

**实 验 报 告**

嵌入式系统设计

课程名称：

雷灿曦

姓 名：

20233006006

学 号：

2023级电子信息

年级专业：

张永辉 徐博

任课教师：

分 数：

海南大学 信息与通信工程学院

温度计设计

1. 实验目的
2. 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；
3. 学习使用嵌入式系统实验平台。学习LPC1114系列I2C、SPI、ADC等功能
4. 实验平台

硬件平台：LPC1114单片机

软件平台：Keil uVision4

1. 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。

1. 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：

PIO1\_6：RXD（数据接收引脚）

PIO1\_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：

PIO1\_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：

PIO2\_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

PIO2\_1：SCK（SPI时钟信号）

配置：通过IOCON寄存器设置为SCK功能

PIO2\_2：MISO（主入从出数据线）

配置：通过IOCON寄存器设置为MISO功能

PIO2\_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：

PIO0\_4：SDA（I2C数据线）

配置：通过IOCON寄存器设置为SDA功能

PIO0\_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PIO1\_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：

实际波特率 = UART\_PCLK / (16 × DLL × FR)，其中FR = MULVAL / (MULVAL + DIVADDVAL)

具体配置：

UART时钟 = 48MHz / 4 = 12MHz

DLL = 4，DLM = 0

DIVADDVAL = 5，MULVAL = 8，FR = 8/13 ≈ 0.615

实际波特率 = 12MHz / (16 × 4 × 0.615) ≈ 115384，

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 >> 5) × 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：

电压值 = (ADC值 × 3.3V) / 1023

NTC电阻值 = (3.3V - 电压值) × 10kΩ / 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

温度值 = (原始数据 >> 5) × 0.125°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。

1. 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

|  |  |
| --- | --- |
|  |  |
| UART初始化 | ADC初始化 |
|  |  |
| 定时器初始化 | Flash初始化 |
|  |  |
| SPI通信初始化 | I2C初始化 |

中断流程图

|  |
| --- |
|  |
| 程序中断流程图 |

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。

|  |
| --- |
|  |
| LM75B温度采集主流程 |

|  |  |
| --- | --- |
|  |  |
| LM75B温度读取流程 | 温度转换函数 |
|  |  |
| 温度存储流程 | 温度显示流程 |

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 |

|  |
| --- |
|  |
| NTC热敏电阻温度测量流程 |
|  |
| NTC热敏电阻线性插值温度计算流程 |

1. 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

}

}

1. 实验结果与问题分析
2. 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。

1. 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。