 控制槽函数（如 relayButtonClicked、beepBtnClicked）；

  - 加速度传感器信号（change180/change0）→ 界面旋转/恢复；

  - 退出按钮信号（clicked）→ exitBtnClicked（关闭程序）。

3. 硬件初始化：设备类构造函数尝试打开硬件设备文件（如蜂鸣器`/dev/buzzer`、I2C`/dev/i2c-4`），失败则输出错误信息。

（2）页面切换流程（核心：线程安全调度）

1. 用户点击标签页切换页面，触发 CustomWidget 的 currentChanged 信号，调用 handleIndexChange 槽函数。

2. 停止所有设备线程：

- 调用传感器类的 changeThreadState(false)，设置运行标志为 false，终止数据读取循环。

- 关闭传感器设备文件描述符（如 ADC、I2C 文件），释放硬件资源。

3. 线程池等待任务结束：调用 threadPool->waitForDone()，确保之前的任务完全退出。

4. 根据页面索引启动对应任务：

- 页面 0（酒精）/1（燃气）/5（火焰）/7（光强）：启动 ReadAdcDataTask 任务，传入传感器类型（A/S/F/L），线程池执行任务，读取 ADC 数据。

- 页面 2 (温度)：启动 ReadTemDataTask 任务，线程池执行，读取 I2C 温度数据。

- 页面 4（光电）：启动 ReadLightStateTask 任务，线程池执行，读取光电开关状态。

- 页面 3（继电器）/6（蜂鸣器）：无需启动线程，仅响应按钮控制。

（3）传感器数据读取与 UI 更新流程

以 ADC 类传感器（酒精 / 燃气 / 火焰 / 光强）为例：

1. 线程池启动 ReadAdcDataTask 任务，调用 mixadcdevice->readData(function)。

2. 传感器数据读取：

- openAdcDevice：遍历`/sys/bus/iio/devices/`目录，筛选 IIO 设备，拼接对应通道的设备文件路径（如酒精=in_voltage4_raw）。

- 打开设备文件，调用 read()读取 ADC 原始数据，转换为实际数值（×3.8 缩放）。

3. 数据发送：通过 emit sendData(senddata)发送数据，循环读取（间隔 500ms），直到 isAdcRunning 为 false。

4. UI 更新：

- 酒精（页面 0）：recvData 槽函数调用 updateAlcoholLedState，根据数据是否 > 140 切换 LED 颜色（红/绿）和文字提示。

- 燃气（页面 1）：updateSeries 槽函数更新折线图，数据点 > 10 时滚动显示。

- 火焰（页面 5）：updateFrameState 槽函数，数据 > 600 时播放火焰动图，否则显示静态图标。

- 光强（页面 7）：updateLightLCD 槽函数，更新 LCD 显示器数值。

以温度传感器（I2C 接口）为例：

1. 线程池启动 ReadTemDataTask 任务，调用 temdevice->readData()。

2. 温度数据读取：

- openTemDevice：打开 I2C 设备文件`/dev/i2c-4`，设置超时时间和重试次数。

- 构造 I2C 读写消息（地址 0x4f，寄存器 0x0），通过 ioctl(I2C_RDWR)读取 2 字节数据。

- 数据转换：将原始数据转换为温度值（÷2），循环读取（间隔 100ms）。

3. 数据发送：emit sendData((float)temp_val / 2)发送温度值。

4. UI 更新：updateCircular 槽函数调用 semicircular->changeValue，更新半圆仪表盘的指针位置和实时温度显示。

（4）设备控制流程（继电器 / 蜂鸣器）

① 继电器控制：

1. 用户点击“打开/关闭”按钮，触发 relayButtonClicked 槽函数。

2. 状态判断：

- 未开启（relayButtonState=false）：调用 relay->changeRelayState(1)，通过 ioctl 发送 RELAY_ON 命令，更新图标为“relayon.png”，按钮文字改为“关闭”。

- 已开启（relayButtonState=true）：调用 relay->changeRelayState(0)，发送 RELAY_OFF 命令，恢复图标和按钮文字。

