重　　庆　　大　　学

国家卓越工程师学院

课 程 设 计 报 告

课程名称：　定量工程设计 2

年　　级：　2023 级

专业班级：　明月科创实验班

学生姓名：　贺禄文

学　　号：　20234232

完成日期：　2025.6.9

成　　绩：

教师签名：

重庆大学　制

二〇二一年一月

目录

1. 上位机 - 1 -
1.1. 代码功能简述 - 1 -
1.2. 代码工作流程 - 1 -
1.2.1. 串口读取线程 - 1 -
1.2.2. 数据处理线程 - 1 -
1.2.3. GUI 线程 - 2 -
1.2.4. 程序退出流程 - 2 -
1.2.5. 流程图 - 3 -
1.3. 亮点与设计原因 - 5 -
1.3.1. 多线程架构 - 5 -
1.3.2. 基于包头和长度的串口数据读取 - 5 -
1.3.3. Pan-Tompkins 算法的简化实现 - 5 -
1.3.4. 动态调整 R 波检测最小间隔 - 5 -
1.3.5. 基于时间戳的精确心率计算 - 5 -
1.3.6. 全面的信号处理阶段显示 - 5 -
1.3.7. 线程退出机制 - 6 -
1.3.8. R 波峰值的准确抓取与实时显示 - 6 -
1.3.9. GUI 交互中的暂停与一键停止功能 - 6 -
1.4. 整体效果 - 6 -
1.5. 小结 - 7 -
2. 下位机 - 8 -
2.1. 硬件工作原理与信号读取 - 8 -
2.2. 信号预处理过程 - 9 -
2.3. 心跳检测算法详细说明 - 10 -
2.4. 心率计算方式与 DWT 计时器的作用 - 11 -
2.5. FFT 频谱分析实现方式 - 11 -
2.6. 数据发送与上位机处理 - 12 -
2.7. LCD 屏幕波形显示方式 - 12 -
2.8. 蜂鸣器心跳提示控制逻辑 - 12 -
2.9. 项目测试结果与信号处理效果分析 - 13 -
3. 项目心得 - 14 -
4. 附录 - 15 -
4.1. 上位机代码 - 15 -
4.1.1. 串口读取线程 - 15 -
4.1.2. 数据处理线程 - 17 -
4.1.3. GUI 显示线程 - 22 -
4.1.4. Pan-Tompkins 算法处理函数 - 28 -
4.1.5. 主函数进程 - 31 -
4.2. 下位机代码 - 33 -
4.2.1. 心电数据转换程序 - 33 -
4.2.2. IIR 滤波器实现 - 34 -
4.2.3. FIR 滤波器实现 - 34 -
4.2.4. FFT 实现 - 38 -
4.2.5. 数据发送协议实现 - 39 -

1. 上位机

1.1. 代码功能简述

整套代码用于实时心电信号（ECG）监测和分析。先能够从串口接收原始 ECG 数据，通过多线程处理实现数据的实时读取、信号滤波、R 波检测、心率计算，并将处理后的结果以图形化的方式实时显示出来，包括原始时域波形、滤波后的时域波形、R 波峰值显示、心电图机风格的滚动显示，以及不同处理阶段的频谱分析。

1.2. 代码工作流程

整个代码的工作流程基于多线程并发执行，主要包含三个核心线程：

1.2.1. 串口读取线程

该线程独立运行，负责监听指定串口。它以非阻塞的方式从串口读取原始字节数据，并通过特定的包头和数据长度进行数据包解析。成功解析出的浮点型 ECG 值和当前时间戳会被封装成元组（主要用于心率的计算），并放入 raw_data_queue 队列中，供数据处理线程消费。此线程具备错误处理机制，能够捕获串口读取和数据解包异常，并尝试刷新输入缓冲区以恢复。

1.2.2. 数据处理线程

该线程从 raw_data_queue 中获取原始 ECG 数据点和时间戳，并维护一个固定大小的滑动窗口（窗口大小为 700 点），通过将新接收的数据点添加到窗口尾部并移除最旧的数据点，确保窗口内始终包含最新一批数据。当滑动窗口首次填满后，开始对窗口内的 ECG 信号进行处理。处理阶段包括：采用一阶 IIR 高通滤波器去除基线漂移（由呼吸、电极移动或身体运动等引起的低频干扰，通常低于 0.5Hz），使 QRS 波群更突出；使用 scipy.signal.iirnotch 设计的 IIR 陷波滤波器去除 50Hz 工频干扰（由交流电源引起的固定频率噪声），精确抑制该频率而不影响有用信号；再应用 FIR 低通滤波器（firls 设计，截止频率 35Hz）去除高频噪声，获得平滑的 ECG 信号。之后进行心率的计算，其中 Pan-Tompkins R 波检测是核心分析步骤，依次包括：使用 3 阶巴特沃斯带通滤波器（5Hz-15Hz）突出 QRS 波群（其能量集中于此范围）；采用五点微分滤波器增强 QRS 波的陡峭程度，抑制平缓的 P 波和 T 波；对信号平方，使幅值变为正并进一步突出 QRS 波；通过 150ms 滑动窗口积分平滑信号，形成更明显的峰；利用 scipy.signal.find_peaks 检测峰值（通过 height 和 distance 参数模拟自适应阈值）；从积分信号峰值回溯到原始或预处理信号，在前后 100ms 窗口内定位精确 R 波位置；根据心率动态调整 R 波最小间隔以适应变化。心率计算基于 RR 间期（R 波间时间间隔），通过两个 R 波峰值的时间戳进行计算。之后对原始信号、直流移除后信号、50Hz 陷波后信号及低通滤波后信号分别进行快速傅里叶变换，计

算单边幅值谱以供频域分析。处理后的数据（包括滤波信号、R 波位置、心率、各阶段频谱等）存储在 latest_batch_data 共享变量中，通过 batch_data_lock 同步；最新的单点滤波数据放入 processed_data_queue 队列，供 GUI 线程实时滚动显示。该线程还支持暂停/恢复功能，通过 is_paused 标志和 pause_condition 条件变量控制。

1.2.3. GUI 线程

该线程负责初始化 Tkinter 窗口和 Matplotlib 图表，创建了两个独立的 Tkinter 窗口：一个用于显示时域波形、原始 ECG 和心电图机风格的滚动显示，另一个用于显示不同处理阶段的频域频谱。实时滚动显示通过 update_plots_gui 函数实现，该函数不断从 processed_data_queue 获取最新的单点处理数据，更新 ecg_scroll_buffer，从而在“心电图机显示”子图中实现波形的实时滚动，滚动缓冲区大小为 60 秒的数据量。此外，update_plots_gui 函数还以 0.05 秒的频率从 latest_batch_data 读取最新的批处理结果，更新原始 ECG 波形图、滤波后的时域波形图（标注 R 波位置并显示心率）、四个频域子图（展示不同处理阶段的频谱，Y 轴以 dB 表示幅值）以及最新的 R 波峰值信息。通过 root.after(10, update_plots_gui)实现图表的周期性刷新，确保界面实时且流畅。同时，提供了“暂停”和“一键结束所有进程”按钮，分别用于控制数据处理线程的暂停/恢复以及程序的优雅退出。

1.2.4. 程序退出流程

当用户点击“一键结束”按钮时，on_closing 函数会被调用。它会设置 stop_event 线程事件，并通知 pause_condition（唤醒所有等待中的线程）。主程序会等待串口读取线程和数据处理线程终止，然后程序退出。

