



实 验 报 告

课程名称: 嵌入式系统设计

姓 名: 雷灿曦

学 号: 20233006006

年级专业: 2023 级电子信息

任课教师: 张永辉 徐博

分 数: _____

海南大学 信息与通信工程学院

温度计设计

一、实验目的

- (1) 掌握建立嵌入式系统软件工程项目的方法，编辑、编译和调试运行程序代码；
- (2) 学习使用嵌入式系统实验平台。学习 LPC1114 系列 I2C、SPI、ADC 等功能

二、实验平台

硬件平台：LPC1114 单片机

软件平台：Keil uVision4

三、实验内容

(1) 利用通用定时器实现定时 1s 中断，在定时器中断服务子程序中读取 LM75BD (I2C 总线) 当前温度值，并通过 SPI 接口将读到的温度值存储到 FLASH 存储器 XT25F02 中，在 PC 上利用串口调试助手向 LPC1114 的 UART 接口发送读取命令，从 FLASH 存储器中读出温度数据，并通过 UART 接口发送到 PC，利用串口调试助手接收温度数据（保留 3 位小数）。

(2) AD 转换器初始化为软件控制模式，开启 AD 中断，设置 AD7 为模拟输入引脚（AD7 引脚连接一个 10K 的 NTC 负温度系数热敏电阻，见 LPC1114 DevKit 电路图）。利用 32 位定时器 0 的 MAT0 定时 1s 匹配启动 AD 转换，在 AD 中断服务子程序中读取 AD 转换的值（AD7 通道），并通过 UART 接口将移位转换后的值（10 位，两个字节）发送到 PC，利用串口调试助手接收数据。

四、实验原理

4.1 所用器件引脚说明

本实验基于 LPC1100 系列微控制器，使用了以下关键引脚：

UART 通信引脚：

PI01_6: RXD (数据接收引脚)

PI01_7: TXD (数据发送引脚)

配置：通过 IOCON 寄存器设置为 UART 功能（功能选择位设置为 01）

ADC 模拟输入引脚：

PI01_11: AD7 (ADC 通道 7 输入)

配置: 通过 IOCON 寄存器设置为 ADC 功能, 模拟输入模式

SPI Flash 存储器引脚:

PI02_0: Flash_CS (Flash 片选信号)

配置: 通过 GPIO 方向寄存器设置为输出模式

PI02_1: SCK (SPI 时钟信号)

配置: 通过 IOCON 寄存器设置为 SCK 功能

PI02_2: MISO (主入从出数据线)

配置: 通过 IOCON 寄存器设置为 MISO 功能

PI02_3: MOSI (主出从入数据线)

配置: 通过 IOCON 寄存器设置为 MOSI 功能

I2C 温度传感器引脚:

PI00_4: SDA (I2C 数据线)

配置: 通过 IOCON 寄存器设置为 SDA 功能

PI00_5: SCL (I2C 时钟线)

配置: 通过 IOCON 寄存器设置为 SCL 功能

电源和参考电压:

VDD: 3.3V 电源输入

VSS: 电源地

VREFP: ADC 正参考电压 (3.3V)

VREFN: ADC 负参考电压 (0V)

4.2 相关寄存器配置说明

定时器相关寄存器:

TCR (定时器控制寄存器)

位 0 - 计数器使能: 为 1 时, 定时器/计数器和预分频计数器使能计数; 为 0 时, 计数器禁能

位 1 - 计数器复位: 为 1 时, 定时计数器和预分频计数器在 PCLK 的下一个上升沿同步复位

代码中: LPC_TMR32B0->TCR = 0x01 启动定时器; LPC_TMR32B1->TCR = 0x01 启动定时器 1

PR (预分频寄存器)

预分频计数器 PC 在每个 PCLK 周期加 1，直到与预分频寄存器 PR 的值相等。下一个 PCLK 将使预分频计数器 PC 复位，定时计数器 TC 加 1。

代码中：LPC_TMR32B0->PR = 47999 设置 48000 分频；
LPC_TMR32B1->PR = 47999 同样设置

MR0（匹配寄存器 0）

匹配寄存器值会不断地与定时计数器 TC 的值进行比较。当两个值相等时，自动触发相应动作。

代码中：LPC_TMR32B0->MR0 = 1000 设置 1 秒定时；LPC_TMR32B1->MR0 = 1000 同样设置

MCR（匹配控制寄存器）

位 0 - MR0I：MR0 上的中断，当 MR0 与 TC 值匹配时产生中断

位 1 - MR0R：MR0 上的复位，MR0 与 TC 值匹配将使 TC 复位

代码中：LPC_TMR32B0->MCR = (1UL << 0) | (1UL << 1) 使能匹配中断和复位；LPC_TMR32B1->MCR 同样配置

IR（中断寄存器）

位 0 - MR0 中断：匹配通道 0 的中断标志

代码中：LPC_TMR32B0->IR = 0x01 清除 MR0 中断标志；
LPC_TMR32B1->IR = 0x01 同样清除

ADC 相关寄存器：

CR（控制寄存器）

位 7:0 - 通道选择：选择要转换的 ADC 通道

位 15:8 - CLKDIV：ADC 时钟分频值

位 24 - START：软件启动转换

代码中：LPC_ADC->CR = (11UL << 8) | (1UL << 7) 设置 CLKDIV=11，
选择 AD7 通道；LPC_ADC->CR |= (1UL << 24) 启动转换

INTEN（中断使能寄存器）

位 7:0 - 通道中断使能：分别使能各通道的转换完成中断

代码中：LPC_ADC->INTEN = (1UL << 7) 使能 AD7 通道中断

DR[7]（数据寄存器）

位 15:6 - RESULT：ADC 转换结果（10 位有效）

位 31 - DONE: 转换完成标志

代码中: `adc_value = (LPC_ADC->DR[7] >> 6) & 0x3FF` 读取 AD7 转换结果

UART 相关寄存器:

LCR (线控制寄存器)

位 1:0 - 字长度选择: 11=8 位

位 2 - 停止位选择: 0=1 个停止位

位 3 - 奇偶校验使能: 1=使能校验

位 5:4 - 奇偶校验控制: 01=奇校验

位 7 - 除数锁存访问位 DLAB: 1=使能对除数锁存器访问

代码中: `LPC_UART->LCR = 0x83` 设置 DLAB=1; `LPC_UART->LCR = 0x0B` 设置通信参数

DLL/DLM (除数锁存寄存器)

UART 除数锁存器是 UART 波特率发生器的一部分, 用于分频 UART_PCLK 时钟以产生波特率时钟。

代码中: `LPC_UART->DLL = 4` 设置波特率除数低位; `LPC_UART->DLM = 0` 设置波特率除数高位

FDR (小数分频器寄存器)

位 3:0 - DIVADDVAL: 产生波特率的预分频除加数值

位 7:4 - MULVAL: 波特率预分频乘数值

代码中: `LPC_UART->FDR = 0x85` 设置 DIVADDVAL=5, MULVAL=8

FCR (FIFO 控制寄存器)

位 0 - FIFO 使能: 1=使能 UART 接收 FIFO 和发送 FIFO

位 1 - 接收 FIFO 复位: 写 1 清零 UART 接收 FIFO

位 2 - 发送 FIFO 复位: 写 1 清零 UART 发送 FIFO

代码中: `LPC_UART->FCR = 0x81` 使能 FIFO 并设置接收触发点

IER (中断使能寄存器)

位 0 - 接收中断使能: 使能 UART 的接收数据可用中断

代码中: `LPC_UART->IER = 0x01` 使能 RDA 中断

LSR (线状态寄存器)

位 0 - 接收数据就绪: 0=RBR 为空, 1=RBR 包含有效数据

位 5 - 发送 FIFO 空: 0=THR 包含有效数据, 1=THR 为空

代码中: `while((LPC_UART->LSR & (1UL << 5)) == 0)` 等待 THRE;
`if(LPC_UART->LSR & (1<<0))` 检查数据就绪

THR (发送保持寄存器)

写 UART 发送保持寄存器会使数据保存到 UART 发送 FIFO 中。

代码中: `LPC_UART->THR = data` 发送数据

RBR (接收缓冲寄存器)

RBR 是 UART 接收 FIFO 的最高字节, 包含最早接收到的字符。

代码中: `uart_command = LPC_UART->RBR` 读取接收数据

SPI 相关寄存器:

CR0 (控制寄存器 0)

位 3:0 - DSS: 数据长度选择, 0111=8 位数据

位 5:4 - FRF: 帧格式, 00=SPI

代码中: `LPC_SSP1->CR0 = 0x01C7` 配置 8 位数据, SPI 模式 0

CPSR (时钟预分频寄存器)

位 7:0 - CPSDVSR: 时钟预分频值

代码中: `LPC_SSP1->CPSR = 0x04` 设置预分频值

CR1 (控制寄存器 1)

位 1 - SSE: SSP 使能

代码中: `LPC_SSP1->CR1 = (1<<1)` 使能 SSP, 主模式

DR (数据寄存器)

数据寄存器用于数据的发送和接收

代码中: `LPC_SSP1->DR = tx_data` 发送数据; `return LPC_SSP1->DR`
接收数据

I2C 相关寄存器:

SCLH (SCL 高电平寄存器)

设置 I2C 时钟高电平周期

代码中: `LPC_I2C->SCLH = 250` 配置 100kHz 时钟

SCLL (SCL 低电平寄存器)

设置 I2C 时钟低电平周期

代码中: `LPC_I2C->SCLL = 250` 配置 100kHz 时钟

CONSET (控制置位寄存器)

位 4 - STO: 停止标志

位 5 - STA: 起始标志

位 6 - I2EN: I2C 接口使能

代码中: `LPC_I2C->CONSET |= (1<<5)` 产生起始条件;

`LPC_I2C->CONSET |= (1<<4)` 产生停止条件

CONCLR (控制清零寄存器)

位 3 - SIC: 清除 I2C 中断标志

位 4 - STOC: 清除 STO 标志

位 5 - STAC: 清除 STA 标志

代码中: `LPC_I2C->CONCLR = (1<<5) | (1<<3)` 清除起始条件和中断标志

DAT (数据寄存器)

包含要发送或刚接收的数据

代码中: `LPC_I2C->DAT = data` 发送数据; `data = (uint8_t)LPC_I2C->DAT` 接收数据

系统控制寄存器:

SYSAHBCLKCTRL (系统时钟控制寄存器)

控制各外设模块的时钟使能

代码中: `LPC_SYSCON->SYSAHBCLKCTRL |= (1UL << 16)` 使能 IOCON 时钟; `LPC_SYSCON->SYSAHBCLKCTRL |= (1UL << 12)` 使能 UART 时钟; `LPC_SYSCON->SYSAHBCLKCTRL |= (1UL << 13)` 使能 ADC 时钟; `LPC_SYSCON->SYSAHBCLKCTRL |= (1UL << 18)` 使能 SSP1 时钟

UARTCLKDIV (UART 时钟分频寄存器)

设置 UART 模块的时钟分频

代码中: `LPC_SYSCON->UARTCLKDIV = 4` UART 时钟=48MHz/4=12MHz

PRESETCTRL (外设复位控制寄存器)

控制各外设的复位状态

代码中：LPC_SYSCON->PRESETCTRL |= (1<<2) 释放 SSP1 复位；
LPC_SYSCON->PRESETCTRL |= (1<<1) 释放 I2C 复位