3. 更新状态标记：relayButtonState 取反。

② 蜂鸣器控制：

1. 用户点击蜂鸣器图标按钮，触发 beepBtnClicked 槽函数。

2. 状态判断：

- 未开启（isBeepOn=false）：调用 beep->changeBeepState(1)，通过 ioctl 发送 BEEP_ON 命令，切换图标为“beepmax.png”。

- 已开启 (isBeepOn=true)：调用 beep->changeBeepState(0)，发送 BEEP_OFF 命令，恢复图标为“beepoff.png”。

3. 更新状态标记：isBeepOn 取反。

（5）界面翻转流程

与之前工程一致，核心逻辑：

1. 加速度传感器线程启动，lis3dh->run()循环读取 X/Y/Z 轴数据。

2. 条件判断：Y 轴数据＞700→emit change180()；否则→emit change0()。

3. 界面响应：接收信号后，调用 view->rotate(180)实现旋转/恢复，更新 is180/is0 标记。

核心.cpp 文件代码注解如下

主逻辑核心：widget.cpp

```
#include "widget.h"

// 主窗口构造函数：初始化UI、设备、线程、信号槽
Widget::Widget(QWidget* parent)
    : QMainWindow(parent)
{
```

```
    // 创建图形场景和视图（用于界面旋转）
    QGraphicsScene* scene = new QGraphicsScene(this);
    view = new QGraphicsView(scene, this);
    customWidget = new CustomWidget();  // 创建自定义多页面组件
    QScreen* screen = QGuiApplication::primaryScreen();
    QRect screenGeometry = screen->geometry();
    int screenWidth = screenGeometry.width();
    int screenHeight = screenGeometry.height();
    customWidget->resize(screenWidth, screenHeight);
    customWidget->showFullScreen();  // 全屏显示

    // 禁用滚动条，添加自定义组件到场景
    view->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    view->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    proxyWidget = scene->addWidget(customWidget);
    proxyWidget->setTransformOriginPoint(proxyWidget->boundingRect().center());  // 旋
转原点为中心
    proxyWidget->setZValue(100);  // 置顶显示

    // 初始化加速度传感器（界面翻转用）
    lis3dh = new Lis3dhDevice();
    thread3dh = new QThread();
    lis3dh->moveToThread(thread3dh);
    thread3dh->start();  // 启动传感器线程
    // 绑定传感器线程启动信号→读取数据
    connect(thread3dh, &QThread::started, lis3dh, &Lis3dhDevice::run);
    // 传感器读取失败→安全退出线程
    connect(lis3dh, &Lis3dhDevice::stopthread, [&]() {
        if (thread3dh->isRunning()) {
            lis3dh->changeRunningState(false);
            thread3dh->quit();
        }
        });
    // 绑定界面翻转信号
    connect(lis3dh, &Lis3dhDevice::change180, [&]() {
        if (!customWidget->is180) {
            qDebug() << "turn 180";
            this->view->rotate(180);
            update();
            customWidget->is180 = true;
            customWidget->is0 = true;
        }
        });
    connect(lis3dh, &Lis3dhDevice::change0, [&]() {
```

```
        if (customWidget->is0) {
            qDebug() << "turn 0";
            this->view->rotate(180);
            update();
            customWidget->is180 = false;
            customWidget->is0 = false;
        }
        });

    // 绑定传感器数据信号→数据处理槽函数
    connect(customWidget->adc, &mixAdcDevice::sendData, this, &Widget::recvData);
    // 退出按钮→关闭程序
    connect(customWidget->exitButton, &QPushButton::clicked, this,
&Widget::exitBtnClicked);
    // 页面切换→线程调度
    connect(customWidget, &CustomWidget::currentChanged, this,
&Widget::handleIndexChange);
    // 温度数据→更新仪表盘
    connect(customWidget->tem, &temDevice::sendData, this, &Widget::updateCircular);
    // 继电器按钮→控制槽函数
    connect(customWidget->relayButton, &QPushButton::clicked, this,
&Widget::relayButtonClicked);