1.2.5. 流程图

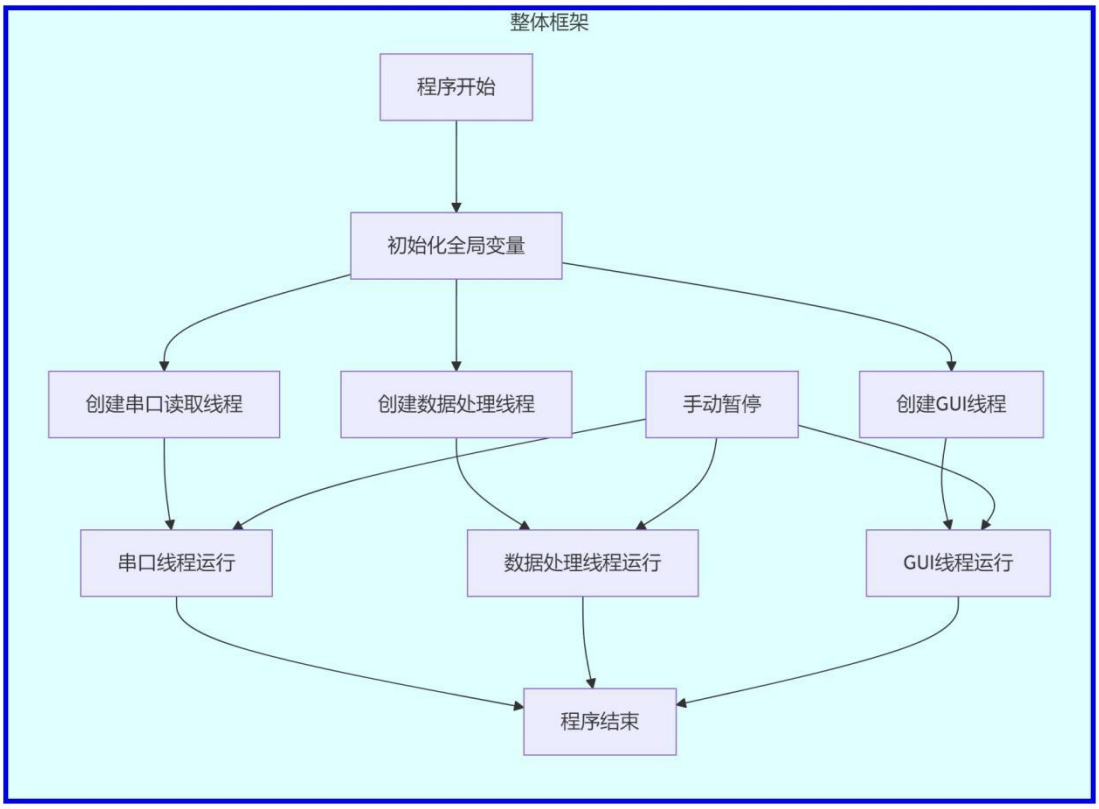

图 整体框架

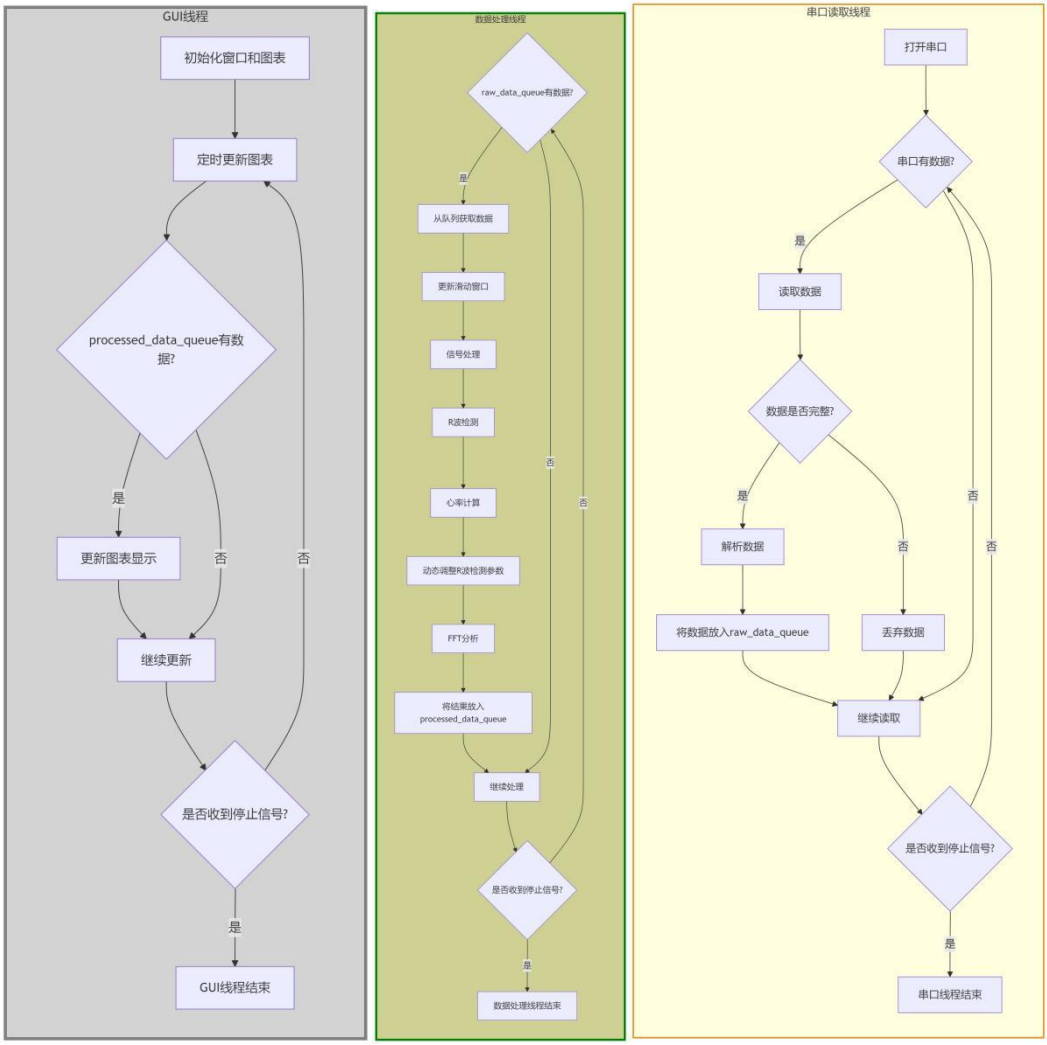
GUI线程
初始化窗口和图表
定时更新图表
processed_data_queue有数据?
是
否
更新图表显示
否
继续更新
是否收到停止信号?
否
是
GUI线程结束
数据处理线程
raw_data_queue有数据?
是
否
从队列获取数据
更新滑动窗口
信号处理
R波检测
心率计算
动态调整R波检测参数
FFT分析
将结果放入processed_data_queue
继续处理
是否收到停止信号?
否
是
数据处理线程结束
串口读取线程
打开串口
串口有数据?
是
否
读取数据
数据是否完整?
是
否
解析数据
否
将数据放入raw_data_queue
丢弃数据
继续读取
是否收到停止信号?
否
是
串口线程结束

图 各个线程的流程图

1.3. 亮点与设计原因

1.3.1. 多线程架构

将串口数据读取、数据处理和 GUI 显示划分到独立的线程中，有效地解决了实时系统中常见的“数据 I/O 阻塞 UI”问题。串口读取是 I/O 密集型操作，信号处理是计算密集型操作，如果都在主线程中执行，会导致 GUI 更新速度为实时信号的 1/3。

1.3.2. 基于包头和长度的串口数据读取

串口读取线程并非简单地连续读取字节流，而是采用了基于固定包头（0XA5）和预定义数据长度（4 字节浮点数）的数据帧解析机制。这意味着程序能够识别和提取完整、有效的数据包，即使在传输过程中出现数据错位或部分丢失，也能尝试重新同步。同时，代码内置了健壮的 try-except 异常处理，用于捕获串口读取和数据解包过程中可能发生的错误（如 serial.SerialException 或 struct.error），并在发生错误时，通过调用 ser.flushInput()来清空输入缓冲区，尝试恢复正常的通信状态。

1.3.3. Pan-Tompkins 算法的简化实现

Pan-Tompkins 算法是 ECG R 波检测的经典且成熟的算法，其分步处理能够有效突出 QRS 波群并抑制噪声，适合实时应用。代码中我们使用了 Pan-Tompkins R 波检测算法的经典步骤（带通滤波、微分、平方、积分），并利用 scipy.signal.find_peaks 简化了自适应阈值和回溯的复杂性，使其不仅易于实现，还便于理解。

1.3.4. 动态调整 R 波检测最小间隔

我们根据实时计算的心率动态调整 min_peak_distance_sec 参数。当心率过快时，会适当减小最小间隔，以避免漏检；当心率过慢时，则会适当增大最小间隔，以减少误报。

1.3.5. 基于时间戳的精确心率计算

心率的计算并非简单地基于采样点的序号差，而是精确地利用了每个 ECG 数据点被采集并传输到电脑的时间戳。在实际测试中，特别是当单片机功能或负载变化时，我们发现数据传输到上位机的频率存在很大不均匀性。通过直接使用时间戳，即使采样间隔存在微小差异，也能准确计算出 R 波间期（RR 间期）和心率，从而显著减少因采样不均匀带来的误差。

1.3.6. 全面的信号处理阶段显示

在频域图表中，分别展示了原始信号、DC 移除后、50Hz 陷波后以及低通滤波后的频谱。每个子图都配有明确的标题、轴标签和参考线（如工频线、基线漂移上限等），方便用户分析。

1.3.7. 线程退出机制

使用 threading.Event(stop_event)作为所有线程的退出信号，并在主程序结束时 set()该事件，同时利用 pause_condition.notify_all()唤醒所有可能在等待的线程。最后通过 thread.join()等待线程终止，并设置超时，防止线程意外挂起。

1.3.8. R 波峰值的准确抓取与实时显示

在 R 波检测过程中，我们在代码中不仅能识别出 R 波出现的时机，更进一步地，能够回溯到原始的 ECG 信号中，精确地定位到 R 波的真实峰值（最大值）。这些检测到的 R 波峰值会在实时界面上显示其原始幅值和降噪后的幅值。

1.3.9. GUI 交互中的暂停与一键停止功能

用户界面提供了直观的“暂停”和“一键结束所有进程”按钮。

- “暂停”功能允许用户随时冻结数据处理，方便详细观察当前屏幕上的波形和数据，而无需中断数据流的接收。当再次点击时，处理会从暂停的地方继续。
- “一键结束所有进程”按钮则提供了一个程序退出机制，能够确保所有后台线程（包括串口读取和数据处理线程）安全、有序地终止，避免了资源泄露或程序僵死的问题。