PDRUNCFG（功耗控制寄存器）

控制各模拟模块的电源

代码中：LPC_SYSCON->PDRUNCFG &= ~(1UL << 4) ADC 电源使能

4.3 重要的器件原理说明

系统时钟与定时器工作原理：

LPC1100 微控制器的定时器模块由系统时钟驱动，系统时钟频率为 48MHz。定时器的核心是一个 32 位递增计数器(TC)，其工作时钟经过预分频器分频得到。预分频器通过 PR 寄存器设置，分频系数为 PR+1。

定时器实现定时的具体工作流程：预分频计数器(PC)在每个 PCLK 时钟周期加 1。当 PC 的值等于 PR 的值时，在下一个 PCLK 周期 PC 复位为 0，同时 TC 加 1。当 TC 的值与匹配寄存器 MR0 的值相等时，触发匹配事件。

根据以上原理，本实验的两个定时器分别实现 1s 定时，需要计算相关参数和进行相关配置：

预分频值 PR = 47999，实际分频系数为 48000

定时器计数频率 = 48MHz / 48000 = 1kHz

匹配值 MR0 = 1000，对应定时时间 = 1000 / 1kHz = 1 秒

MCR 寄存器设置为 0x03（二进制 00000011）：位 0=1：MR0 匹配时产生中断；位 1=1：MR0 匹配时复位 TC

代码实现：

```
LPC_TMR32B0->PR = 47999;          // 48000分频
LPC_TMR32B0->MR0 = 1000;           // 1秒定时
LPC_TMR32B0->MCR = (1UL << 0) | (1UL << 1); // 中断+复位
LPC_TMR32B1->PR = 47999;          // 定时器1同样配置
LPC_TMR32B1->MR0 = 1000;
LPC_TMR32B1->MCR = (1UL << 0) | (1UL << 1);
```

定时器中断机制详细原理：

定时器中断流程：当 TC == MR0 时，硬件自动设置 IR[0] 标志位，NVIC 检测到中断请求，保存当前上下文，根据中断向量表跳转到对应的中断服务函数，在中断服务函数中读取 IR 寄存器判断中断源，接着清除中断标志（写 1 到 IR[0]），执行相应的中断服务操作，最后中断返回，恢复上下文。在本程序中，定时器 0 中断用于启动 ADC 转换，即每秒启动一次 ADC 温度模块的测温，而定时器 1 中断用于 LM75BD 温度模块温度的读取和记录。

具体代码中的中断处理：

```
void TIMER32_0_IRQHandler(void) {
    if ((LPC_TMR32B0->IR & 0x01) == 0x01) {
        LPC_TMR32B0->IR = 0x01; // 清除中断标志
        LPC_ADC->CR |= (1UL << 24); // 启动ADC转换
    }
}

void TIMER32_1_IRQHandler(void) {
    if ((LPC_TMR32B1->IR & 0x01) == 0x01) {
        LPC_TMR32B1->IR = 0x01; // 清除中断标志
        int16_t raw_temp = ReadRawTemperature(); // 读取LM75温度
        SaveTemperatureData(raw_temp); // 存储到Flash
        last_raw_temp = raw_temp;
        flash_data_ready = 1;
    }
}
```

ADC 模数转换器工作原理：

ADC 模块将模拟电压信号转换为数字值，采用逐次逼近型转换原理。本实验使用 ADC 通道 7（AD7）采集 NTC 热敏电阻的电压信号。

ADC 工作流程：首先配置 ADC 时钟分频和通道选择，然后通过软件触发启动转换。转换完成后产生中断，在中断服务函数中读取转换结果。

ADC 参数配置：

时钟分频 CLKDIV = 11，ADC 时钟频率 = 48MHz / (11+1) = 4MHz

选择 AD7 通道，配置 PI01_11 为模拟输入模式

使能 AD7 通道中断，转换完成后进入 ADC_IRQHandler

代码实现：

```
LPC_ADC->CR = (11UL << 8) | (1UL << 7); // CLKDIV=11，选择AD7
LPC_ADC->INTEN = (1UL << 7); // 使能AD7中断
```

```

void ADC_IRQHandler(void) {
    if (LPC_ADC->DR[7] & (1UL << 31)) {
        adc_value = (LPC_ADC->DR[7] >> 6) & 0x3FF; // 读取10位ADC值
        adc_data_ready = 1;
    }
}

```

UART 串口通信原理：

UART 采用异步串行通信方式，通过起始位、数据位、校验位和停止位构成数据帧。本实验配置为 115200 波特率、8 位数据、奇校验、1 位停止位。

UART 波特率生成原理：使用小数分频器实现精确的波特率控制。
计算公式：

$$\text{实际波特率} = \text{UART_PCLK} / (16 \times \text{DLL} \times \text{FR}), \text{ 其中 } \text{FR} = \text{MULVAL} / (\text{MULVAL} + \text{DIVADDVAL})$$

具体配置：

$$\text{UART 时钟} = 48\text{MHz} / 4 = 12\text{MHz}$$

$$\text{DLL} = 4, \text{DLM} = 0$$

$$\text{DIVADDVAL} = 5, \text{MULVAL} = 8, \text{FR} = 8/13 \approx 0.615$$

$$\text{实际波特率} = 12\text{MHz} / (16 \times 4 \times 0.615) \approx 115384, \\ \text{误差 } 0.16\%$$

代码实现：

```

LPC_SYSCON->UARTCLKDIV = 4; // UART时钟分频
LPC_UART->LCR = 0x83; // DLAB=1
LPC_UART->DLL = 4; LPC_UART->DLM = 0; // 除数设置
LPC_UART->FDR = 0x85; // DIVADDVAL=5, MULVAL=8
LPC_UART->LCR = 0x0B; // 8位数据，奇校验，1停止位

```

SPI Flash 存储器工作原理：

SPI 接口采用主从模式，通过 SCK、MOSI、MISO、CS 四线进行全双工通信。本实验使用 SPI 模式 0（CPOL=0，CPHA=0）。LPC1114 接入的 XT25F02 使用 SPI 协议通信。

SPI 数据交换机制：

SPI_ExchangeByte 函数实现 SPI 数据的全双工传输。当主设备向 DR 寄存器写入数据时，数据被装入发送 FIFO 并开始传输，同时接收 FIFO 开始接收从设备返回的数据。传输过程中需要检查状态寄存器

SR 的位 4（BSY 标志）判断 SPI 控制器是否繁忙，检查位 2（RNE 标志）判断接收 FIFO 是否有数据可读。

代码实现细节：

```
uint8_t SPI_ExchangeByte(uint8_t tx_data) {  
    while ((LPC_SSP1->SR & (1 << 4)) == (1 << 4)); // 等待BSY=0，控制器不忙  
    LPC_SSP1->DR = tx_data; // 写入发送数据，启动传输  
    while ((LPC_SSP1->SR & (1 << 2)) != (1 << 2)); // 等待RNE=1，接收FIFO非空  
    return LPC_SSP1->DR; // 读取接收到的数据  
}
```

Flash 读数据操作详细机制：

以 Flash_ReadData 函数为例，完整展示 Flash 读取操作的全过程：

1. 片选使能：拉低 CS 信号，选中 Flash 器件

```
FLASH_CS_LOW();
```

2. 发送读指令：发送 0x03 指令字节，指示进行数据读取操作

```
SPI_ExchangeByte(FLASH_ReadData); // 0x03
```

3. 发送 24 位地址：分三次发送地址的高字节、中字节和低字节

```
SPI_ExchangeByte((uint8_t)(address >> 16)); // 地址位23-16
```

```
SPI_ExchangeByte((uint8_t)(address >> 8)); // 地址位15-8
```

```
SPI_ExchangeByte((uint8_t)address); // 地址位7-0
```

4. 连续读取数据：循环调用 SPI_ExchangeByte 读取指定长度的数据

```
for (i = 0; i < length; i++) {  
    buffer[i] = SPI_ExchangeByte(0xFF); // 发送哑元数据0xFF，同时接收有效数据  
}
```

5. 片选禁用：拉高 CS 信号，结束本次通信

```
FLASH_CS_HIGH();
```

Flash 页编程（写）操作机制：

Flash_WritePage 函数实现数据写入，流程更为复杂：

1. 写使能：必须先发送写使能指令，否则写入操作被拒绝

```
Flash_WriteEnable(); // 发送 0x06 指令
```

2. 等待就绪：检查状态寄存器的 BUSY 位，确保 Flash 准备好接收数据

```
Flash_WaitBusy(); // 等待 BUSY=0
```

3. 页编程指令序列:

```
FLASH_CS_LOW();  
SPI_ExchangeByte(FLASH_PageProgram); // 0x02  
SPI_ExchangeByte((uint8_t)(address >> 16));  
SPI_ExchangeByte((uint8_t)(address >> 8));  
SPI_ExchangeByte((uint8_t)address);
```

4. 数据写入: 连续写入一页数据 (最多 256 字节)

```
for (i = 0; i < length; i++) {  
    SPI_ExchangeByte(data[i]); // 发送要写入的数据  
}
```

5. 等待写入完成: 写入操作需要时间, 必须等待 BUSY 位清零

```
FLASH_CS_HIGH();  
Flash_WaitBusy(); // 等待编程操作完成
```

Flash 状态管理机制:

通过 Flash_ReadStatus 函数读取状态寄存器, 关键状态位包括:

- 位 0 (BUSY): 1 表示忙, 0 表示就绪
- 位 1 (WEL): 写使能锁存, 1 表示写使能

在执行任何写入操作前, 必须先发送写使能指令, 且该指令在电源周期或写禁用指令前保持有效。

代码中的实际应用:

在温度数据存储中, SaveTemperatureData 函数将 16 位温度数据拆分为 2 字节后写入 Flash:

```
uint8_t temp_data[2];  
temp_data[0] = (raw_temp >> 8) & 0xFF; // 高字节  
temp_data[1] = raw_temp & 0xFF; // 低字节  
Flash_WritePage(temp_data, flash_address, 2);
```

这种机制确保了温度数据能够可靠地存储在非易失性存储器中, 即使系统断电数据也不会丢失。此外 Flash 存储器采用顺序写入策略:

- 每个温度记录占用 2 字节 (16 位原始数据)
- 地址自动递增, 记录计数器跟踪数据量
- 当地址接近容量上限时自动回绕
- 支持记录读取和 Flash 擦除命令

I2C 温度传感器工作原理:

I2C 总线采用两线制 (SDA、SCL)，支持多主多从通信。本实验配置为 100kHz 标准模式，与 LM75 温度传感器通信。

I2C 通信时序:

起始条件: SCL 高电平时 SDA 由高变低

设备地址: LM75 地址为 0x91 (读模式)

数据传送: 每个字节后跟应答位

停止条件: SCL 高电平时 SDA 由低变高

LM75 温度数据格式:

16 位数据，高 9 位为温度值，分辨率 0.125°C

温度值 = (原始数据 $\gg 5$) $\times 0.125$

代码实现:

```
void I2C_Start(void) {  
    LPC_I2C->CONSET |= (1 << 5); // STA=1  
    while (!(LPC_I2C->CONSET & (1 << 3))); // 等待SI置位  
    LPC_I2C->CONCLR = (1 << 5) | (1 << 3); // 清除STA和SI  
}
```

温度计算原理:

本实验同时使用 NTC 热敏电阻和 LM75BD，使用的温度转换机制不同，下面具体介绍。

对于 NTC 热敏电阻温度计算，基于查找表和线性插值法，通过 ADC 值计算 NTC 电阻值，再通过查找表转换为温度值。

计算公式:

$$\text{电压值} = (\text{ADC 值} \times 3.3\text{V}) / 1023$$

$$\text{NTC 电阻值} = (3.3\text{V} - \text{电压值}) \times 10\text{k}\Omega / \text{电压值}$$

LM75 数字温度计算:

原始 16 位数据，高 9 位为有效温度值

$$\text{温度值} = (\text{原始数据} \gg 5) \times 0.125^{\circ}\text{C}$$

中断系统协调原理:

系统同时使用两个计数器中断、ADC 中断、UART 中断，为了保证程序协调，系统采用多级中断协调机制:

定时器 0 中断 (1Hz) → 触发 ADC 转换 → ADC 转换完成中断 → 读取 ADC 值

定时器 1 中断 (1Hz) → 读取 LM75 温度 → 存储到 Flash

UART 接收中断 → 处理用户命令

通过全局变量 `adc_data_ready` 和 `flash_data_ready` 实现任务同步，在主循环中统一处理数据显示，然后继续等待下一轮中断。

五、软件流程图

软件分为主函数、ADC 模块、定时器模块和 NTC 相关函数部分。采用定时器 32_0 启动 ADC 转换，具体运行流程如下：

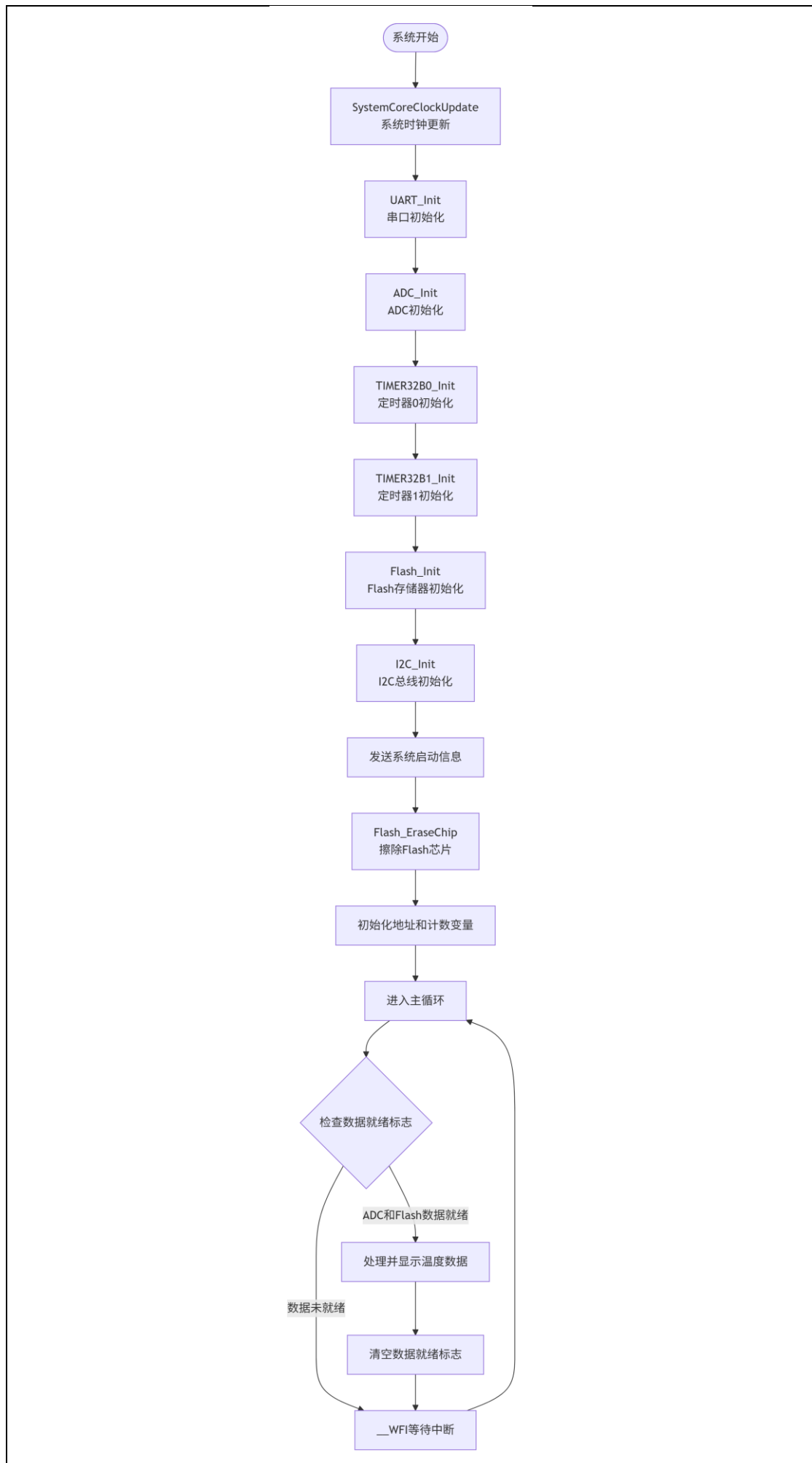
系统上电后，主函数首先执行硬件初始化。依次初始化 UART 串口用于通信显示，配置 ADC 模块的 AD7 通道用于采集 NTC 热敏电阻电压，设置定时器 32_0 产生 1 秒定时中断来触发 ADC 转换，同时初始化 Flash 存储器和 I2C 总线用于 LM75 温度传感器。

初始化完成后，系统进入主循环等待状态。定时器 32_0 每 1 秒产生一次中断，在中断服务程序中通过软件触发启动 ADC 转换。ADC 转换完成后产生中断，读取转换结果并计算 NTC 电阻值，通过查找表线性插值法将电阻值转换为温度值，设置数据就绪标志。

当 NTC 温度数据和 LM75 温度数据都准备就绪时，主程序通过 UART 串口显示两种传感器的温度值。显示内容包括 ADC 原始值、NTC 计算温度、LM75 数字温度以及当前记录编号。数据显示完成后清空就绪标志，系统继续等待下一次定时触发。

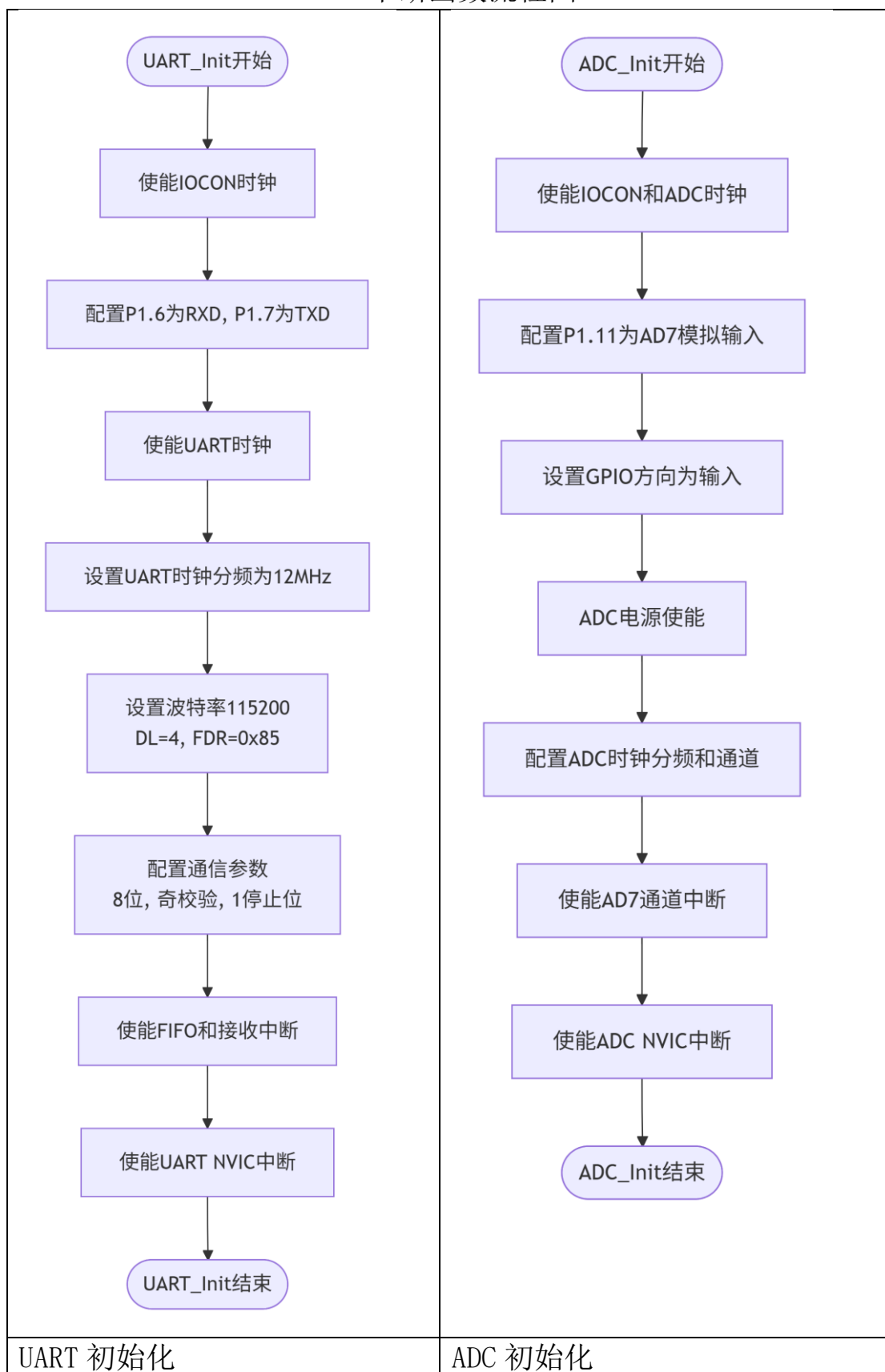
整个系统通过定时器中断驱动的方式，实现了对 NTC 热敏电阻和 LM75 温度传感器的同步采集、存储和显示，形成了完整的温度监测解决方案。用户还可以通过串口命令读取历史记录或清除 Flash 存储数据。