    // 光电开关状态→更新图标
    connect(customWidget->light, &lightElectric::sendState, this,
&Widget::handleLightElectricState);
    // 蜂鸣器按钮→控制槽函数
    connect(customWidget->beepBtn, &QPushButton::clicked, this,
&Widget::beepBtnClicked);

    setCentralWidget(view);  // 设置视图为中心控件
}

// 析构函数：释放所有设备资源
Widget::~Widget()
{
    customWidget->adc->closeAdcFd();
    delete customWidget->adc;
    customWidget->tem->closeTemFd();
    delete customWidget->tem;
    customWidget->relay->closeRelayFd();
    delete customWidget->relay;
    customWidget->light->closeLightElectric();
    delete customWidget->light;
}
```

```
// 页面切换槽函数：线程安全调度传感器任务
void Widget::handleIndexChange(int index) {
    // 停止所有传感器线程和设备操作
    customWidget->adc->changeThreadState(false);
    customWidget->tem->changeThreadState(false);
    customWidget->light->changeLightState(false);
    // 关闭设备文件描述符
    customWidget->adc->closeAdcFd();
    customWidget->tem->closeTemFd();

    // 根据页面索引启动对应任务
    switch (index) {
        // ADC类传感器页面（酒精/A、燃气/S、火焰/F、光强/L）
    case 0: case 1: case 5: case 7:
        customWidget->threadPool->waitForDone();  // 等待之前任务结束
        customWidget->adc->changeThreadState(false);
        customWidget->adc->closeAdcFd();
        // 启动ADC读取任务，传入传感器类型
        customWidget->threadPool->start(new ReadAdcDataTask(
            customWidget->adc,
            index == 0 ? "A" : (index == 1 ? "S" : (index == 5 ? "F" : "L"))
        ));
        break;
        // 温度传感器页面
    case 2:
        customWidget->tem->changeThreadState(true);
        customWidget->threadPool->start(new ReadTemDataTask(customWidget->tem));
        break;
        // 光电开关页面
    case 4:
        customWidget->light->changeLightState(true);
        customWidget->threadPool->start(new ReadLightStateTask(customWidget->light));
        break;
    default:
        break;
    }
}

// 接收ADC传感器数据，分发到对应页面更新UI
void Widget::recvData(int data)
{
    int index = customWidget->currentIndex();
    switch (index) {
```

```
    case 0:  // 酒精页面→更新LED状态
        updateAlcoholLedState(data);
        break;
    case 1:  // 燃气页面→更新折线图
        updateSeries(data);
        break;
    case 5:  // 火焰页面→更新动图状态
        updateFrameState(data);
        break;
    case 7:  // 光强页面→更新LCD
        updateLightLCD(data);
        break;
    default:
        break;
    }
}

// 更新燃气折线图：最多显示10个数据点
void Widget::updateSeries(int data)
{
    if (customWidget->count > 10) {
        customWidget->chart->scroll(1, 0);  // 滚动图表
        customWidget->chart->axisX()->setRange(customWidget->count - 9,
customWidget->count);  // 调整X轴范围
    }
    customWidget->gasSeries->append(customWidget->count, data);  // 添加数据点
    customWidget->count++;
}

// 更新温度半圆仪表盘
void Widget::updateCircular(float data)
{
    int updateData = static_cast<int>(data);
    customWidget->temCircular->changeValue(updateData);  // 刷新仪表盘指针和数值
}

// 蜂鸣器控制槽函数
void Widget::beepBtnClicked()
{
    if (!customWidget->isBeepOn) {
        customWidget->beep->changeBeepState(1);  // 开启蜂鸣器
        customWidget->isBeepOn = true;