1.4. 整体效果

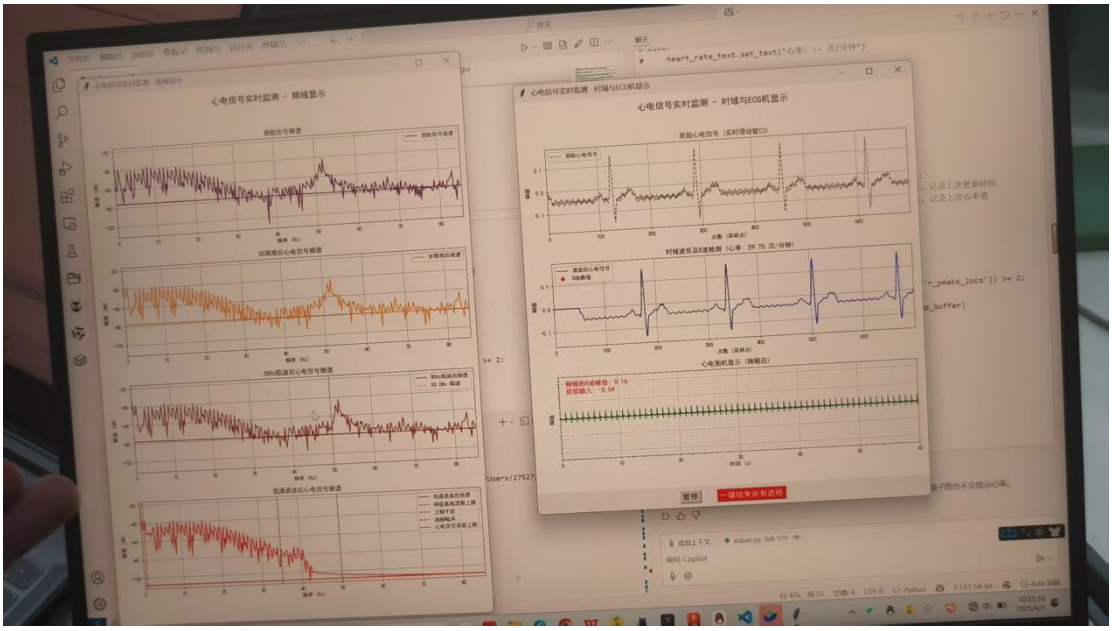

图 PC 端心电数据分析结果

在 PC 端可以在屏幕上打开两个窗口。一个窗口显示在信号处理过程中各个阶段的频域图像，一个窗口显示处理后的心电信号，包括原始信号的图像，滤波后的心电信号，滚动的心电信号和实时获取的 R 波峰值。

整体上图像清晰且数据准确，符合我们的预期效果。

1.5. 小结

整个代码是一个结构良好、功能全面的实时 ECG 监测和分析系统。我们利用 Python 的多线程和队列机制，有效地解决了实时数据流处理中常见的并发和同步挑战。通过实现经典的 Pan-Tompkins R 波检测算法，并辅以多种数字滤波器进行信号预处理和噪声抑制，程序能够准确地提取心电信号中的关键特征（R 波），并计算心率。

2. 下位机

下位机系统硬件部分以 STM32F407 开发板为核心，外接 ADS1296 生物电信号采集芯片、LCD 显示屏、蜂鸣器和串口通信模块等。STM32F407 基于 32 位 Arm Cortex-M4 内核（含 FPU），主频高达 168MHz，提供最高 210 DMIPS 的运算能力，片上 Flash 可达 1MB，SRAM 可达 192KB。ADS1296 是 TI 推出的 6 通道 24 位 ADC 芯片，集成了 ECG 前端放大器、可编程增益、右腿驱动（RLD）放大器和导联脱落检测功能。它通过 SPI 总线与 STM32 通信，片选（CS）和时钟（SCLK）、串行输入（DIN）、串行输出（DOUT）线连接 MCU 的 SPI 引脚，DRDY（数据就绪）脚连接 MCU 的外部中断输入。LCD 屏用于实时显示波形，蜂鸣器用于心跳提示，串口（UART/USB 转串口）则用于向上位机发送数据。

软件方面，本系统主要划分为数据采集模块（ADS1296 初始化、SPI 通信、DRDY 中断处理等）、信号预处理模块（IIR 高通滤波、FIR 低通滤波）、心跳检测与心率计算模块（基于差分、绝对值、积分和自适应阈值的 Pan-Tompkins 算法）、频谱分析模块（使用 CMSIS-DSP FFT）、LCD 绘图模块、蜂鸣器控制模块和上位机通信协议模块等。

2.1. 硬件工作原理与信号读取

ADS1296 芯片每通道具有输入多路开关和可编程增益放大器，可对心电信号进行差分采样。启动采集后，ADS1296 连续不断地进行 Δ Σ 型转换，在内部数字滤波器稳定后，每完成一次转换便在 DRDY 引脚输出一个低电平脉冲。这一 DRDY 下降沿被 STM32 配置为外部中断触发信号，MCU 在中断服务中通过 SPI 读取转换结果寄存器。读取时，每个通道的数据按照 MSB 优先的顺序通过 DOUT 引出。图为示意：MCU 拉低 CS 后，每个 SCLK 下降沿让 ADS1296 将一位数据推至 DOUT，先输出状态字的 24 位数据，然后依次输出各通道的 24 位 ADC 值。

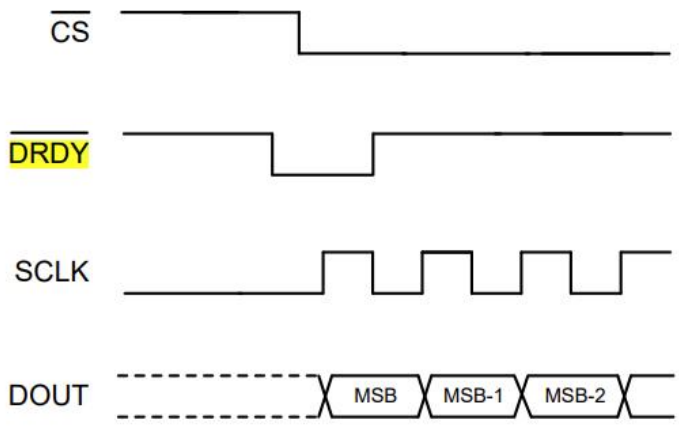

图 DRDY 与数据检索

ADS1296 的 SPI 接口兼容标准 SPI 协议；MCU 通过 SPI 总线发送指令配置寄存器（如设置采样率、增益、偏置等），并启动转换（通过命令或 START 引脚）。当 DRDY 脚输出低电平时，即可知新数据已准备好。之后 STM32 驱动 SCLK 读取数据，同时 DRDY 在第一个 SCLK 下降沿被拉高，数据流从状态字开始依次输出各通道电压值。在硬件连接上，ADS1296 的 6 个输入通道分别连接到被试者的不同心电导联，右腿驱动（RLD）输出连接到人体，用以生成虚拟参考；此外，ADS1296 集成了导联脱落检测功能，可通过注入已知电流或阻抗检测电极是否接触良好。TI 文献指出导联脱落可通过已知激励信号（恒流源或参考电压）注入后，检测输入信号响应是否符合预期来判断。总体上，ADS1296 的内部右腿驱动放大器和导联检测电路使得心电采集更稳健可靠。

ADS1296 每个通道输出为 24 位二进制补码格式（MSB 先传），需要先将 3 字节数据组合成一个带符号的整数（int32_t），再根据公式进行电压换算：

$$V_{in} = \frac{Data_{Raw}}{2^{23}} * \frac{V_{ref}}{Gain}$$

- $Data_{Raw}$：解析后的带符号整数（范围：-8,388,608 到 +8,388,607）
- V_{ref}：参考电压
- Gain:通道增益

2.2. 信号预处理过程

在采样得到原始心电信号后，需要滤除基线漂移和高频噪声。使用一个 IIR 高通滤波器加一个 FIR 低通滤波器完成。

IIR 高通滤波常用于去除直流基线漂移和低频工频干扰，避免心电信号基线飘移导致特征提取错误。可以设计一阶高通 IIR 滤波器。文献指出，巴特沃斯（Butterworth）高通滤波器在医疗应用中运算效率高，能够有效去除基线漂移且对 ST 段影响较小。由于一阶 IIR 计算量低、实时性好，适合在 STM32 上实现。应用高通滤波后，原信号的直流分量和缓慢变化成分被抑制，保留了 QRS 等感兴趣波形。

FIR 低通滤波用于滤除高频干扰（如肌电噪声、量化噪声等）。设计 161 阶有限冲击响应滤波器，通带截止约 40 Hz，过渡带至 60 Hz，用窗函数设计法获得系数。FIR 滤波器系数通过滤波设计工具（MATLAB 的 fir1 函数）获得。由于 FIR 滤波器具有严格的线性相位特性。161 阶滤波器的群延迟为 80 个采样点，这意味着输出波形相对于输入将有固定延迟。在实时系统中可以通过延时补偿加以处理。FIR 低通滤波器稳定性好，不会引入额外振荡，可有效滤除高于设定截

止频率的噪声分量，同时保证 QRS 波形形状保持良好。具体实现时，为了进一步提高运算效率，使用定长环形队列缓存最近 161 个采样点，与预先量化后的系数数组逐点内积；将系数放大 65536，卷积结果再乘以标度系数。