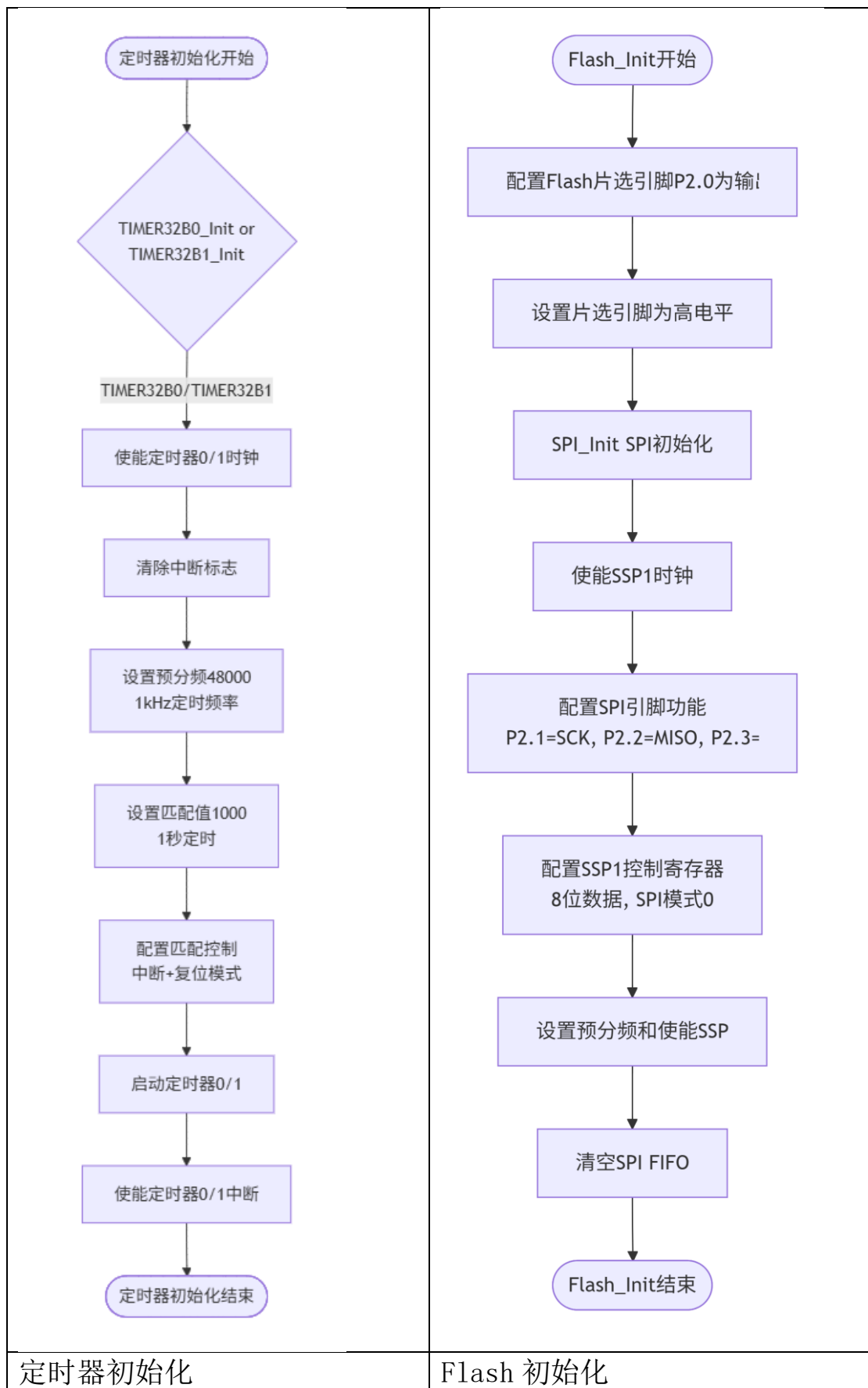
主函数流程图：

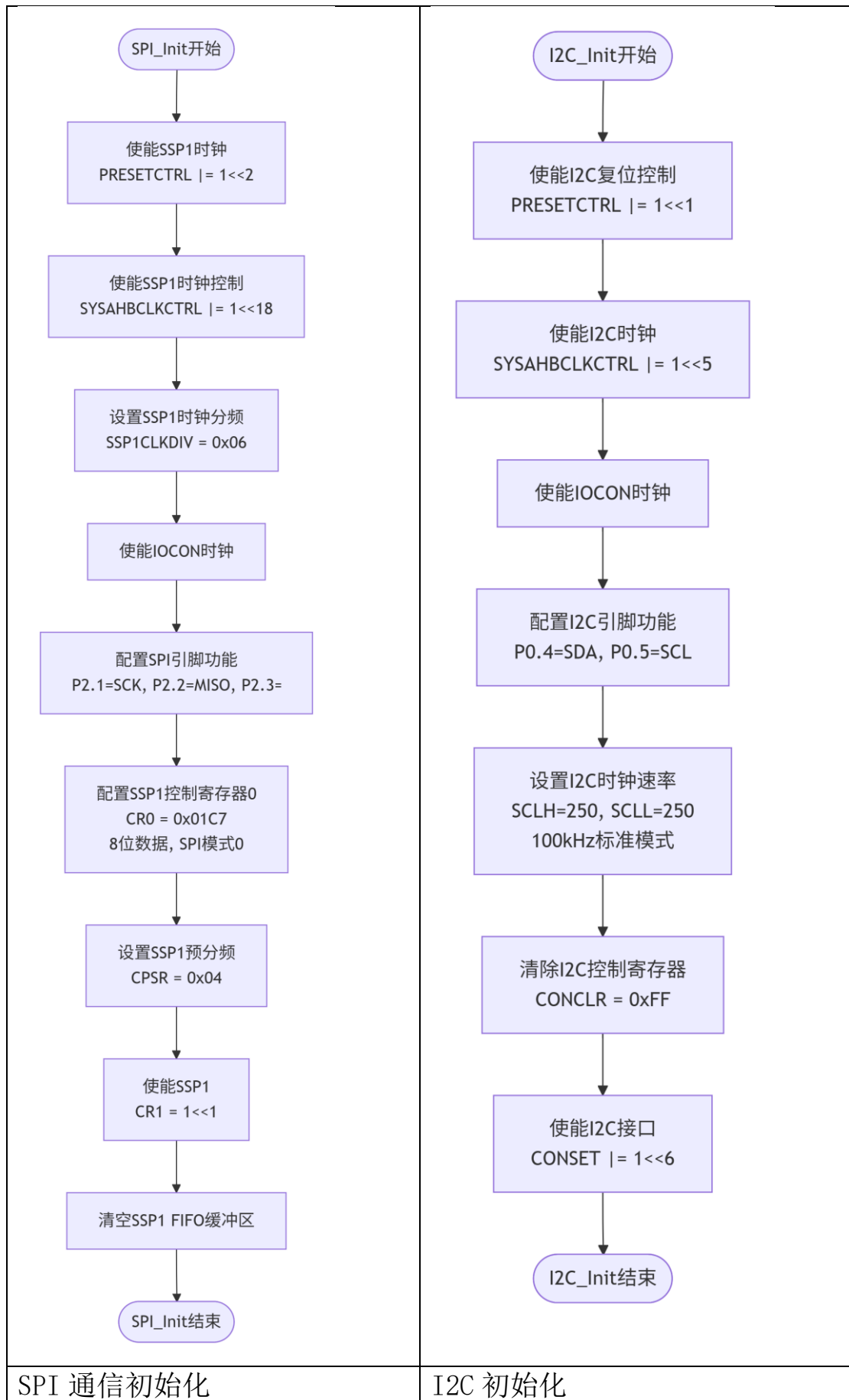


主函数流程图

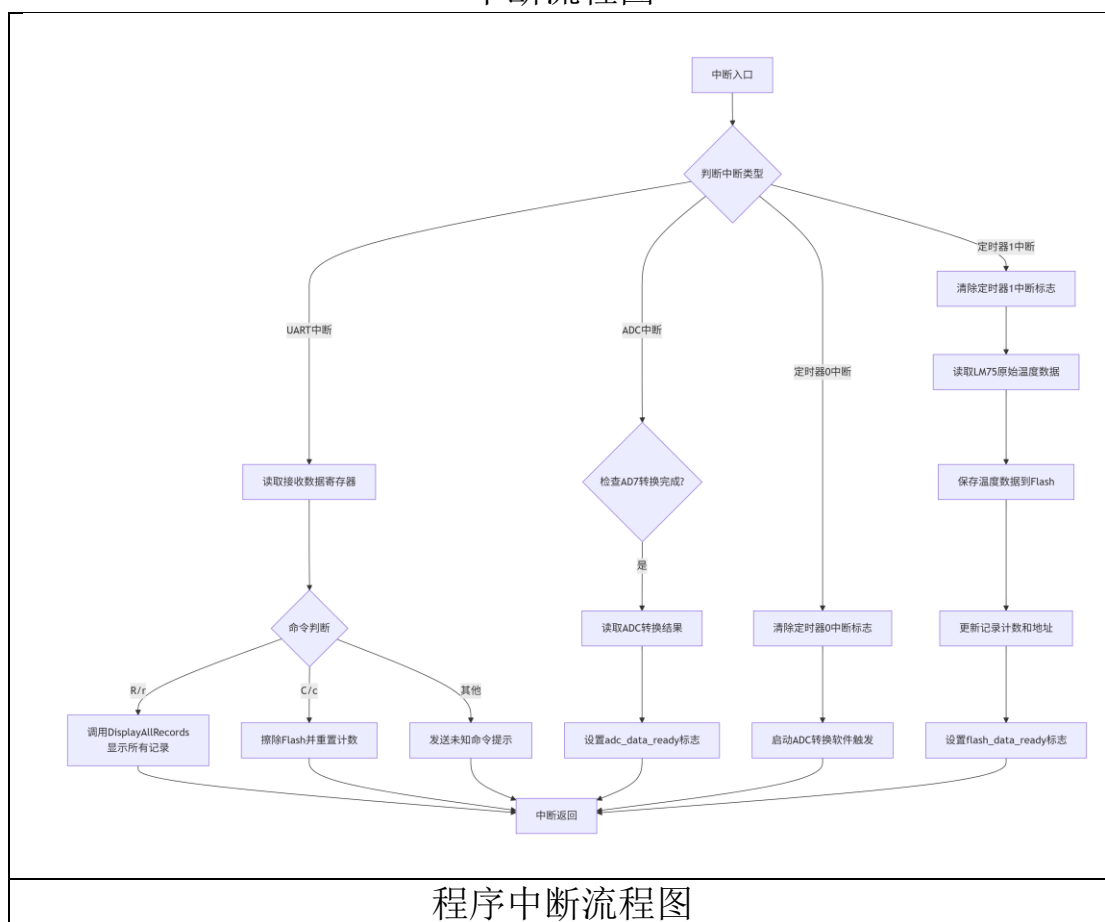
中断函数流程图





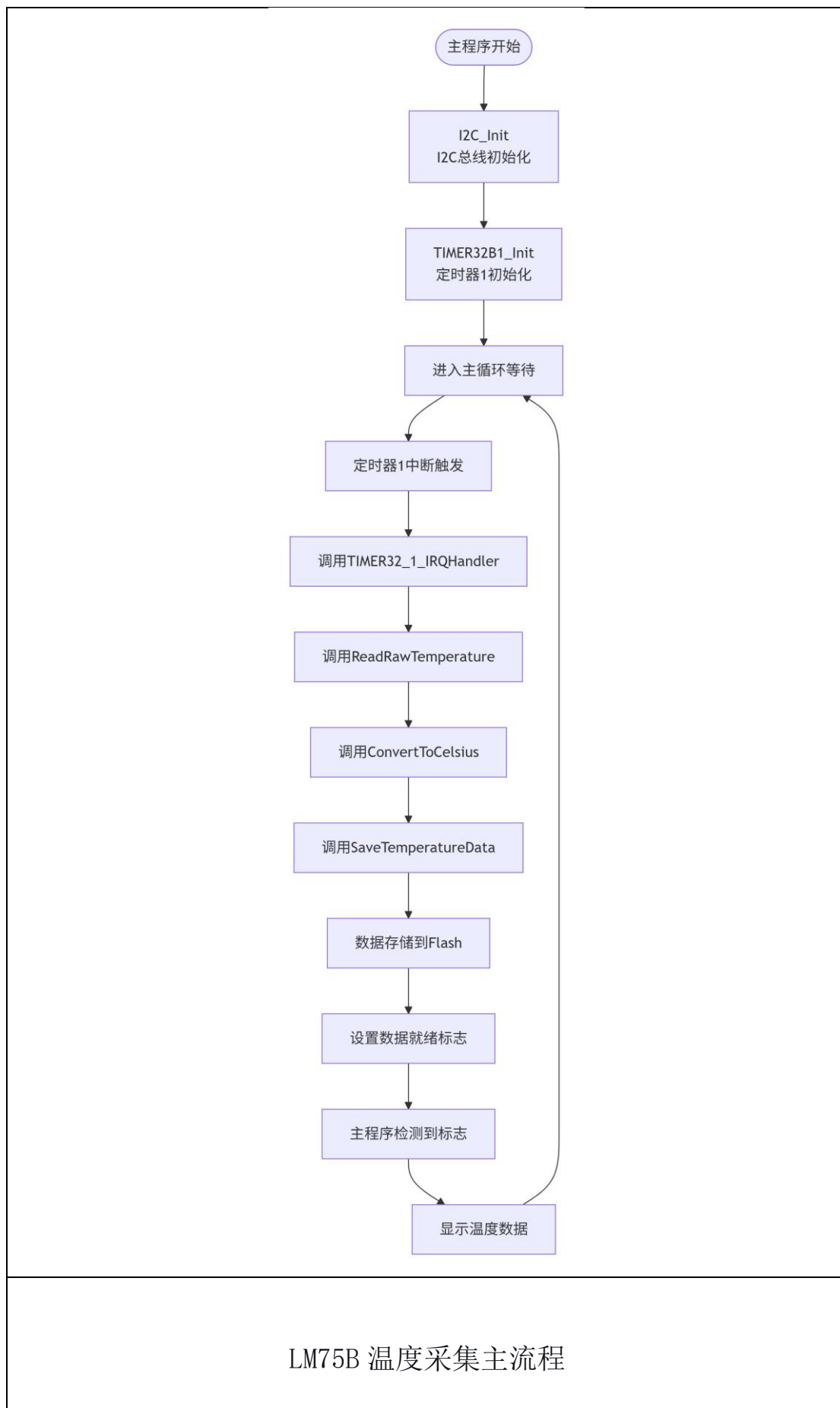


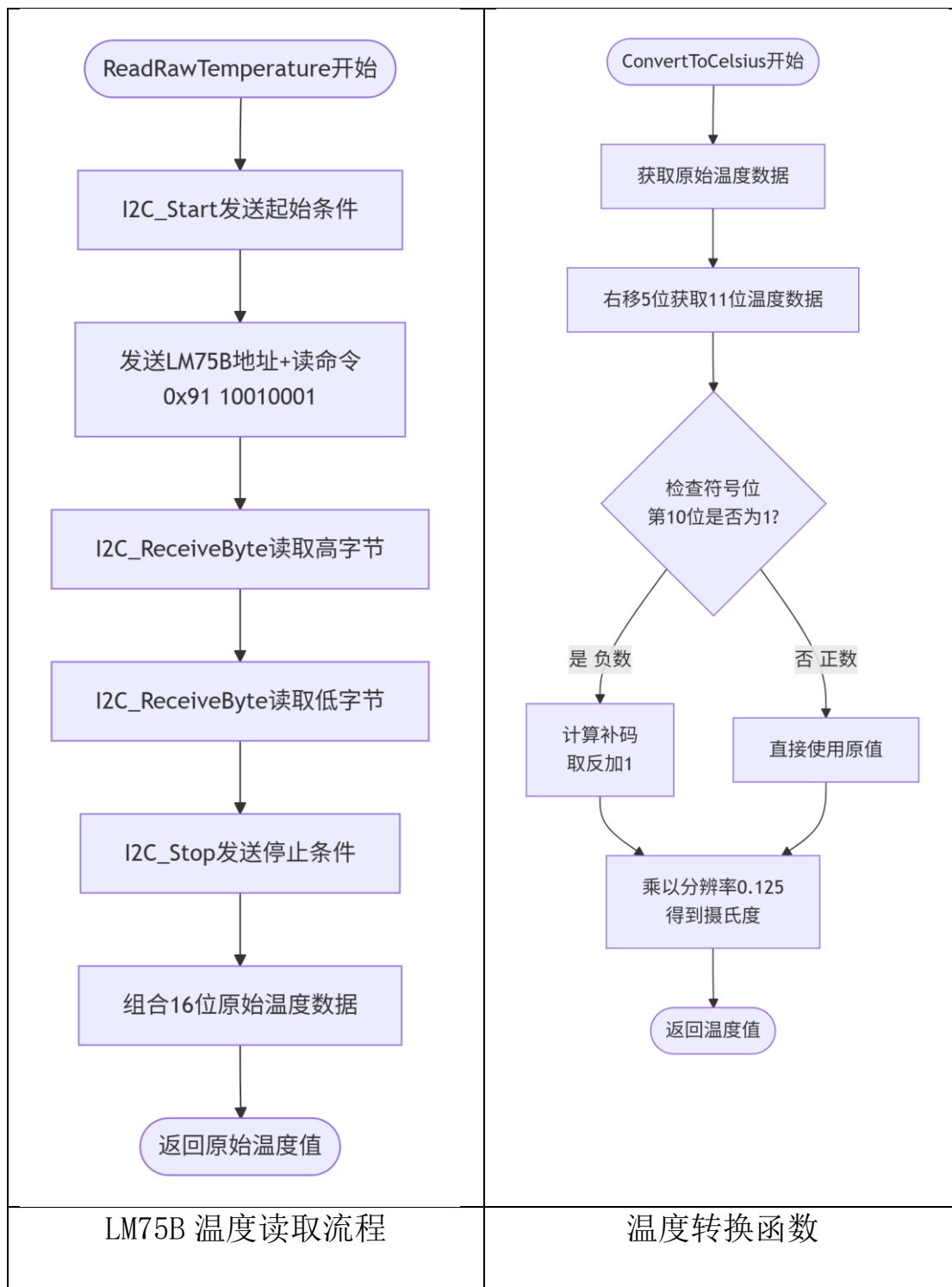
中断流程图