        customWidget->beepBtn->setIcon(QIcon(":/icon/icon/beepmax.png"));  // 切换图标
        customWidget->beepBtn->setIconSize(QSize(300, 300));
```

```
    }
    else {
        customWidget->beep->changeBeepState(0);  // 关闭蜂鸣器
        customWidget->isBeepOn = false;
        customWidget->beepBtn->setIcon(QIcon(":/icon/icon/beepoff.png"));
        customWidget->beepBtn->setIconSize(QSize(300, 300));
    }
}

// 退出按钮槽函数：释放资源并退出程序
void Widget::exitBtnClicked()
{
    customWidget->beep->closeBeepFd();  // 关闭蜂鸣器设备文件
    this->close();
    QApplication::quit();
}

// 更新酒精检测LED状态
void Widget::updateAlcoholLedState(int data)
{
    if (data > 140) {  // 浓度超标→红色
        customWidget->Alcohol->setLedColor("red");
        customWidget->alcoholText->setText("酒精浓度超标");
    }
    else {  // 未超标→绿色
        customWidget->Alcohol->setLedColor("green");
        customWidget->alcoholText->setText("酒精浓度未超标");
    }
}

// 更新火焰检测状态（动图/静态图）
void Widget::updateFrameState(int data)
{
    if (data > 600) {  // 检测到火焰→播放动图
        customWidget->frameLabel->setMovie(customWidget->frameMovie);
        customWidget->frameMovie->start();
        customWidget->frameLabelText->setText("检测到火焰");
    }
    else {  // 未检测到→静态图
        customWidget->frameMovie->stop();
        QPixmap frameoffPixmap(":/icon/icon/frameoff.png");
        QPixmap scaledPixmapFrameOff = frameoffPixmap.scaled(300, 300,
Qt::KeepAspectRatio, Qt::SmoothTransformation);
        customWidget->frameLabel->setPixmap(scaledPixmapFrameOff);
```

```
        customWidget->frameLabelText->setText("未检测到火焰");
    }
}

// 更新光强LCD显示
void Widget::updateLightLCD(int data)
{
    customWidget->lightLCD->display(data);
}

// 继电器控制槽函数
void Widget::relayButtonClicked()
{
    QPixmap relayLabelOnPixmap(":/icon/icon/relayon.png");
    QPixmap relayLabelPixmap(":/icon/icon/relay.png");
    QPixmap newrelayOn = relayLabelOnPixmap.scaled(150, 150);
    QPixmap newrelay = relayLabelPixmap.scaled(150, 150);

    if (!customWidget->relayButtonState)  // 未开启→打开
    {
        customWidget->relay->changeRelayState(1);
        customWidget->relayLabel->setPixmap(newrelayOn);
        customWidget->relayButton->setText("关闭");
        customWidget->relayButtonState = true;
    }
    else {  // 已开启→关闭
        customWidget->relay->changeRelayState(0);
        customWidget->relayLabel->setPixmap(newrelay);
        customWidget->relayButton->setText("打开");
        customWidget->relayButtonState = false;
    }
}

// 处理光电开关状态更新
void Widget::handleLightElectricState(int state)
{
    QPixmap lightLabelPixmap;
    if (state == 0) {  // 未遮挡→默认图标
        lightLabelPixmap.load(":/icon/icon/norlightswitch.png");
        customWidget->lightLabelText->setText("未检测遮挡");
    }
    else {  // 遮挡→切换图标
        lightLabelPixmap.load(":/icon/icon/lightswitch.png");
        customWidget->lightLabelText->setText("检测遮挡");
```

```
    }
    QPixmap scaledLightLabelPixmap = lightLabelPixmap.scaled(200, 200,
Qt::KeepAspectRatio);
    customWidget->lightLabel->setPixmap(scaledLightLabelPixmap);
    }
```