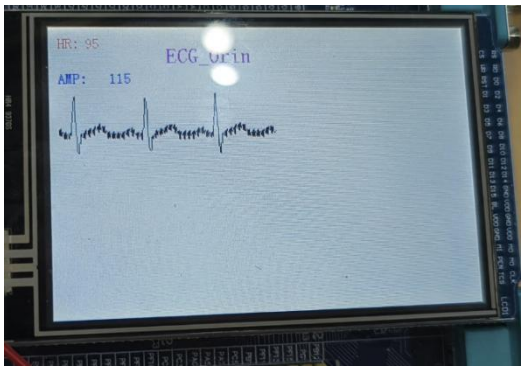

图 原始心电信号

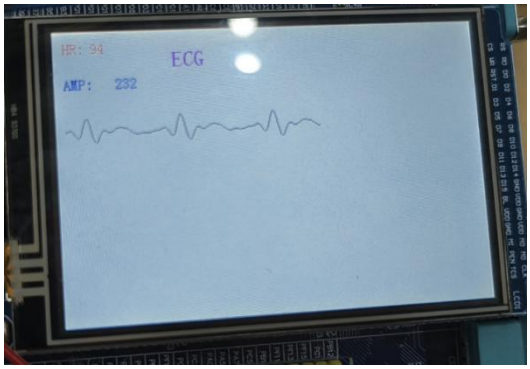

图 滤波后的心电信号

可以看到，采用级联的一阶 IIR 高通和高阶 FIR 低通组合滤波，效果显著，可以分别去除基线漂移和高频干扰，使得心电信号更平滑清晰。

2.3. 心跳检测算法详细说明

在预处理后的信号上采用峰值检测算法识别 R 波。这里采用经典的 Pan-Tompkins 算法步骤：

- 差分操作：计算当前采样与前一个采样的差值，得到信号的一阶差分（近似微分）。该步骤突出心电信号急剧的斜率变化（QRS 波群）。Pan-Tompkins 算法通常使用 5 点差分滤波器，其差分方程为：

$$y[n] = \frac{1}{8}(x[n-2] + 2x[n-1] + 2x[n+1] + x[n+2] + 2x[n])$$

- 取绝对值或平方：对差分结果取绝对值或平方运算，使所有信号值为正并进一步增强幅度大的 QRS 成分。Squaring 操作尤其有效，可将 R 波峰值放大，同时抑制低幅值噪声。
- 滑动窗口积分：设置宽度约 150ms 的滑动窗口，对平方后的信号进行积分运算，得到平滑的包络曲线。积分输出信号在 QRS 持续期内保持较高值，在

非 QRS 区间则较低。窗口积分长度的选择一般参考心电的典型 QRS 宽度。

- 自适应阈值判断：分别对滤波输出和积分输出计算阈值。典型做法是：在前 2 秒内统计所有峰值，令初始信号阈值 ST_1 为最大峰值的$\frac{1}{3}$，噪声阈值 NT_1 为峰值平均值的一半；滤波后信号 ST_2 和 NT_2 类似。每次检测到峰值后，根据其是否超过阈值来判断是否为 R 波，同时动态更新阈值和噪声水平。只有当经过滤波和积分后的对应峰值同时超过各自阈值时，才能确认该峰为真正的 R 波。此方法能够自适应背景噪声强度，提高检测准确率。
- 不应期设计：为避免一个 QRS 被多次检测，在识别到 R 波后启动不应期（200ms），在此期间忽略其他峰值。Pan-Tompkins 算法中还包括回溯机制：若在预设 RR 间隔时长内未检测到 R 波，则认为可能漏检，回溯查找最高峰值作为 R 波候选，并更新阈值。上述流程能有效检测心电信号中的 R 波峰值，为心率计算提供准确事件。

2.4. 心率计算方式与 DWT 计时器的作用

心率的计算基于相邻两个 R 波之间的时间间隔 RR 间隔。具体做法是在检测到两个连续 R 波时，测量它们之间的采样点差（或时间差），然后通过 HR = 60 / RR 间隔(秒)计算每分钟心跳数。为了获得高精度的时间测量，我们利用了 Cortex-M4 内核中的 DWT（Data Watchpoint and Trace）单元。DWT 包含一个 32 位的 CYCCNT 寄存器，它以 CPU 时钟为计数基础，可提供纳秒级别精度的计时。系统初始化时使能 DWT 并清零 CYCCNT，之后每经过一个 CPU 时钟周期该寄存器加 1。在检测到 R 波时，可读取此寄存器并换算为时间（例如除以时钟频率获得微秒级时间），从而精确计算 R 波间隔。借助 DWT 计时器记录的高分辨率时间戳，心率的计算结果更加准确且实时（避免了传统定时器较低精度的问题）。如文献所述，DWT 单元提供了方便的微秒/纳秒级时基，非常适合嵌入式实时测量和性能分析。

2.5. FFT 频谱分析实现方式

为了分析心电信号的频率成分，我们在 STM32 上采用 CMSIS-DSP 库提供的 FFT 函数（如 arm_rfft_fast_f32）进行快速傅里叶变换。CMSIS-DSP 库在 STM32F4 等带 FPU 的 MCU 上运行效率很高，可实时对片上采集到的心电数据进行谱分析。通过 FFT 将时域信号转换到频域后，我们计算功率谱，即各频率分量的能量分布。频谱图可以揭示 ECG 信号中的关键频率特征：例如，心跳节律通常对应 1–2Hz 左右的低频成分，而 50Hz 工频噪声在频谱上也会显现为尖峰。文献指出，FFT 及功率谱分析能够帮助识别噪声频段和与心脏自主节律相关的成分。在实际应用中，我们可通过分析谱图来评估噪声滤除效果或诊断心律失常等问题。图形上，我们利用 FFT 输出的复数幅度谱绘制频谱图，直观地观察

不同频率下的信号能量分布。例如，采样窗内心率对应频率的峰值反映了心跳周期，较高频分量则代表肌电或其他噪声,如下图所示，同时滤波前后频谱图的对比也进一步表明前文设计的滤波器非常有效。

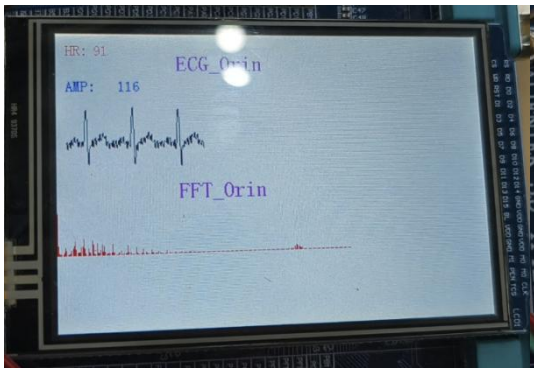

图 滤波前信号频谱图

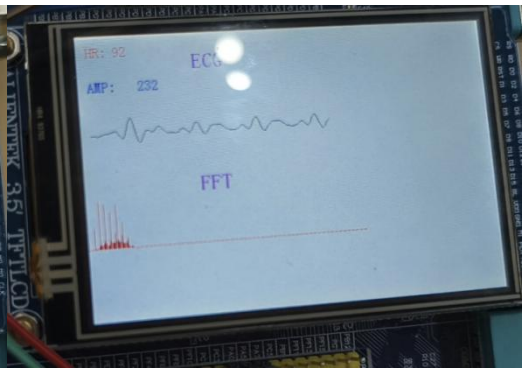

图 滤波后信号频谱图

2.6. 数据发送与上位机处理

为了进一步对数据进行处理，我们将处理前的心电数据通过串口发送给 PC，在 PC 端同步实现了一套完整的 ECG 数据处理系统。处理前的 ECG 数据通过串口发送给上位机。为此，我们设计了简单的数据帧协议：每帧由固定的起始标志（0XA5）开头，随后将电平转换后的浮点心电数据拆分为四个字节进行发送。在数据到达上位机之后，就进入了 PC 端的数据处理部分。由于前文对于整个 PC 系统的具体组成部分，运行逻辑以及代码构成都进行了详细的描述，这里就不进行过多的赘述。

2.7. LCD 屏幕波形显示方式

LCD 屏幕用于实时绘制采集到的 ECG 波形。将 X 轴对应时间轴，Y 轴对应电压幅度。波形显示采用滚动更新的方式：每次获得新采样点后，将该点映射到屏幕最右侧像素，同时将之前的波形向左平移一个像素，形成连贯曲线。为了保证显示平滑，设置了定时器中断，在固定帧率（60Hz）下刷新波形。同时采用局部刷新技术：在一块屏幕缓冲区绘制完整的当前帧后，再一次性切换到 LCD 上显示，以避免闪烁。LCD 像素高度和宽度根据显示屏分辨率和设计比例预先设定，确保 QRS 峰完整呈现。软件中，波形更新逻辑与数据采集基本同步：新样本到来即绘制。显示刷新与 SPI 通信由独立 DMA 来控制，确保波形显示与信号采集保持同步且流畅。

2.8. 蜂鸣器心跳提示控制逻辑

蜂鸣器用于发出心跳提示音，提醒用户当前心跳情况。控制逻辑很简单：当检测到一个 R 波时，即可触发一次蜂鸣。通常使用单片机定时器产生一定频率（如 1–2kHz）的 PWM 信号驱动蜂鸣器发声，使其发出短促的“哔”声。具体实现中，在中断或主循环检测到 R 波标志后，启动蜂鸣 PWM 输出一定时间（例

如 50–100ms），然后关闭。在此过程中，还加入防抖和延迟措施，避免连续检测到多个极邻近峰值而发出连续声。在 R 波检测的 200ms 不应期内禁止重复响铃。整体上，蜂鸣器由 GPIO 脚控制，经简单的定时器或软件延时产生音调，其打开和关闭与心跳检测结果直接关联，方便直观地提示生理节律。

2.9. 项目测试结果与信号处理效果分析

在最终验收中，在下位机中可以准确显示处理前的时域频域图，处理后的时域频域图以及估算得到的心率。整体信号处理方案在实测中运行稳定可靠，波形与频谱符合预期。

3. 项目心得

本次实时心电信号监测与分析项目，作为一个完整的系统，在整体性能上表现出色。无论是在上位机的实时数据处理和直观的图形化显示，还是在下位机的高精度数据采集和高效预处理方面，都达到了预期效果。