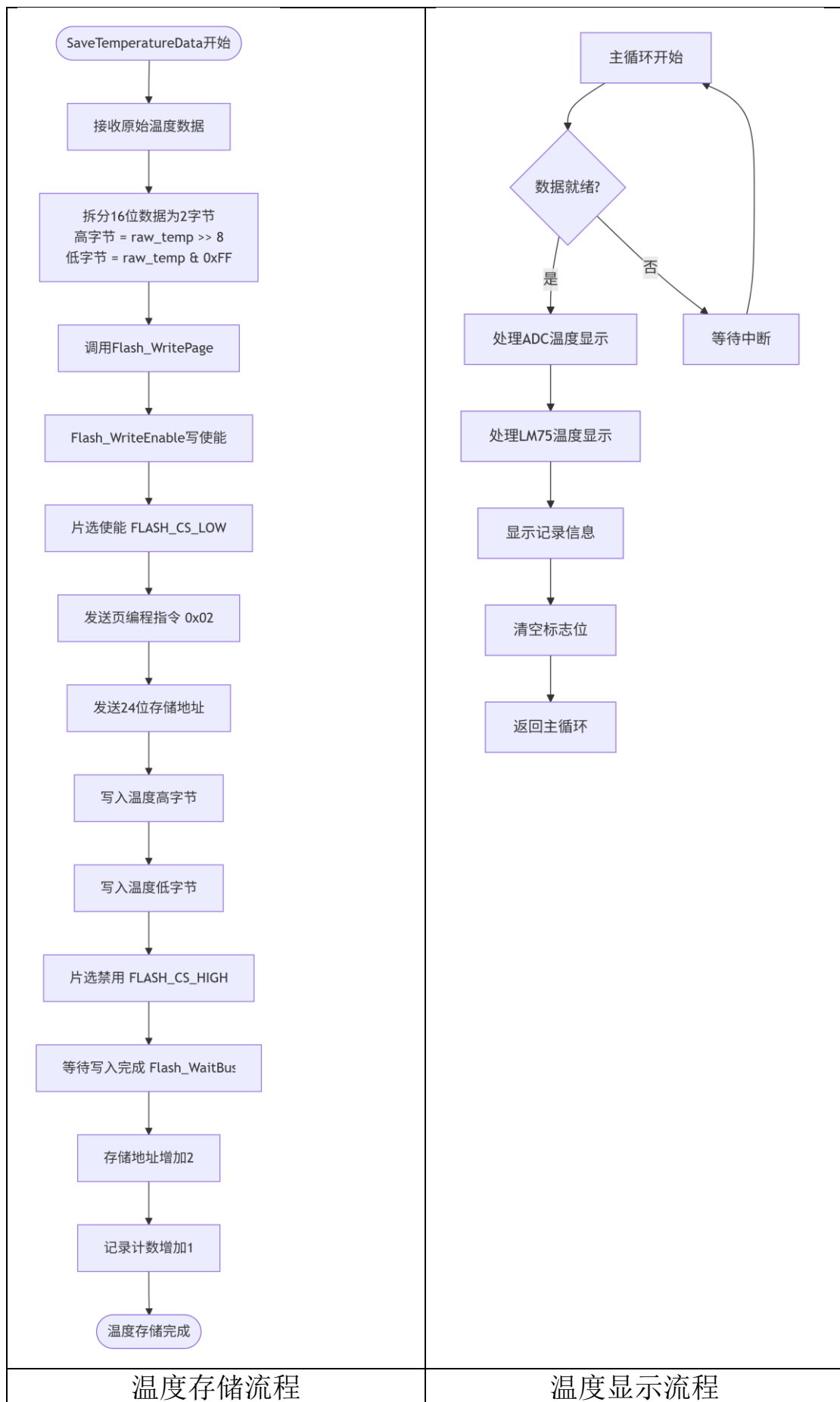


LM75BD 温度相关函数

LM75BD 温度读取通过定时器 1 中断激活，激活后，通过 ReadRawTemperature 函数读取原始温度值（二进制原始数据），再由 SaveTemperatureData（）保存到 flash 中，并不在中断函数中进行转换，这样可以直接在 flash 中存储原始数据，节省空间，提升程序运行效率，随后设置标志位，等待主函数调用，温度转换函数在主函数中使用并通过 UART 发送，这样实现了温度的读取。