ADC 类传感器核心：mixadcdevice.cpp

```
#include "mixadcdevice.h"

// 筛选函数：scandir遍历目录时，保留IIO设备节点（d_name以"iio:"开头的符号链接）
int mixAdcDevice::filter(const dirent* entry)
{
    return entry->d_type == DT_LNK && strncmp(entry->d_name, "iio:", 4) == 0;
}

// 构造函数：初始化ADC传感器对象
mixAdcDevice::mixAdcDevice(QObject* parent) : QObject(parent)
{
}

// 打开ADC设备文件：根据功能类型拼接设备路径
void mixAdcDevice::openAdcDevice(QString function)
{
    struct dirent** namelist;
    // 遍历/sys/bus/iio/devices/目录，筛选IIO设备
    int n = scandir("/sys/bus/iio/devices/", &namelist, filter, alphasort);
    if (n < 0) {
        perror("scandir");
    }
    for (int i = 0; i < n; i++) {
        char device_path[1024];
        // 根据功能类型拼接不同通道的设备文件路径
        if (function.compare("A") == 0) {  // 酒精→voltage4
            snprintf(device_path, sizeof(device_path),
"/sys/bus/iio/devices/%s/in_voltage4_raw", namelist[i]->d_name);
        }
        else if (function.compare("F") == 0) {  // 火焰→voltage3
            snprintf(device_path, sizeof(device_path),
"/sys/bus/iio/devices/%s/in_voltage3_raw", namelist[i]->d_name);
        }
        else if (function.compare("S") == 0) {  // 燃气→voltage0
            snprintf(device_path, sizeof(device_path),
"/sys/bus/iio/devices/%s/in_voltage0_raw", namelist[i]->d_name);
        }
```

```
        else if (function.compare("L") == 0) {  // 光强→voltage3
            snprintf(device_path, sizeof(device_path),
"/sys/bus/iio/devices/%s/in_voltage3_raw", namelist[i]->d_name);
        }
        // 检查设备文件是否存在
        struct stat device_file_stat;
        if (stat(device_path, &device_file_stat) == 0) {
            qDebug() << "IIO device with 'in_voltage_raw': " << device_path;
            adcfd = open(device_path, O_RDWR);  // 打开设备文件（读写模式）
            qDebug() << "device_path" << device_path;
            if (adcfd < 0) {
                perror("open device");  // 打开失败输出错误
            }
            else {
                qDebug() << "device openning";
            }
        }
        else {
            qDebug() << "No 'in_voltage_raw' in device directory:  " << device_path;
        }
        free(namelist[i]);  // 释放目录项内存
    }
    free(namelist);  // 释放目录列表内存
}

// 读取ADC数据：循环读取直到isAdcRunning为false
void mixAdcDevice::readData(QString function)
{
    closeAdcFd();  // 先关闭之前的设备文件
    while (isAdcRunning) {  // 运行标志为true时持续读取
        unsigned char data[20];
        int newdata = 0;
        openAdcDevice(function);  // 打开对应功能的ADC设备
        int err = read(adcfd, &data, sizeof(data));  // 读取原始数据
        if (err > 0) {
            int i = 0;
            // 解析数据：仅保留数字字符，转换为整数
            for (i = 0; i < err; i++) {
                if (data[i] >= '0' && data[i] <= '9') {
                    newdata = newdata * 10 + (data[i] - 48);
                }
            }
        }
        else {
```

```
            qDebug() << "read failed err:" << err;
        }
        int senddata = newdata * 3.8;  // 数据缩放，匹配实际物理量
        emit sendData(senddata);  // 发送数据给UI
        QThread::usleep(500000);  // 间隔500ms，减少CPU占用
        closeAdcFd();  // 关闭设备文件，避免资源泄漏
    }
}

// 改变线程运行状态：控制数据读取循环的启停
void mixAdcDevice::changeThreadState(bool state)
{
    isAdcRunning = state;
}

// 关闭ADC设备文件描述符
void mixAdcDevice::closeAdcFd()
{
    close(adcfd);
    adcfd = 0;
    }
```