通过这个项目，我对 ECG 信号处理算法的理解得到了显著深化。我学习了各种数字滤波器的应用及其在实时信号处理中的重要性，例如采用一阶 IIR 高通滤波器去除基线漂移 、使用 scipy.signal.iirnotch 设计的陷波滤波器去除 50Hz 工频干扰 、以及 FIR 低通滤波器去除高频噪声 。更重要的是，我掌握了如何在实际应用中设计和实现一个高并发、高实时性的多线程系统。上位机采用独立的多线程架构，将串口数据读取、数据处理和 GUI 显示划分到独立的线程中，这有效解决了实时系统中常见的“数据 I/O 阻塞 UI”问题 ，确保了用户界面的流畅性和数据的实时性。

此外，项目中的跨平台协作经验让我深刻体会到清晰的接口定义和协议设计在系统集成中的重要性。下位机通过串口向上位机发送处理前的 ECG 数据，为此，我们设计了简单的数据帧协议：每帧由固定的起始标志（0XA5）开头，随后将电平转换后的浮点心电数据拆分为四个字节进行发送。这种规范化的通信方式是上位机能够稳定接收和处理下位机数据的基石。

整个项目的成功实施，无疑离不开我和另一位同学的紧密配合。感谢另一位同学的努力。

4. 附录

在整个报告的最后，我们附上在本次项目当中比较重要的代码。

4.1. 上位机代码

4.1.1. 串口读取线程

```
Python
串口读取线程
def serial_reader_thread(port, baudRate, dataLength, header):
    """
    串口读取线程：负责从串口读取原始 ECG 数据和时间戳，放入 raw_data_queue。

    参数:
        port: 串口号 (如 'COM3' 或 '/dev/ttyUSB0')
        baudRate: 波特率 (如 115200)
        dataLength: 数据包长度 (字节数，float 类型为 4 字节)
        header: 数据包起始头标识 (1 字节)
    """
    ser = None  # 串口对象初始化
    try:
        # 创建串口连接对象
        ser = serial.Serial(
            port=port,
            baudrate=baudRate,
            bytesize=dataBits,          # 数据位 (外部变量，通常为 8)
            # 根据 stopBits 设置停止位 (1 或 2)
            stopbits=serial.STOPBITS_ONE if stopBits == 1 else
serial.STOPBITS_TWO,
            timeout=0.05                # 读超时(秒)，防止线程阻塞
        )
        ser.flushInput()  # 清空输入缓冲区，丢弃旧数据
        print(f'串口读取线程已在 {port} 启动。')

        # 主循环：持续读取直到收到停止信号
        while not stop_event.is_set():  # stop_event 是外部全局停止事
件
            # 检查缓冲区是否有足够数据（包头+数据包）
            if ser.in_waiting >= (dataLength + 1):
```

```
                # 读取 1 字节检测包头
                current_byte_bytes = ser.read(1)

                # 验证包头匹配
                if current_byte_bytes[0] == header:
                    try:
                        # 读取完整数据包
                        bytes_read = ser.read(dataLength)

                        if len(bytes_read) == dataLength:
                            # 将字节数据解析为小端 float
                            combined_data_float = struct.unpack('<f',
bytes_read)[0]

                            # 将(ECG 值, 当前时间戳)放入队列
                            raw_data_queue.put((combined_data_float,
time.time()))
                        else:
                            # 数据长度不符，丢弃并清空缓冲区
                            print("警告: 串口读取到不完整数据包，正在刷新
输入。")
                            ser.flushInput()
                    except struct.error as e:
                        # 数据解析失败处理
                        print(f"串口读取错误: 解包浮点数失败 - {e}。正在
刷新输入。")
                        ser.flushInput()
                else:
                    # 非包头字节，静默丢弃继续搜索
                    pass
            else:
                # 缓冲区数据不足时短暂休眠，降低 CPU 占用
                time.sleep(0.001)

    except serial.SerialException as e:
        # 串口通信错误处理
        print(f'串口读取线程遇到串口错误: {e}')
```

```
    except Exception as e:
        # 其他未知错误处理
        print(f"串口读取线程发生未知错误: {e}")
    finally:
        # 确保关闭串口释放资源
        if ser and ser.is_open:
            ser.close()
        print('串口读取线程已停止。')
```

4.1.2. 数据处理线程

```
Python
数据处理线程
def data_processor_thread():
    """
    数据处理线程：从 raw_data_queue 获取数据，更新滑动窗口，完成所有信号处理与分析。
    主要功能：
      1. 从串口原始数据队列获取 ECG 数据点
      2. 维护滑动窗口缓冲区
      3. 对窗口数据进行滤波处理
      4. 执行 Pan-Tompkins 算法检测 R 波
      5. 计算实时心率
      6. 动态调整 R 波检测参数
      7. 执行 FFT 频谱分析
      8. 存储处理结果供 GUI 线程使用
    """
    # 声明全局变量
    global min_peak_distance_sec, min_peak_distance_samples, latest_heart_rate
    global latest_batch_data, main_data_buffer, main_timestamp_buffer, is_paused

    print('数据处理线程已启动。')
    # 首次处理前需要积累足够数据
    print(f"数据处理线程：等待数据填充至 {num_points_to_process_batch} 个点以启动处理...")
    initial_fill_done = False  # 标记初始填充是否完成
```

```
    # 主循环：持续处理直到收到停止信号
    while not stop_event.is_set():
        # --- 暂停/恢复控制 ---
        with pause_condition:  # 获取条件变量的锁
            # 如果暂停且程序未停止，则等待
            while is_paused and not stop_event.is_set():
                pause_condition.wait()  # 释放锁并等待通知

        # 唤醒后再次检查停止事件，防止等待期间收到停止信号
        if stop_event.is_set():
            break

        try:
            # 尝试从队列获取新数据点，如果队列为空则短暂等待
            raw_value, timestamp = raw_data_queue.get(timeout=0.01)

            # --- 更新主滑动窗口缓冲区 ---
            main_data_buffer.append(raw_value)       # 添加原始 ECG 值
            main_timestamp_buffer.append(timestamp)  # 添加对应时间戳

            # 当主滑动窗口完全填满时，开始完整处理流程
            # 之后每个新点替换最旧点，整个窗口会被重新处理
            if len(main_data_buffer) == num_points_to_process_batch:
                if not initial_fill_done:
                    print("数据处理线程：滑动窗口已首次填满，开始处理数据。
")
                    initial_fill_done = True

                # 将 deque 转换为 numpy 数组以便批量处理
                segment_raw = np.array(main_data_buffer)
                segment_timestamps = np.array(main_timestamp_buffer)

                # --- 信号预处理 ---
                # 1. 去除直流偏移
                ecg_dc_removed = signal.lfilter(b_dc, a_dc,
segment_raw)
```

```
                # 2. 50Hz 工频陷波滤波
                ecg_notch_50hz_removed = signal.lfilter(b_notch_50hz,
a_notch_50hz, ecg_dc_removed)

                # 3. 低通滤波（用于显示的最终 ECG 信号）
                ecg_filtered_for_display = signal.lfilter(b_lp_firls,
1, ecg_notch_50hz_removed)

                # --- R 波检测 ---
                # 获取当前 R 波检测参数（加锁访问全局变量）
                with r_peak_params_lock:
                    current_min_peak_distance_sec =
min_peak_distance_sec

                # 使用 Pan-Tompkins 算法检测 R 波位置
                locs_r_peaks = r_peak_detector_pan_tompkins(
                    ecg_notch_50hz_removed,
                    Fs,
                    min_distance_sec=current_min_peak_distance_sec
                )

                # 获取最新 R 波的实际幅值（来自最终滤波数据，用于显示）
                latest_r_peak_value_actual = np.nan
                if len(locs_r_peaks) > 0:
                    last_r_peak_idx = locs_r_peaks[-1]
                    # 确保索引有效
                    if 0 <= last_r_peak_idx <
len(ecg_filtered_for_display):
                        latest_r_peak_value_actual =
ecg_filtered_for_display[last_r_peak_idx]

                # --- 心率计算 ---
                heart_rate = np.nan  # 默认设为 NaN
                if len(locs_r_peaks) >= 2:  # 至少需要两个 R 波
                    # 使用 R 波对应的时间戳计算 RR 间期
                    time_r2 = segment_timestamps[locs_r_peaks[-1]]
                    time_r1 = segment_timestamps[locs_r_peaks[-2]]
```

```
                    rr_interval_sec = time_r2 - time_r1

                    if rr_interval_sec > 0:  # 避免除零错误
                        heart_rate = 60 / rr_interval_sec  # 转换为 bpm
                        latest_heart_rate = heart_rate  # 更新全局心率
变量

                # --- 动态调整 R 波检测最小间隔 ---
                # 根据心率动态调整 R 波最小间隔，防止误检
                if not np.isnan(heart_rate):
                    new_min_peak_distance_sec_candidate =
min_peak_distance_sec

                    # 心率过快时减小最小间隔
                    if heart_rate > 150:
                        new_min_peak_distance_sec_candidate = 60 /
(heart_rate * 1.2)
                        # 设置下限
                        if new_min_peak_distance_sec_candidate < 0.2:
                            new_min_peak_distance_sec_candidate = 0.2
                    # 心率过慢时增大最小间隔
                    elif heart_rate < 50:
                        new_min_peak_distance_sec_candidate = 60 /
(heart_rate * 0.8)
                        # 设置上限
                        if new_min_peak_distance_sec_candidate > 1.0:
                            new_min_peak_distance_sec_candidate = 1.0
                    # 正常心率使用默认值
                    else:
                        new_min_peak_distance_sec_candidate = 0.3

                    # 更新全局参数（加锁）
                    with r_peak_params_lock:
                        # 仅在变化超过阈值时更新
                        if abs(new_min_peak_distance_sec_candidate -
min_peak_distance_sec) > 0.01:
                            min_peak_distance_sec =
```

```
new_min_peak_distance_sec_candidate
                        min_peak_distance_samples =
round(min_peak_distance_sec * Fs)
                        # 调试信息（可选）
                        # print(f"心率: {heart_rate:.2f} 次/分钟,
动态调整 R 波检测最小间距为: {min_peak_distance_sec:.2f}秒")

            # --- FFT 频谱分析 ---
            N_fft = len(segment_raw)  # FFT 点数等于滑动窗口大小

            # 辅助函数：计算信号的 FFT 单边功率谱
            def calculate_fft_power(data_segment):
                Y = fft(data_segment)  # 快速傅里叶变换
                # 计算单边谱并归一化
                P1 = 2.0 / N_fft * np.abs(Y[0:N_fft // 2 + 1])
                f = fftfreq(N_fft, 1 / Fs)[:N_fft // 2 + 1]  # 频
率轴
                return P1, f

            # 计算各处理阶段的频谱
            P1_raw, f_raw = calculate_fft_power(segment_raw)  # 原
始信号
            P1_dc_removed, f_dc_removed =
calculate_fft_power(ecg_dc_removed)  # 去直流后
            P1_notch_50hz_removed, f_notch_50hz_removed =
calculate_fft_power(ecg_notch_50hz_removed)  # 陷波后
            P1_lp, f_lp =
calculate_fft_power(ecg_filtered_for_display)  # 最终滤波后

            # --- 存储处理结果供 GUI 线程使用 ---
            with batch_data_lock:  # 加锁写入共享数据
                latest_batch_data = {
                    'ecg_raw_for_plot': segment_raw,  # 原始 ECG 用
于绘图
                    'ecg_filtered_for_plot':
ecg_filtered_for_display,  # 滤波后 ECG 用于时域图
                    'r_peaks_locs': locs_r_peaks,  # R 波位置索引
```

```
                        'heart_rate': heart_rate,  # 当前心率
                        'freq_spectrum_lp': P1_lp,  # 最终滤波信号频谱
                        'freq_axis_lp': f_lp,  # 频谱频率轴
                        'freq_spectrum_dc_removed': P1_dc_removed,  #
去直流后频谱
                        'freq_axis_dc_removed': f_dc_removed,  # 频率
轴
                        'freq_spectrum_notch_50hz_removed':
P1_notch_50hz_removed,  # 陷波后频谱
                        'freq_axis_notch_50hz_removed':
f_notch_50hz_removed,  # 频率轴
                        'freq_spectrum_raw': P1_raw,  # 原始信号频谱
                        'freq_axis_raw': f_raw,  # 频率轴
                        'latest_r_peak_value_actual':
latest_r_peak_value_actual  # 最新 R 波幅值
                    }

                # 将最新处理点添加到实时显示队列

processed_data_queue.put(ecg_filtered_for_display[-1])

        except queue.Empty:
            # 队列为空时继续循环
            pass
        except Exception as e:
            # 捕获并打印其他异常
            print(f"数据处理线程错误: {e}")

    # 线程结束处理
    print('数据处理线程已停止。')
```