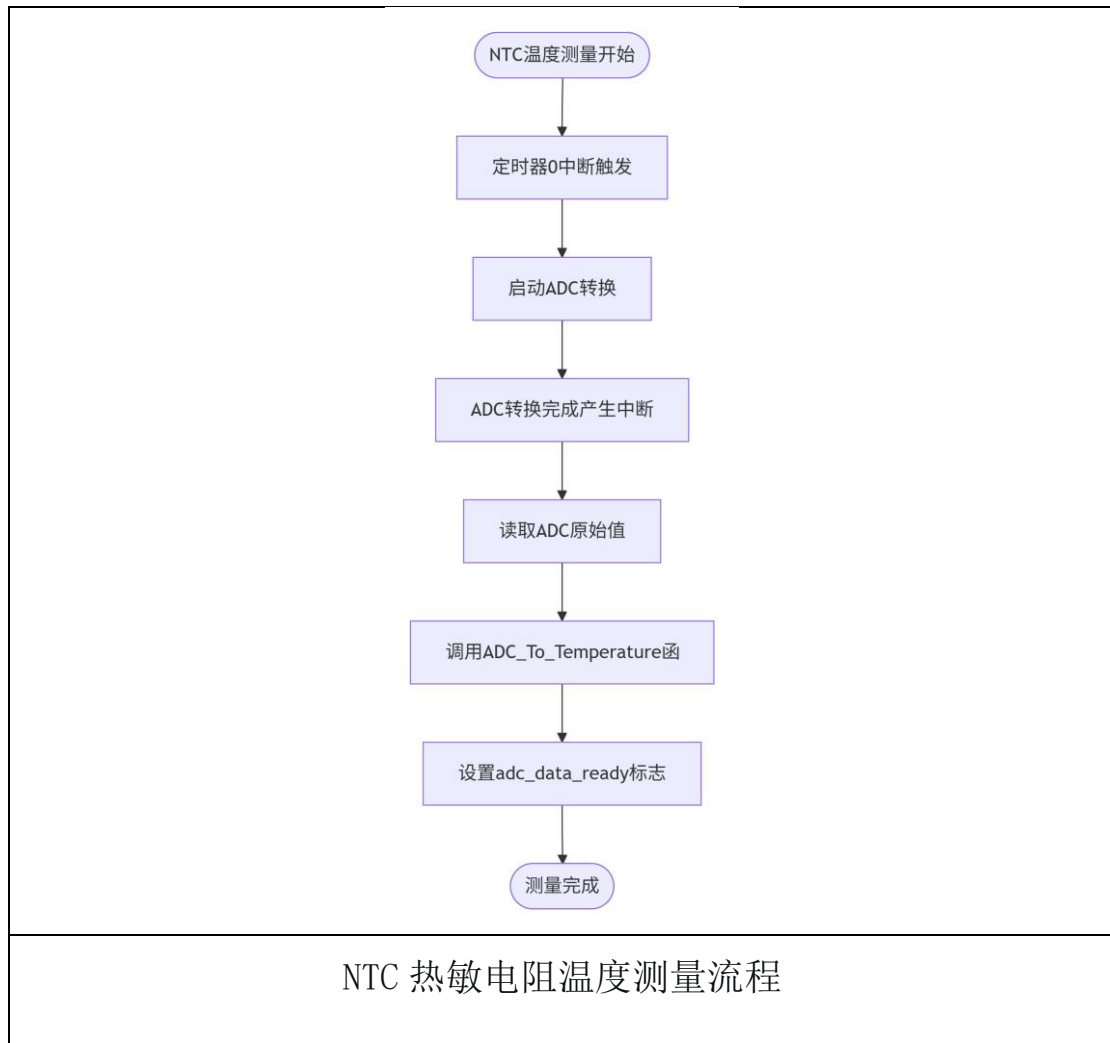


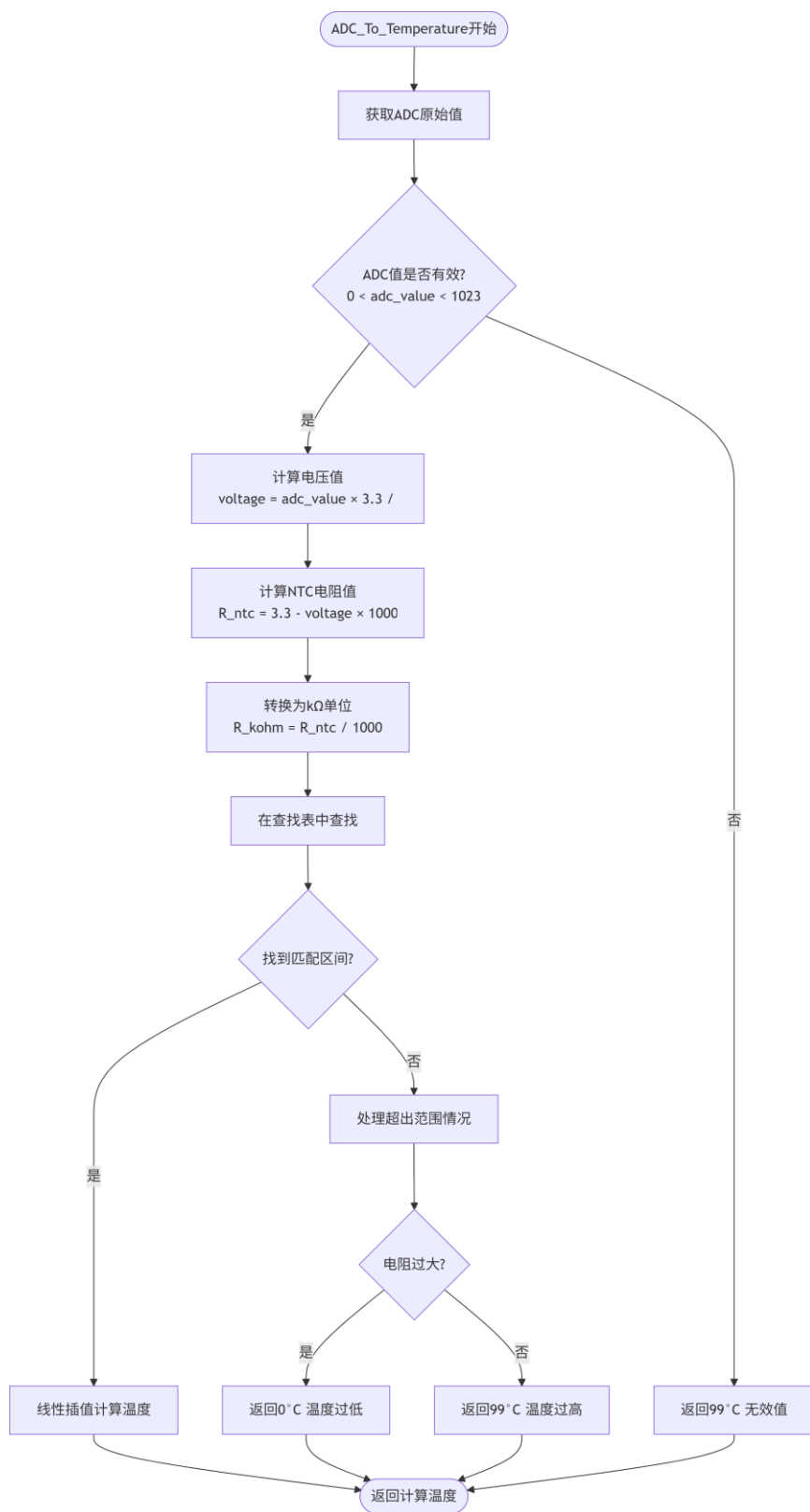




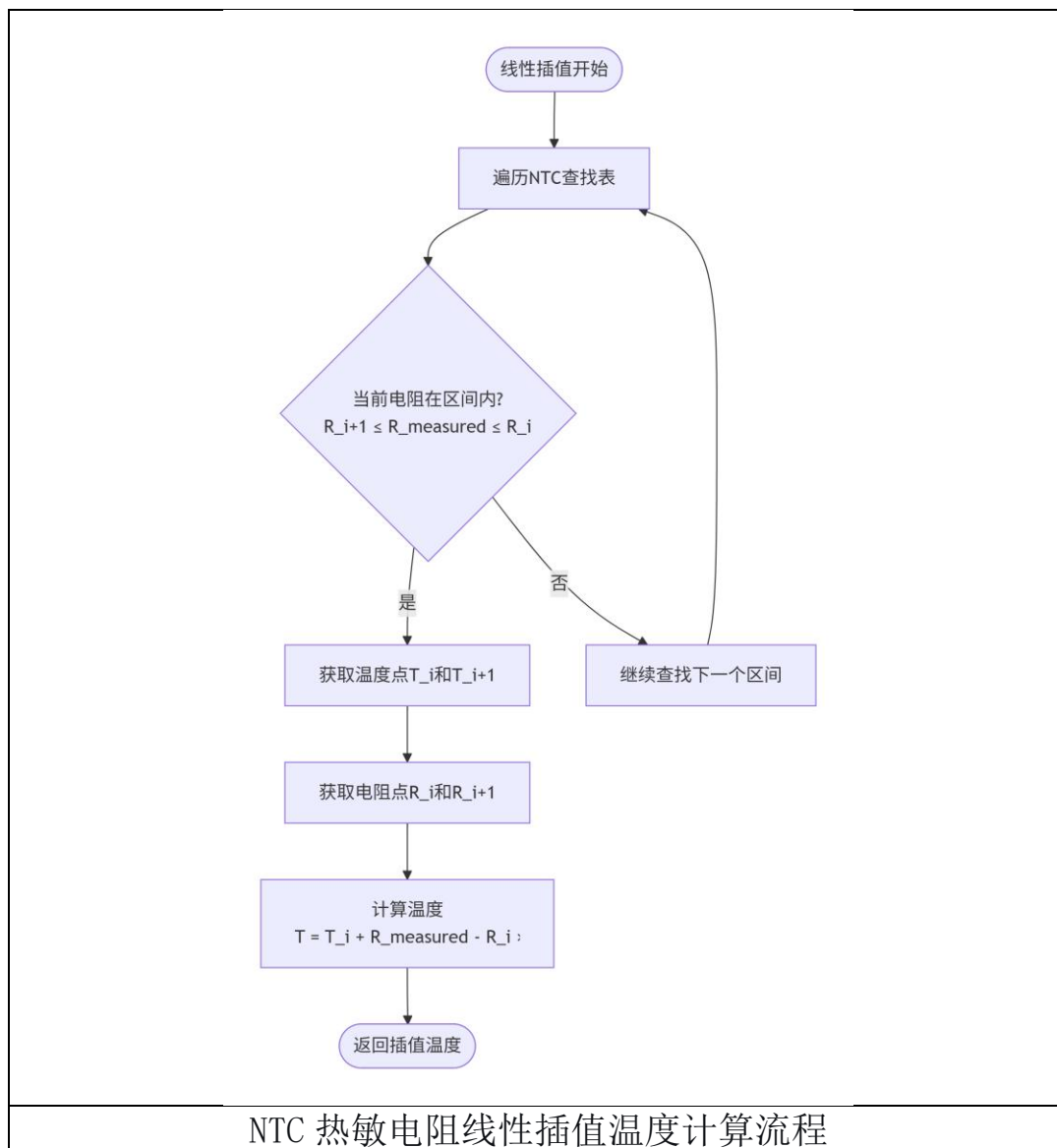
NTC 热敏电阻温度采集相关流程

NTC 热敏电阻温度测量依赖定时器触发,即定时器 0 每 1 秒触发一次 ADC 转换,ADC 完成后进入其中断,并获取数据,然后设置标志位,等待主函数使用,主函数调用转换函数并完成 UART 发送。





NTC 热敏电阻温度测量流程



六、程序代码

```

/*
 * 姓名：雷灿曦
 * 日期：2025-11-10
 * 描述：温度测量程序
 */

```

```

#include "LPC11xx.h"
#include <stdio.h>
#include <string.h>

```

```

// 引脚定义

```

```

#define FLASH_CS_HIGH()  LPC_GPIO2->DATA |= (1<<0)
#define FLASH_CS_LOW()   LPC_GPIO2->DATA &= ~(1<<0)

```

```

// Flash指令
#define FLASH_WriteEnable 0x06
#define FLASH_ReadStatusReg 0x05
#define FLASH_ReadData 0x03
#define FLASH_PageProgram 0x02
#define FLASH_SectorErase 0x20
#define FLASH_ChipErase 0xC7

// 全局变量
volatile uint16_t adc_value = 0;
volatile uint8_t adc_data_ready = 0;
volatile uint8_t flash_data_ready = 0;
volatile uint8_t uart_command = 0;
uint32_t flash_address = 0;
uint16_t record_count = 0;
int16_t last_raw_temp;

// NTC温度查找表
typedef struct {
    int16_t temp;
    uint16_t resistance;
} ntc_table_t;

// NTC温度查找表 - 基于规格书R-T表 (10kΩ @ 25° C, B=3950K)
const ntc_table_t ntc_table[] = {
    {0, 32814}, {10, 19958}, {20, 12504}, {25, 10000},
    {30, 8049}, {40, 5313}, {50, 3586}, {60, 2472},
    {70, 1742}, {80, 1249}, {90, 911}, {100, 675},
    {110, 508}, {120, 388}, {125, 341}
};

const uint8_t ntc_table_size = 19;

// 函数声明
void UART_Init(void);
void ADC_Init(void);
void TIMER32B0_Init(void);
void TIMER32B1_Init(void);
void GPIO_Init(void);
void SPI_Init(void);
void Flash_Init(void);
void I2C_Init(void);
void UART_SendString(char* str);
void UART_SendByte(uint8_t data);