温度传感器核心：temdevice.cpp

```
#include "temdevice.h"

// 构造函数：初始化温度传感器对象
temDevice::temDevice(QObject* parent) : QObject(parent)
{
}

// 打开I2C设备文件（温度传感器LM75连接在i2c-4总线）
void temDevice::openTemDevice()
{
    temfd = open("/dev/i2c-4", O_RDWR);  // 以读写模式打开I2C设备
    if (temfd < 0)
    {
        perror("open error");  // 打开失败输出错误
    }
}

// 读取温度数据：通过I2C接口与LM75通信
void temDevice::readData()
{
    openTemDevice();  // 打开I2C设备
```

```
int ret = 0;
struct i2c_rdwr_ioctl_data lm75_data;  // I2C读写消息结构体
short temp_val = 0;

// 初始化I2C消息参数
lm75_data.nmsgs = 2;
lm75_data.msgs = (struct i2c_msg*)malloc(lm75_data.nmsgs * sizeof(struct i2c_msg));
if (!lm75_data.msgs)
{
    perror("malloc error");
    exit(1);
}
ioctl(temfd, I2C_TIMEOUT, 1);/* 设置超时时间1秒 */
ioctl(temfd, I2C_RETRIES, 2);/* 设置重试次数2次 */
sleep(1);

// 循环读取温度，直到isTemRunning为false
while (isTemRunning) {
    lm75_data.nmsgs = 2;
    // 第一个消息：写入LM75数据寄存器地址（0x0）
    (lm75_data.msgs[0]).len = 1;
    (lm75_data.msgs[0]).addr = 0x4f;  // LM75设备地址0x4f
    (lm75_data.msgs[0]).flags = 0;/* 写操作 */
    (lm75_data.msgs[0]).buf = (unsigned char*)malloc(2);
    (lm75_data.msgs[0]).buf[0] = 0x0;/* 温度数据寄存器地址 */

    // 第二个消息：读取温度数据（2字节）
    (lm75_data.msgs[1]).len = 2;
    (lm75_data.msgs[1]).addr = 0x4f;
    (lm75_data.msgs[1]).flags = I2C_M_RD;/* 读操作 */
    (lm75_data.msgs[1]).buf = (unsigned char*)malloc(2);
    (lm75_data.msgs[1]).buf[0] = 0;/* 初始化读缓冲 */
    (lm75_data.msgs[1]).buf[1] = 0;

    // 执行I2C读写操作
    ret = ioctl(temfd, I2C_RDWR, (unsigned long)&lm75_data);
    if (ret < 0)
    {
        perror("ioctl error2");
        return;
    }

    // 数据转换：拼接2字节数据，处理正负温度
    temp_val = (lm75_data.msgs[1]).buf[0] << 8 | (lm75_data.msgs[1]).buf[1];
```

```
        if (temp_val >> 15)  // 负温度处理
            temp_val = (~(temp_val - 0x80) >> 7);
        else  // 正温度处理
            temp_val = temp_val >> 7;

        printf("temp =  %01f\n", (float)temp_val / 2);  // 调试输出温度
        emit sendData((float)temp_val / 2);  // 发送温度值（÷2得到实际温度）
        usleep(100000);  // 间隔100ms，减少CPU占用
    }
    closeTemFd();  // 停止读取后关闭设备文件
}

// 改变线程运行状态：控制温度读取循环的启停
void temDevice::changeThreadState(bool state)
{
    isTemRunning = state;
}

// 关闭I2C设备文件描述符
void temDevice::closeTemFd()
{
    close(temfd);
    }
```