4.1.3. GUI 显示线程

```
Python
GUI 显示线程
def update_plots_gui():
    """
    定期更新所有图表（心电图机显示、时域图、频域图）。
```

```
    主要功能：
      1. 更新实时心电图机滚动显示
      2. 更新批处理时域图（含 R 波检测）
      3. 更新频域图（各处理阶段的频谱）
      4. 更新心率显示
      5. 更新 R 波峰值信息
    """
    # 声明全局变量（图表对象、数据缓冲区等）
    global fig_main, fig_freq_domain, ax_time_domain, ax_freq_domain,
ax_freq_domain_dc_removed, ax_freq_domain_notch_50hz_removed,
ax_freq_domain_raw, ax_ecg_machine, ax_raw_ecg
    global line_time_domain, scatter_r_peaks, line_freq_domain,
line_freq_domain_dc_removed, line_freq_domain_notch_50hz_removed,
line_freq_domain_raw, line_ecg_machine, line_raw_ecg,
r_peak_value_text
    global ecg_scroll_buffer, last_batch_update_time,
latest_heart_rate
    global latest_batch_data

    # --- 1. 更新实时心电图机显示 ---
    new_processed_points = []
    # 从处理队列中获取所有新点
    while not processed_data_queue.empty():
        try:

new_processed_points.append(processed_data_queue.get_nowait())
        except queue.Empty:
            break

    if new_processed_points:
        # 将新点添加到滚动缓冲区
        ecg_scroll_buffer.extend(new_processed_points)
        # 更新心电图机显示
        line_ecg_machine.set_ydata(ecg_scroll_buffer)

        # 每隔 4 秒计算并更新一次心率显示
        if not hasattr(update_plots_gui, "last_hr_update_time"):
```

```
            # 初始化静态变量（首次调用时）
            update_plots_gui.last_hr_update_time = 0  # 记录上次更新时
间
            update_plots_gui.last_hr_value = None     # 记录上次心率值

        now = time.time()
        if now - update_plots_gui.last_hr_update_time > 4:
            # 重新计算心率
            hr_value = np.nan
            with batch_data_lock:  # 加锁访问共享数据
                if latest_batch_data and
len(latest_batch_data['r_peaks_locs']) >= 2:
                    r_peaks = latest_batch_data['r_peaks_locs']
                    segment_timestamps =
np.array(main_timestamp_buffer)
                    # 获取最后两个 R 波的时间戳
                    time_r2 = segment_timestamps[r_peaks[-1]]
                    time_r1 = segment_timestamps[r_peaks[-2]]
                    rr_interval_sec = time_r2 - time_r1
                    if rr_interval_sec > 0:
                        hr_value = 60 / rr_interval_sec  # 计算心率
            # 更新静态变量
            update_plots_gui.last_hr_value = hr_value
            update_plots_gui.last_hr_update_time = now

        # 刷新心电图机显示
        fig_main.canvas.draw_idle()

    # --- 2. 更新批处理图（时域图、频域图） ---
    current_time = time.time()
    batch_update_interval_sec = 0.05  # 批处理更新间隔（20Hz）

    # 检查是否需要更新批处理图
    if current_time - last_batch_update_time >
batch_update_interval_sec:
        with batch_data_lock:  # 加锁访问批处理数据
            if latest_batch_data:
```

```
                # --- 更新原始心电信号图 ---
                ax_raw_ecg.set_xlim(0, num_points_to_process_batch -
1)  # 设置 X 轴范围
                # 计算 Y 轴范围（带 20%边距）
                min_y_raw = np.min(main_data_buffer)
                max_y_raw = np.max(main_data_buffer)
                ax_raw_ecg.set_ylim(
                    min_y_raw - 0.2 * np.abs(min_y_raw) if min_y_raw <
0 else -0.2,
                    max_y_raw + 0.2 * np.abs(max_y_raw) if max_y_raw >
0 else 0.2
                )
                # 更新原始信号曲线

line_raw_ecg.set_data(np.arange(num_points_to_process_batch),
main_data_buffer)

                # --- 更新时域图 ---
                # 设置 X 轴范围（0 到数据点数量-1）
                ax_time_domain.set_xlim(0,
num_points_to_process_batch - 1)

                # 计算滤波后信号的 Y 轴范围
                min_y_td =
np.min(latest_batch_data['ecg_filtered_for_plot'])
                max_y_td =
np.max(latest_batch_data['ecg_filtered_for_plot'])
                ax_time_domain.set_ylim(
                    min_y_td - 0.2 * np.abs(min_y_td) if min_y_td < 0
else -0.2,
                    max_y_td + 0.2 * np.abs(max_y_td) if max_y_td > 0
else 0.2
                )
                # 更新时域曲线
                line_time_domain.set_data(
                    np.arange(num_points_to_process_batch),
                    latest_batch_data['ecg_filtered_for_plot']
```

```
                )

                # 更新标题（显示当前心率）
                if not np.isnan(latest_batch_data['heart_rate']):
                    ax_time_domain.set_title(f"时域波形及 R 波检测 (心
率: {latest_batch_data['heart_rate']:.2f} 次/分钟)")
                else:
                    ax_time_domain.set_title("时域波形及 R 波检测 (心
率: -- 次/分钟)")

                # 更新 R 峰值文本信息
                if not
np.isnan(latest_batch_data['ecg_filtered_for_plot']).all():
                    # 1. 找到滤波后数据的最大值及其索引（即 R 波位置）
                    max_filtered =
np.max(latest_batch_data['ecg_filtered_for_plot'])
                    max_filtered_idx =
np.argmax(latest_batch_data['ecg_filtered_for_plot'])
                    # 2. 在相同位置的原始信号中查找值
                    if 0 <= max_filtered_idx <
len(latest_batch_data['ecg_raw_for_plot']):
                        raw_value_at_max =
latest_batch_data['ecg_raw_for_plot'][max_filtered_idx]
                        r_peak_value_text.set_text(
                            f"降噪后 R 波峰值: {max_filtered:.2f}\n 原始
输入: {raw_value_at_max:.2f}"
                        )
                    else:
                        r_peak_value_text.set_text(f"降噪后峰值:
{max_filtered:.2f}\n 原始输入: --")
                else:
                    r_peak_value_text.set_text("降噪后 R 波峰值: --\n
原始输入: --")

                # --- 更新频域图 ---
                # 辅助函数：将幅度转换为分贝(dB)
                def to_db(arr):
```

```
                    # 避免 log10(0)或负值
                    return 20 * np.log10(np.where(arr > 1e-10, arr,
1e-10))

                # 计算统一的 Y 轴范围（基于原始信号频谱）
                db_raw_for_lim =
to_db(latest_batch_data['freq_spectrum_raw'])
                if np.any(db_raw_for_lim > -np.inf):
                    min_y_freq_db = np.min(db_raw_for_lim) - 5  # 下
限减 5dB
                    max_y_freq_db = np.max(db_raw_for_lim) + 5  # 上
限加 5dB
                else:
                    min_y_freq_db = -80  # 默认下限
                    max_y_freq_db = 0    # 默认上限

                # 原始信号频谱
                db_raw =
to_db(latest_batch_data['freq_spectrum_raw'])

line_freq_domain_raw.set_data(latest_batch_data['freq_axis_raw'],
db_raw)
                ax_freq_domain_raw.set_ylim(min_y_freq_db,
max_y_freq_db)
                ax_freq_domain_raw.relim()  # 重新计算限制
                ax_freq_domain_raw.autoscale_view()  # 自动缩放视图

                # DC 移除后频谱
                db_dc_removed =
to_db(latest_batch_data['freq_spectrum_dc_removed'])

line_freq_domain_dc_removed.set_data(latest_batch_data['freq_axis_
dc_removed'], db_dc_removed)
                ax_freq_domain_dc_removed.set_ylim(min_y_freq_db,
max_y_freq_db)
                ax_freq_domain_dc_removed.relim()
                ax_freq_domain_dc_removed.autoscale_view()
```

```
                # 50Hz 陷波后频谱
                db_notch_50hz_removed =
to_db(latest_batch_data['freq_spectrum_notch_50hz_removed'])

line_freq_domain_notch_50hz_removed.set_data(latest_batch_data['fr
eq_axis_notch_50hz_removed'], db_notch_50hz_removed)

ax_freq_domain_notch_50hz_removed.set_ylim(min_y_freq_db,
max_y_freq_db)
                ax_freq_domain_notch_50hz_removed.relim()
                ax_freq_domain_notch_50hz_removed.autoscale_view()

                # 低通滤波后频谱
                db_lp = to_db(latest_batch_data['freq_spectrum_lp'])

line_freq_domain.set_data(latest_batch_data['freq_axis_lp'], db_lp)
                ax_freq_domain.set_ylim(min_y_freq_db,
max_y_freq_db)
                ax_freq_domain.relim()
                ax_freq_domain.autoscale_view()

                # 刷新频域图
                fig_freq_domain.canvas.draw_idle()

                # 更新最后批处理时间并清空批处理数据
                last_batch_update_time = current_time
                latest_batch_data = None

    # 如果不是停止状态，安排下一次更新（10ms 后）
    if not stop_event.is_set():
        root.after(10, update_plots_gui)
```