```

```

void SendNumber(uint16_t num);
void SendFloat(float value, uint8_t decimals);
uint8_t SPI_ExchangeByte(uint8_t tx_data);
uint8_t Flash_ReadStatus(void);
void Flash_WriteEnable(void);
void Flash_WaitBusy(void);
void Flash_ReadData(uint8_t* buffer, uint32_t address, uint16_t length);
void Flash_WritePage(uint8_t* data, uint32_t address, uint16_t length);
void Flash_EraseChip(void);
void I2C_Start(void);
void I2C_Stop(void);
void I2C_SendByte(uint8_t data);
uint8_t I2C_ReceiveByte(void);
int16_t ReadRawTemperature(void);
float ConvertToCelsius(int16_t raw_temp);
void SaveTemperatureData(int16_t raw_temp);
void DisplayAllRecords(void);
void ProcessUARTCommand(void);
float ADC_To_Temperature(uint16_t adc_value);

/*****UART函数*****/
void UART_Init(void) {
    LPC_SYSCON->SYSAHBCLKCTRL |= (1UL << 16); /* enable clock for IOCON */

    // 配置P1.6为RXD, P1.7为TXD
    LPC_IOCON->PIO1_6 &= ~0x07; // 清除功能位
    LPC_IOCON->PIO1_6 |= 0x01; /* P1.6 is RxD */
    LPC_IOCON->PIO1_7 &= ~0x07; // 清除功能位
    LPC_IOCON->PIO1_7 |= 0x01; /* P1.7 is TxD */

    LPC_SYSCON->SYSAHBCLKCTRL |= (1UL << 12); /* Enable clock to UART */
    LPC_SYSCON->UARTCLKDIV = 4; /* UART clock = 48MHz / 4 = 12MHz */

    // 设置波特率: DLAB=1, 允许访问DLL/DLM
    LPC_UART->LCR = 0x83; /* 8 bits, 1 Stop bit, DLAB=1 */

    // 设置波特率参数
    LPC_UART->DLL = 4; /* 115200 Baud Rate @ 12.0 MHz PCLK */
    LPC_UART->FDR = 0x85; /* FR 0.615, DIVADDVAL=5, MULVAL=8 */
    LPC_UART->DLM = 0; /* High divisor latch = 0 */

    // 设置通信参数: 8位数据, 奇校验, 1位停止位, DLAB=0
    LPC_UART->LCR = 0x0B; /* 8 bits, Odd Parity, 1 Stop bit, DLAB=0 */

    // 使能FIFO并设置触发点

```

```

LPC_UART->FCR = 0x81; /* 使能FIFO, 接收触发点为1字节 */

// 使能接收中断
LPC_UART->IER = 0x01;

// 使能UART中断
NVIC_EnableIRQ(UART_IRQn);
}

void UART_SendString(char* str) {
    while (*str) {
        UART_SendByte(*str++);
    }
}

void UART_SendByte(uint8_t data) {
    while ((LPC_UART->LSR & (1UL << 5)) == 0); // 等待THRE为空
    LPC_UART->THR = data;
}

void SendNumber(uint16_t num) {
    char buffer[6];
    uint8_t i = 0;

    if (num == 0) {
        UART_SendByte('0');
        return;
    }

    // 将数字转换为字符串
    while (num > 0) {
        buffer[i++] = '0' + (num % 10);
        num /= 10;
    }

    // 反向发送
    while (i > 0) {
        UART_SendByte(buffer[--i]);
    }
}

void SendFloat(float value, uint8_t decimals) {
    // 发送整数部分
    int integer_part = (int)value;

```

```

    if (integer_part < 0) {
        UART_SendByte('-');
        integer_part = -integer_part;
        value = -value;
    }
    SendNumber(integer_part);

    if (decimals > 0) {
        UART_SendByte('.');

        // 发送小数部分
        float fractional = value - (int)value;
        for (uint8_t i = 0; i < decimals; i++) {
            fractional *= 10;
            int digit = (int)fractional;
            UART_SendByte('0' + digit);
            fractional -= digit;
        }
    }
}

/*****ADC函数*****/
void ADC_Init(void) {
    // 使能IOCON和ADC时钟
    LPC_SYSCON->SYSAHBCLKCTRL |= (1UL << 6) | (1UL << 16) | (1UL << 13);

    // 配置P1.11为AD7模拟输入
    LPC_IOCON->PIO1_11 &= ~0x07; // 清除功能位
    LPC_IOCON->PIO1_11 = 0x01;    // 设置为ADC功能，模拟输入

    // 确保GPIO方向为输入
    LPC_GPIO1->DIR &= ~(1UL << 11);

    // ADC电源使能
    LPC_SYSCON->PDRUNCFG &= ~(1UL << 4);

    // ADC时钟分频和通道选择
    LPC_ADC->CR = (11UL << 8) | // CLKDIV = 11
                  (1UL << 7);  // 选择AD7通道

    // 使能AD7通道中断
    LPC_ADC->INTEN = (1UL << 7);

    // NVIC中断使能

```

```

    NVIC_EnableIRQ(ADC_IRQn);
}

// 更正的ADC转温度函数, version2
float ADC_To_Temperature(uint16_t adc_value) {
    if (adc_value == 0 || adc_value >= 1023) {
        return 99.0f; // 无效值
    }

    // 计算NTC电阻值 (使用10k上拉电阻)
    float voltage = (adc_value * 3.3f) / 1023.0f;
    float ntc_resistance = (3.3f - voltage) * 10000.0f / voltage;

    // 将电阻值转换为kΩ单位以匹配查找表
    float ntc_resistance_kohm = ntc_resistance / 1000.0f;

    // 查找相邻点进行线性插值
    for (uint8_t i = 0; i < ntc_table_size - 1; i++) {
        if (ntc_resistance_kohm <= ntc_table[i].resistance &&
            ntc_resistance_kohm >= ntc_table[i + 1].resistance) {

            // 线性插值
            float temp1 = ntc_table[i].temp;
            float temp2 = ntc_table[i + 1].temp;
            float R1 = ntc_table[i].resistance;
            float R2 = ntc_table[i + 1].resistance;

            return temp1 + (ntc_resistance_kohm - R1) * (temp2 - temp1) / (R2 - R1);
        }
    }

    // 超出表格范围的处理
    if (ntc_resistance_kohm > ntc_table[0].resistance) {
        return 0.0f; // 温度过低
    }
    else {
        return 99.0f; // 温度过高
    }
}

/*****定时器初始化*****/
// 32位定时器0初始化 - 1秒定时, 用于触发ADC转换
void TIMER32B0_Init(void) {
    // 使能定时器0时钟

```

```

LPC_SYSCON->SYSAHBCLKCTRL |= (1UL << 9);

// 清除所有中断标志
LPC_TMR32B0->IR = 0x1F;

// 预分频设置 (48MHz/48000 = 1kHz)
LPC_TMR32B0->PR = 47999; // 48000分频

// 匹配寄存器 - 1秒定时 (1kHz * 1000 = 1秒)
LPC_TMR32B0->MR0 = 1000;

// 匹配控制：中断 + 复位
LPC_TMR32B0->MCR = (1UL << 0) | (1UL << 1);

// 启动定时器
LPC_TMR32B0->TCR = 0x01;

// NVIC中断使能
NVIC_EnableIRQ(TIMER_32_0_IRQn);
}

// 32位定时器1初始化 - 1秒定时，用于读取LM75温度并存储到Flash
void TIMER32B1_Init(void) {
    // 使能定时器1时钟
    LPC_SYSCON->SYSAHBCLKCTRL |= (1UL << 10);

    // 清除所有中断标志
    LPC_TMR32B1->IR = 0x1F;

    // 预分频设置 (48MHz/48000 = 1kHz)
    LPC_TMR32B1->PR = 47999; // 48000分频

    // 匹配寄存器 - 1秒定时 (1kHz * 1000 = 1秒)
    LPC_TMR32B1->MR0 = 1000;

    // 匹配控制：中断 + 复位
    LPC_TMR32B1->MCR = (1UL << 0) | (1UL << 1);

    // 启动定时器
    LPC_TMR32B1->TCR = 0x01;

    // NVIC中断使能
    NVIC_EnableIRQ(TIMER_32_1_IRQn);
}

```

```

/*****SPI和Flash函数*****/
uint8_t SPI_ExchangeByte(uint8_t tx_data)
{
    while ((LPC_SSP1->SR & (1 << 4)) == (1 << 4)); // 等待不忙
    LPC_SSP1->DR = tx_data;
    while ((LPC_SSP1->SR & (1 << 2)) != (1 << 2)); // 等待接收完成
    return LPC_SSP1->DR;
}

void SPI_Init(void)
{
    uint8_t i;

    // 使能SSP1时钟
    LPC_SYSCON->PRESETCTRL |= (1 << 2);
    LPC_SYSCON->SYSAHBCLKCTRL |= (1 << 18);
    LPC_SYSCON->SSP1CLKDIV = 0x06;

    // 配置SPI引脚
    LPC_SYSCON->SYSAHBCLKCTRL |= (1 << 16);
    LPC_IOCON->PIO2_1 = 0x02; // SCK
    LPC_IOCON->PIO2_2 = 0x02; // MISO
    LPC_IOCON->PIO2_3 = 0x02; // MOSI
    LPC_SYSCON->SYSAHBCLKCTRL &= ~(1 << 16);

    // 配置SSP1
    LPC_SSP1->CR0 = 0x01C7; // 8位数据，SPI模式0
    LPC_SSP1->CPSR = 0x04; // 预分频
    LPC_SSP1->CR1 = (1 << 1); // 使能SSP

    // 清空FIFO
    for (i = 0; i < 8; i++)
    {
        volatile uint8_t clear = LPC_SSP1->DR;
    }
}

void Flash_Init(void)
{
    LPC_GPIO2->DIR |= (1 << 0);
    FLASH_CS_HIGH();
    SPI_Init();
}

```

```

uint8_t Flash_ReadStatus(void)
{
    uint8_t status;
    FLASH_CS_LOW();
    SPI_ExchangeByte(FLASH_ReadStatusReg);
    status = SPI_ExchangeByte(0xFF);
    FLASH_CS_HIGH();
    return status;
}

void Flash_WriteEnable(void)
{
    FLASH_CS_LOW();
    SPI_ExchangeByte(FLASH_WriteEnable);
    FLASH_CS_HIGH();
}

void Flash_WaitBusy(void)
{
    while ((Flash_ReadStatus() & 0x01) == 0x01);
}

void Flash_ReadData(uint8_t* buffer, uint32_t address, uint16_t length)
{
    uint16_t i;
    FLASH_CS_LOW();
    SPI_ExchangeByte(FLASH_ReadData);
    SPI_ExchangeByte((uint8_t)(address >> 16));
    SPI_ExchangeByte((uint8_t)(address >> 8));
    SPI_ExchangeByte((uint8_t)address);
    for (i = 0; i < length; i++)
    {
        buffer[i] = SPI_ExchangeByte(0xFF);
    }
    FLASH_CS_HIGH();
}

void Flash_WritePage(uint8_t* data, uint32_t address, uint16_t length)
{
    uint16_t i;
    Flash_WriteEnable();
    FLASH_CS_LOW();
    SPI_ExchangeByte(FLASH_PageProgram);