半圆仪表盘 UI 核心：semicircular.cpp（关键绘制函数）

```
#include "semicircular.h"

// 构造函数：初始化半圆仪表盘
SemiCircular::SemiCircular(QWidget* parent)
    : QWidget(parent)
{
}

// 设置仪表盘数值，计算旋转角度并刷新
void SemiCircular::changeValue(int newvalue)
{
    value = newvalue;
    degRotate = static_cast<int>(value * 1.8);  // 0-100℃对应0-180度，比例1.8
    update();  // 触发paintEvent重绘
}

// 绘制渐变区域（仪表盘背景色）
void SemiCircular::drawGradientArea(QPainter& painter, int radius)
{
```

```
    QRect rect(-radius, -radius, 2 * radius, 2 * radius);
    QConicalGradient Conical(0, 0, 0);  // 锥形渐变（中心原点，0度起始）
    // 配置渐变颜色（红→黄→绿→浅蓝）
    Conical.setColorAt(0.1, QColor(255, 34, 54));//红色
    Conical.setColorAt(0.2, QColor(222, 201, 21));//黄色
    Conical.setColorAt(0.3, QColor(28, 198, 52));//绿色
    Conical.setColorAt(0.5, QColor(61, 79, 255)); //浅蓝色
    painter.setBrush(Conical);
    // 绘制上半圆：startAngle=180*16，spanAngle=-degRotate*16（Qt角度单位为1/16度）
    painter.drawPie(rect, 180 * 16, -(degRotate) * 16);
}

// 绘制刻度线（大刻度每5个小刻度一个）
void SemiCircular::drawScale(QPainter& painter, int radius)
{
    // 动态计算刻度尺寸（基于控件基准大小）
    int scaleLengthSmall = baseSize * 0.01;
    int scaleWidthSmall = baseSize * 0.002;
    int scaleLengthBig = baseSize * 0.02;
    int scaleWidthBig = baseSize * 0.004;

    // 小刻度路径（矩形）
    QPainterPath pointPath_small;
    pointPath_small.moveTo(-scaleWidthSmall, -scaleWidthSmall);
    pointPath_small.lineTo(scaleWidthSmall, -scaleWidthSmall);
    pointPath_small.lineTo(scaleWidthSmall, scaleLengthSmall);
    pointPath_small.lineTo(-scaleWidthSmall, scaleLengthSmall);

    // 大刻度路径（矩形）
    QPainterPath pointPath_big;
    pointPath_big.moveTo(-scaleWidthBig, -scaleWidthBig);
    pointPath_big.lineTo(scaleWidthBig, -scaleWidthBig);
    pointPath_big.lineTo(scaleWidthBig, scaleLengthBig);
    pointPath_big.lineTo(-scaleWidthBig, scaleLengthBig);

    // 绘制0-180度刻度（每1.8度一个小刻度，共100个）
    for (int i = 0; i <= 100; ++i) {
        QPointF point(0, 0);
        painter.save();
        double angle = i * 1.8;  // 计算当前角度
        double radian = qDegreesToRadians(angle);
        // 计算刻度位置（上半圆）
        point.setX(radius * qCos(radian));
        point.setY(-radius * qSin(radian));
```

```
        painter.translate(point.x(), point.y());  // 平移到刻度位置
        painter.rotate(90 - angle);  // 旋转刻度使其垂直于半径
        painter.setBrush(QColor(255, 255, 255)); // 白色刻度
        if (i % 5 == 0) {
            painter.drawPath(pointPath_big);  // 每5个小刻度画一个大刻度
        }
        else {
            painter.drawPath(pointPath_small);  // 小刻度
        }
        painter.restore();
    }
}

// 绘制刻度数字（0-100，每10个单位一个）
void SemiCircular::drawNumScale(QPainter& painter, int radius)
{
    painter.setPen(QColor(255, 255, 255));  // 白色数字
    int fontSize = max(8, baseSize * 0.016);  // 动态字体大小（最小8号）
    QFont font;
    font.setFamily("Noto Sans CJK SC Regular");
    font.setPointSize(fontSize);
    font.setBold(true);
    painter.setFont(font);

    // 绘制0、10、20...100数字
    for (int i = 0; i <= 100; i += 10) {
        QPointF point(0, 0);
        painter.save();
        double angle = 180 - i * 1.8;  // 计算数字位置角度
        double radian = qDegreesToRadians(angle);
        point.setX(radius * qCos(radian));