4.1.4. Pan-Tompkins 算法处理函数

```
Python
Pan-Tompkins 算法处理函数
def r_peak_detector_pan_tompkins(ecg_signal, Fs,
```

```
min_distance_sec=0.3):
    """
    简化版 Pan-Tompkins 算法实现 R 波检测。

    参数:
        ecg_signal (np.array): 原始或预处理 ECG 信号
        Fs (float): 采样频率 (Hz)
        min_distance_sec (float): R 波最小间隔（秒），用于抑制误报

    返回:
        np.array: 检测到的 R 波索引
    """

    # 1. 带通滤波 (5-15Hz)
    # 目的：抑制基线漂移和高频噪声，突出 QRS 复合波
    # b_bandpass_pt, a_bandpass_pt 是预定义的滤波器系数
    filtered_bandpass = signal.lfilter(b_bandpass_pt, a_bandpass_pt,
ecg_signal)

    # 2. 微分（一阶差分）
    # 目的：增强 QRS 复合波的陡峭斜率，抑制 P 波和 T 波的缓慢变化
    # b_diff_pt, a_diff_pt 是预定义的微分滤波器系数
    differentiated_signal = signal.lfilter(b_diff_pt, a_diff_pt,
filtered_bandpass)

    # 3. 平方运算
    # 目的：使所有样本值为正，进一步突出高幅值的 QRS 复合波
    #       同时抑制低幅值噪声和 T 波
    squared_signal = differentiated_signal**2

    # 4. 积分（移动平均）
    # 目的：获取 QRS 复合波持续时间信息，平滑信号
    #       使 QRS 复合波对应的峰值更宽、更明显
    # b_integrator_pt, a_integrator_pt 是预定义的积分滤波器系数
    integrated_signal = signal.lfilter(b_integrator_pt,
a_integrator_pt, squared_signal)
```

```
    # 5. 峰值检测和自适应阈值处理
    # 原始 Pan-Tompkins 算法包含复杂的自适应阈值和回溯机制
    # 此处简化：使用 scipy 的 find_peaks 函数进行峰值检测

    # 归一化积分信号，使高度参数更具通用性
    # 确保信号不全为零（避免除以零错误）
    if np.max(integrated_signal) != 0:
        normalized_integrated = integrated_signal /
np.max(integrated_signal)
    else:
        normalized_integrated = np.zeros_like(integrated_signal)  #
全零则设为 0

    # 基于心率动态调整最小峰值距离（样本数）
    min_peak_distance_samples_pt = int(min_distance_sec * Fs)

    # 在积分信号中使用 find_peaks 检测峰值
    # height: 相对高度阈值，过滤小峰值
    # distance: 确保峰值间有足够间隔，避免重复检测
    peaks_in_integrated, _ = signal.find_peaks(
        normalized_integrated,
        height=min_peak_height_integrated,  # 预定义的最小峰值高度阈
值
        distance=min_peak_distance_samples_pt
    )

    # 回溯：从积分信号的峰值位置定位原始 ECG 信号的精确 R 峰位置
    # R 峰通常位于积分信号峰值附近，是原始 ECG 信号中的最大值点
    r_peaks_locations = []

    # 定义搜索窗口（积分峰值前后各 100ms）
    search_window_around_peak_ms = 100  # 毫秒
    search_window_samples = int(search_window_around_peak_ms / 1000
* Fs)

    # 遍历积分信号中找到的所有峰值
    for peak_idx_integrated in peaks_in_integrated:
```

```
        # 计算原始 ECG 信号中的搜索窗口边界
        start_idx = max(0, peak_idx_integrated -
search_window_samples)
        end_idx = min(len(ecg_signal), peak_idx_integrated +
search_window_samples)

        # 确保有效搜索范围
        if start_idx < end_idx:
            # 提取原始 ECG 信号的局部片段
            local_segment_ecg = ecg_signal[start_idx:end_idx]

            if len(local_segment_ecg) > 0:
                # 在局部片段中寻找最大值位置作为 R 峰
                local_peak_offset = np.argmax(local_segment_ecg)

                # 计算原始信号中的实际索引位置
                actual_r_peak_idx = start_idx + local_peak_offset
                r_peaks_locations.append(actual_r_peak_idx)

    return np.array(r_peaks_locations)
```

4.1.5. 主函数进程

```
Python
主函数进程
# --- 主程序执行区 ---
if __name__ == "__main__":
    # 创建串口读取线程
    serial_thread = threading.Thread(
        target=serial_reader_thread,  # 线程目标函数
        args=(port, baudRate, dataLength, header),  # 传递给函数的参
数
        name="SerialReaderThread"  # 线程名称
    )

    # 创建数据处理线程
    processor_thread = threading.Thread(
        target=data_processor_thread,  # 线程目标函数
```

```
        name="DataProcessorThread"  # 线程名称
    )

    # 启动线程
    serial_thread.start()  # 启动串口读取线程
    processor_thread.start()  # 启动数据处理线程

    try:
        # 启动 GUI，进入主事件循环（会阻塞在此处）
        setup_gui()
    except Exception as e:
        # 捕获并打印未处理的异常
        print(f"主程序发生未处理错误: {e}")
    finally:
        # 无论是否发生异常，都执行清理操作
        # 设置停止事件，通知所有线程退出
        stop_event.set()

        # 在关闭前唤醒所有等待的线程，确保它们能响应停止事件
        with pause_condition:
            pause_condition.notify_all()  # 通知所有等待在条件变量上的
线程

        # 等待线程终止，设置超时防止线程挂起
        print("等待串口读取线程终止...")
        serial_thread.join(timeout=2)  # 最多等待 2 秒

        print("等待数据处理线程终止...")
        processor_thread.join(timeout=2)  # 最多等待 2 秒

        # 检查线程是否仍在运行
        if serial_thread.is_alive():
            print("警告: 串口读取线程未在规定时间内终止。")
        if processor_thread.is_alive():
            print("警告: 数据处理线程未在规定时间内终止。")
```

```
        print("程序已干净终止。")
```