```

```

    SPI_ExchangeByte((uint8_t)(address >> 16));
    SPI_ExchangeByte((uint8_t)(address >> 8));
    SPI_ExchangeByte((uint8_t)address);
    for (i = 0; i < length; i++)
    {
        SPI_ExchangeByte(data[i]);
    }
    FLASH_CS_HIGH();
    Flash_WaitBusy();
}

void Flash_EraseChip(void)
{
    Flash_WriteEnable();
    Flash_WaitBusy();
    FLASH_CS_LOW();
    SPI_ExchangeByte(FLASH_ChipErase);
    FLASH_CS_HIGH();
    Flash_WaitBusy();
}

/*****I2C温度传感器函数*****/
void I2C_Init(void)
{
    // 使能I2C
    LPC_SYSCON->PRESETCTRL |= (1 << 1);
    LPC_SYSCON->SYSAHBCLKCTRL |= (1 << 5);

    // 配置I2C引脚
    LPC_SYSCON->SYSAHBCLKCTRL |= (1 << 16);
    LPC_IOCON->PIO0_4 = 0x01; // SDA
    LPC_IOCON->PIO0_5 = 0x01; // SCL
    LPC_SYSCON->SYSAHBCLKCTRL &= ~(1 << 16);

    // 100kHz
    LPC_I2C->SCLH = 250;
    LPC_I2C->SCLL = 250;

    LPC_I2C->CONCLR = 0xFF;
    LPC_I2C->CONSET |= (1 << 6); // 使能I2C
}

void I2C_Start(void)
{
    LPC_I2C->CONSET |= (1 << 5);
    while (!(LPC_I2C->CONSET & (1 << 3)));
}

```

```

    LPC_I2C->CONCLR = (1 << 5) | (1 << 3);
}

void I2C_Stop(void)
{
    LPC_I2C->CONCLR = (1 << 3);
    LPC_I2C->CONSET |= (1 << 4);
    while (LPC_I2C->CONSET & (1 << 4));
}

void I2C_SendByte(uint8_t data)
{
    uint16_t timeout = 20000;
    LPC_I2C->DAT = data;
    LPC_I2C->CONCLR = (1 << 3);
    while ((!(LPC_I2C->CONSET & (1 << 3))) && (timeout--));
}

uint8_t I2C_ReceiveByte(void)
{
    uint8_t data;
    uint16_t timeout = 20000;
    LPC_I2C->CONSET = (1 << 2);
    LPC_I2C->CONCLR = (1 << 3);
    while ((!(LPC_I2C->CONSET & (1 << 3))) && (timeout--));
    data = (uint8_t)LPC_I2C->DAT;
    return data;
}

// 读取原始温度数据（16位）
int16_t ReadRawTemperature(void)
{
    uint8_t high_byte, low_byte;
    int16_t raw_temp;

    I2C_Start();
    I2C_SendByte(0x91); // LM75地址 + 读
    high_byte = I2C_ReceiveByte();
    low_byte = I2C_ReceiveByte();
    I2C_Stop();

    raw_temp = (high_byte << 8) | low_byte;
    return raw_temp;
}

```

```

// 原始数据转摄氏度
float ConvertToCelsius(int16_t raw_temp)
{
    int16_t temp_data = raw_temp >> 5;

    if (temp_data & 0x0400) {
        // 负数处理
        temp_data = -(~(temp_data & 0x03FF) + 1);
    }

    return temp_data * 0.125f;
}

/*****温度记录功能*****/
// 保存原始温度数据到Flash
void SaveTemperatureData(int16_t raw_temp)
{
    uint8_t temp_data[2];

    // 将16位温度数据拆分为2个字节
    temp_data[0] = (raw_temp >> 8) & 0xFF; // 高字节
    temp_data[1] = raw_temp & 0xFF;        // 低字节

    Flash_WritePage(temp_data, flash_address, 2);

    flash_address += 2;
    record_count++;

    // 防止地址溢出
    if (flash_address >= 0x1FFFE)
    {
        flash_address = 0;
        UART_SendString("Flash address reset to 0\r\n");
    }
}

// 读取并显示所有温度记录
void DisplayAllRecords(void)
{
    uint8_t temp_data[2];
    int16_t raw_temp;
    float temp_c;
    uint32_t address = 0;

```

```

uint16_t i;

UART_SendString("\r\n=== Temperature Records ===\r\n");

for (i = 0; i < record_count; i++)
{
    Flash_ReadData(temp_data, address, 2);

    // 组合16位数据
    raw_temp = (temp_data[0] << 8) | temp_data[1];
    temp_c = ConvertToCelsius(raw_temp);

    // 显示记录
    UART_SendString("Record ");
    SendNumber(i);
    UART_SendString(": ");
    SendFloat(temp_c, 3);
    UART_SendString(" C\r\n");

    address += 2;
}

UART_SendString("Total records: ");
SendNumber(record_count);
UART_SendString("\r\n");
}

/*****中断服务函数*****/
// UART中断服务函数
void UART_IRQHandler(void)
{
    if (LPC_UART->LSR & (1 << 0)) { // 接收中断
        uart_command = LPC_UART->RBR;
        ProcessUARTCommand();
    }
}

// 处理UART命令
void ProcessUARTCommand(void)
{
    switch (uart_command) {
        case 'r':
        case 'R':
            DisplayAllRecords();
    }
}

```

```

        break;
    case 'c':
    case 'C':
        Flash_EraseChip();
        flash_address = 0;
        record_count = 0;
        UART_SendString("Flash erased and reset\r\n");
        break;
    default:
        UART_SendString("Unknown command. Use: r=read records, c=clear flash\r\n");
        break;
}
}

// ADC中断服务函数
void ADC_IRQHandler(void)
{
    // 读取AD7通道的转换结果
    if (LPC_ADC->DR[7] & (1UL << 31)) {
        adc_value = (LPC_ADC->DR[7] >> 6) & 0x3FF;
        adc_data_ready = 1;
    }
}

// 定时器32_0中断服务函数 - 用于触发ADC转换
void TIMER32_0_IRQHandler(void)
{
    if ((LPC_TMR32B0->IR & 0x01) == 0x01) {
        LPC_TMR32B0->IR = 0x01; // 清除中断标志

        // 启动ADC转换（软件触发）
        LPC_ADC->CR |= (1UL << 24);
    }
}

// 定时器32_1中断服务函数 - 用于读取LM75温度并存储到Flash
void TIMER32_1_IRQHandler(void)
{
    if ((LPC_TMR32B1->IR & 0x01) == 0x01) {
        LPC_TMR32B1->IR = 0x01; // 清除中断标志

        // 读取LM75温度
        int16_t raw_temp = ReadRawTemperature();
        // 存储到Flash

```

```

        SaveTemperatureData(raw_temp);
        last_raw_temp = raw_temp;
        // float current_temp = ConvertToCelsius(raw_temp);

        flash_data_ready = 1;

    }
}

/*****主函数*****/
int main(void)
{
    // 系统初始化
    SystemCoreClockUpdate();
    // 外设初始化
    UART_Init();
    ADC_Init();
    TIMER32B0_Init(); // 用于ADC触发
    TIMER32B1_Init(); // 用于LM75温度采集和存储
    Flash_Init();
    I2C_Init();

    UART_SendString("LPC1114 Temperature Monitoring System Started\r\n");
    UART_SendString("Mode 1: Timer1 triggers ADC conversion every 1s\r\n");
    UART_SendString("Mode 2: Timer2 reads LM75 and saves to Flash every 1s\r\n");
    UART_SendString("Commands: r=read records, c=clear flash\r\n\r\n");

    // 擦除Flash并初始化
    Flash_EraseChip();
    flash_address = 0;
    record_count = 0;
    UART_SendString("Flash initialized and ready\r\n");

    while (1) {
        // 处理ADC数据
        if (adc_data_ready && flash_data_ready) {
            // ADC温度计算和显示
            UART_SendString("ADC Raw: ");
            SendNumber(adc_value);
            UART_SendString(" -> NTC Temp: ");
            float ntc_temperature = ADC_To_Temperature(adc_value);

```

```

        SendFloat(ntc_temperature, 2);
        UART_SendString(" C | ");

        // LM75温度计算和显示
        float lm75_temperature = ConvertToCelsius(last_raw_temp);
        UART_SendString("LM75 Temp: ");
        if (lm75_temperature >= 0) UART_SendString("+");
        SendFloat(lm75_temperature, 3);
        UART_SendString(" C");

        // 显示记录计数
        UART_SendString(" [Record #");
        SendNumber(record_count);
        UART_SendString("]\r\n");

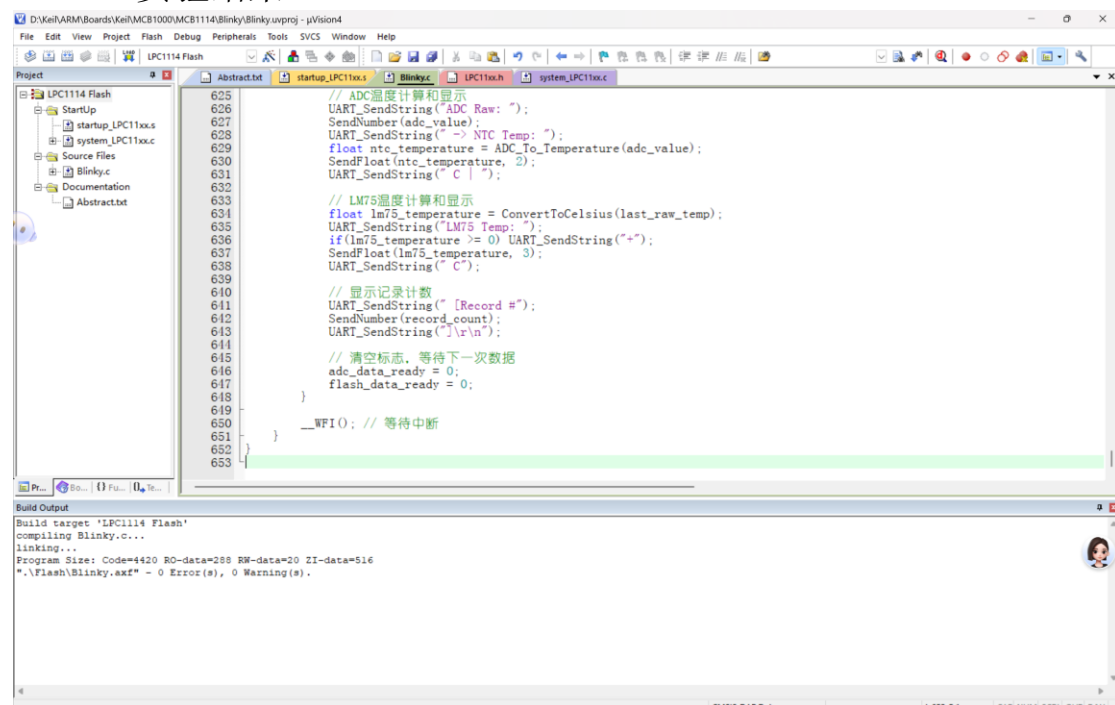
        // 清空标志，等待下一次数据
        adc_data_ready = 0;
        flash_data_ready = 0;
    }

    __WFI(); // 等待中断
}
}
}

```

七、实验结果与问题分析

1. 实验结果





程序成功编译，并且成功运行，功能实现完全且良好，实现 1s 一次温度测量（同时使用 LM75BD 和 NTC 热敏电阻，无冲突），并且可以正确保存数据到 flash 中，可以通过命令读取所有保存的数据，记录拥有单独的 Record 标志戳，正常。完美实现实验内容，并且实现功能改进和优化。

2. 问题分析

本次实验最大的问题是 NTC 热敏电阻和 LM75BD 的冲突问题，一开始由于主函数使用两个 if 语句分别判断数据就绪，并且在各自 if 块中调用 UART，会出现数据不能完全发送的问题，或者 ADC 和 NTC 某方的数据被阻断，后来经过三次迭代，引入两个标志位，并且等标志位同时就绪再转换和发送，同时将转换函数改为主函数内调用，解决了该问题。