        point.setY(-radius * qSin(radian));  // 上半圆位置
        painter.translate(point.x(), point.y());  // 平移到数字位置
        painter.rotate(-angle);  // 数字垂直显示
        painter.drawText(-30, 0, 50, 15, Qt::AlignCenter, QString::number(i));  // 绘制
数字
        painter.restore();
    }
    painter.setPen(Qt::NoPen);
}

// 绘制实时温度数据
void SemiCircular::drawRealTimeData(QPainter& painter, int radius)
{
```

```
    painter.save();
    painter.setPen(QColor(255, 255, 255));  // 白色文字
    int fontSize = max(8, baseSize * 0.032);  // 动态字体大小（突出显示）
    QFont font;
    font.setFamily("Noto Sans CJK SC Regular");
    font.setPointSize(fontSize);
    font.setBold(true);
    painter.setFont(font);
    QString displayText = QString::number(value) + " ℃";  // 温度+单位
    painter.drawText(-75, -radius - 80, 150, 100, Qt::AlignCenter, displayText);  // 居
中显示
    painter.restore();
}

// 绘制指针（指示当前温度）
void SemiCircular::drawPointer(QPainter& painter, int baseSize)
{
    int radius = baseSize / 2;
    // 指针尺寸比例因子（基于半径）
    double pointerTip = 1.8;
    double pointerBaseWidth = 0.04;
    double pointerBaseLength = 0.001;
    double pointerSideLength = 0.001;

    QPainterPath pointPath;
    pointPath.moveTo(0, -radius * pointerTip);  // 指针尖端
    pointPath.lineTo(radius * pointerBaseWidth, -radius + radius * pointerBaseLength);
// 指针右侧
    pointPath.lineTo(radius * pointerSideLength, -radius * pointerSideLength);
    pointPath.lineTo(-radius * pointerBaseWidth, -radius + radius * pointerBaseLength);
// 指针左侧
    pointPath.closeSubpath();  // 封闭路径

    painter.save();
    painter.rotate(degRotate - 90);  // 指针旋转到对应角度（-90校准起始位置）
    painter.setBrush(QColor(255, 34, 54));  // 红色指针（醒目）
    painter.drawPath(pointPath);  // 绘制指针
    painter.restore();
}

// 重绘事件：整合所有绘制函数，绘制完整仪表盘
void SemiCircular::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
```

```
int width = this->width();
int height = this->height() - 100;  // 预留底部文字空间
baseSize = std::min(width, height);
const int minRadius = 50;
int radius = std::max(minRadius, baseSize / 2);  // 仪表盘半径（最小50）

painter.translate(width / 2, height * 0.6);  // 平移原点到仪表盘中心下方
painter.setRenderHint(QPainter::Antialiasing, true);  // 开启抗锯齿
painter.setPen(Qt::NoPen);  // 无轮廓线

// 逐层绘制仪表盘
drawCircle(painter, radius - baseSize * 0.0375);  // 渐变外扇形
drawGradientArea(painter, radius - baseSize * 0.05);  // 动态渐变区域
drawOutMiddleCircle(painter, radius - baseSize * 0.025);  // 外中圆
drawScale(painter, radius - baseSize * 0.1);  // 刻度线
drawNumScale(painter, radius - baseSize * 0.125);  // 刻度数字
drawOutermostLine(painter, radius - baseSize * 0.04375);  // 最外细圆线
drawMiddleBiggestCircle(painter, radius - baseSize * 0.2175);  // 中间大圆
drawMiddleCircle(painter, radius - baseSize * 0.22625);  // 中间圆
drawPointer(painter, radius - baseSize * 0.1625);  // 温度指针
drawMiddleLittleCircle(painter, radius - baseSize * 0.235);  // 中间小圆
drawRealTimeData(painter, radius - baseSize * 0.50375);  // 实时温度文字
}
```

h) 录音和放音

任选 3 个工程文件（电机综合，传感器综合必选 1；LED 灯，蜂鸣器，按键最多选 1 个），进行代码剖析

3. 代码剖析要求：

a) 作图阐述每个工程中各个文件之间的关系；

b) 基于实验现象，结合实验手册和相应代码，分析阐述其工作原理（可采用作图，注解，演示截屏，文字说明等多种方式）。

c) 对于工程中的核心.cpp 文件进行代码注解

4. 作业提交：

a) 命名：姓名+学号+3.pdf

b) 提交至数字化教学平台：course.xmu.edu.cn

c) 截止时间：11 月 23 晚 12:00