4.2. 下位机代码

4.2.1. 心电数据转换程序

```
C
ADS1296 数据解析
static void ECG_process(void)
{
    // 解析状态字节
    ads1292->status = ads1292->raw_data[0];

    // 检测导联状态
    ads1292->lead_left  = !(ads1292->status & 0x04); // 左导联状态
    ads1292->lead_right = !(ads1292->status & 0x02); // 右导联状态
    ads1292->ecg_active = ads1292->lead_left && ads1292->lead_right;

    // 解析呼吸阻抗和 ECG 数据
    int ecg_data = (ads1292->raw_data[3] << 16) |
(ads1292->raw_data[4] << 8) | ads1292->raw_data[5];
    int respirat = (ads1292->raw_data[6] << 16) |
(ads1292->raw_data[7] << 8) | ads1292->raw_data[8];
    // 转换 24 位有符号数为 32 位
    // if (respirat & 0x800000) respirat |= 0xFF000000; // 符号扩展
    // if (ecg_data & 0x800000) ecg_data |= 0xFF000000; // 符号扩展

    // 更新解析后的数据
    respirat                   = get_volt(respirat);
    ecg_data                   = get_volt(ecg_data);
    ads1292->ecg_data           = ecg_data * 3.3f / (32767.0f*5);
    ads1292->respirat_impedance = respirat * 3.3f / (32767.0f*5);
    ecg_watch                 = ads1292->ecg_data;
    ecg_test        = ads1292->ecg_data;
    squer_wave_test = ads1292->respirat_impedance;
    Transe_ECGData();
}
```

```
C++
数据补码
static int32_t get_volt(uint32_t data)
{
    int32_t volt = 0;
    if (data & 0x800000) {
        volt = data | 0xFF000000;
    } else {
        volt = (data& 0x00FFFFFF);
    }
    return volt;
}
```

4.2.2. IIR 滤波器实现

```
C++
// IIR 滤波器相关参数
static float a           = 0.991;
static float last_input    = 0;
static float last_output   = 0;
static float cur_input = 0;
static float cur_output    = 0;

float IIR_Filter(float rawData)
{
    float IIR_Result;
    cur_input  = rawData;
    cur_output = cur_input - last_input + a * last_output;
    IIR_Result = (float)cur_output;
    // printf("%d\n",IIR_Result);
    last_output = cur_output;
    last_input  = cur_input;
    return IIR_Result;
}
```

4.2.3. FIR 滤波器实现

```
C++
```

```
FIR 滤波器系数
#define hLength 161
// FIR 滤波器参数
static double coefficient = 4.0;
static Queue FIRQueue;
static float FIRResult;
static float FIRA[hLength];
static double B[161] = {
    -1.104587824635e-06, -3.30765288084e-07, 6.469557905197e-07,
2.354792370333e-06,
    4.613259754644e-06, 6.877479993928e-06, 8.237049198786e-06,
7.558936626691e-06,
    3.788971148408e-06, -3.623848847902e-06, -1.427538238554e-05,
-2.647155435505e-05,
    -3.717488350579e-05, -4.236786223392e-05, -3.788474498342e-05,
-2.063216063947e-05,
    1.003324496041e-05, 5.113408663599e-05, 9.553680832838e-05,
0.0001324352646621,
    0.0001490628825309, 0.0001335519048098, 7.850129689465e-05,
-1.552301519343e-05,
    -0.000137547975341, -0.0002658042593173, -0.0003701010370572,
-0.0004171261821494,
    -0.0003780894912395, -0.0002373438145375, -1.701986658015e-18,
0.0003036972242474,
    0.000619188232845, 0.0008747139088742, 0.0009944873077503,
0.000916089676601,
    0.0006087713144456, 8.835700253944e-05, -0.0005756611688794,
-0.001264303369241,
    -0.00182600060318, -0.002104876596178, -0.001976059132852,
-0.001381105279486,
    -0.0003552282206677, 0.0009616648412149, 0.00233583791841,
0.003474169866146,
    0.004078695334442, 0.003912505763598, 0.002864124412349,
0.000995570008819,
    -0.001439527443595, -0.004016978287615, -0.006204281185006,
-0.007457327047886,
    -0.00733619066631, -0.005618081778844, -0.002382762583926,
```

```
0.001952112154742,
    0.006661024610152, 0.01080982348891, 0.01341421446485,
0.01363410556943,
    0.01097007287533, 0.005423497886492, -0.002415610166809,
-0.01137922416383,
    -0.01984044191607, -0.02593171901878, -0.02783322176153,
-0.02408663464818,
    -0.01388122282339, 0.002740281371631, 0.02479787236787,
0.05043558413653,
    0.07712685401238, 0.1019977732739, 0.1222205837105,
0.13541600842,
    0.1399996906768, 0.13541600842, 0.1222205837105,
0.1019977732739,
    0.07712685401238, 0.05043558413653, 0.02479787236787,
0.002740281371631,
    -0.01388122282339, -0.02408663464818, -0.02783322176153,
-0.02593171901878,
    -0.01984044191607, -0.01137922416383, -0.002415610166809,
0.005423497886492,
    0.01097007287533, 0.01363410556943, 0.01341421446485,
0.01080982348891,
    0.006661024610152, 0.001952112154742, -0.002382762583926,
-0.005618081778844,
    -0.00733619066631, -0.007457327047886, -0.006204281185006,
-0.004016978287615,
    -0.001439527443595, 0.000995570008819, 0.002864124412349,
0.003912505763598,
    0.004078695334442, 0.003474169866146, 0.00233583791841,
0.0009616648412149,
    -0.0003552282206677, -0.001381105279486, -0.001976059132852,
-0.002104876596178,
    -0.00182600060318, -0.001264303369241, -0.0005756611688794,
8.835700253944e-05,
    0.0006087713144456, 0.000916089676601, 0.0009944873077503,
0.0008747139088742,
    0.000619188232845, 0.0003036972242474, -1.701986658015e-18,
-0.0002373438145375,
```

```
    -0.0003780894912395, -0.0004171261821494, -0.0003701010370572,
-0.0002658042593173,
    -0.000137547975341, -1.552301519343e-05, 7.850129689465e-05,
0.0001335519048098,
    0.0001490628825309, 0.0001324352646621, 9.553680832838e-05,
5.113408663599e-05,
    1.003324496041e-05, -2.063216063947e-05, -3.788474498342e-05,
-4.236786223392e-05,
    -3.717488350579e-05, -2.647155435505e-05, -1.427538238554e-05,
-3.623848847902e-06,
    3.788971148408e-06, 7.558936626691e-06, 8.237049198786e-06,
6.877479993928e-06,
    4.613259754644e-06, 2.354792370333e-06, 6.469557905197e-07,
-3.30765288084e-07,
    -1.104587824635e-06};
```

```
C
FIR 滤波器实现
double h[161];
// 初始化 FIR 滤波器参数 完成 FIR 队列创建工作
void FIRInit()
{
    queueInit(&FIRQueue, FIRA, hLength);
    for (int i = 1; i <= hLength; i++) {
        queuePush(&FIRQueue, 0);
    }
    DWT_Delay(2);
    for (int j = 0; j < hLength; j++) {
        h[j] = B[j] * 65536;
    }
}
float last_data = 0xffff;
float FIR_Filter(float in)
{
    // 过滤掉重复的
    if (in == last_data)
```

```
        return FIRResult;
    else
        last_data = in;
    queuePop(&FIRQueue);      // 先弹出一个
    queuePush(&FIRQueue, in); // 向其中加入新来的元素
    double sum = 0;
    int count  = hLength - 1; // 指向 h 末尾的位置
    // 随后遍历整个队列 计算出滤波之后的结果
    int pointer = FIRQueue.front;
    if (FIRQueue.count == 0) {
        return 0;
    } else if (FIRQueue.front <= FIRQueue.rear) {
        while (pointer <= FIRQueue.rear) {
            sum += (FIRQueue.array[pointer] * B[count]);
            count--;
            pointer++;
        }
    } else {
        for (; pointer <= FIRQueue.maxSize - 1; pointer++) {
            sum += (FIRQueue.array[pointer] * B[count]);
            count--;
        }
        for (pointer = 0; pointer <= FIRQueue.rear; pointer++) {
            sum += (FIRQueue.array[pointer] * B[count]);
            count--;
        }
    }
    FIRResult = sum * coefficient;
    return FIRResult;
}
```

4.2.4. FFT 实现

C
FFT 实现

```
// FFT 计算和绘制频谱图
static void FFT_Calculate(void)
{
```

```
    memcpy(fft_instance->input, fft_instance->input_pointer,
FFT_LENGTH * sizeof(float));

    for (uint16_t i = 0; i < FFT_LENGTH; i++) {
        fft_instance->fft_input[i * 2]     = ecg_buffer[i];
        fft_instance->fft_input[i * 2 + 1] = 0;
    }

    // FFT 计算
    arm_cfft_radix4_f32(&fft_instance->cfft_radix4_instance,
fft_instance->fft_input);
    arm_cmplx_mag_f32(fft_instance->fft_input,
fft_instance->output, FFT_LENGTH);
}
```

4.2.5. 数据发送协议实现

```
C
数据发送协议
static void Transe_ECGData()
{
    if(ads1292->is_collecting == 0){
        return;
    }
    uint8_t data[sizeof(ads1292->ecg_data) + 1];
    data[0] = 0xa5; // 数据包头
    memcpy(data + 1, &ecg_watch, sizeof(ads1292->ecg_data));

    // 发送数据到串口
    USARTSend(ECG_usart, data, sizeof(data),
USART_TRANSFER_BLOCKING);
}
```