编号：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 总评 | 教师签名 |
| 成绩 |  |  |  |  |  |  |  |  |  |  |

武汉大学国家网络安全学院

课程实验(设计)报告

题 目： 嵌入式系统综合实验

专业(班)： 信息安全

学 号： 2021302181156

姓 名： 赵伯俣

课程名称： 嵌入式系统实践

任课教师： 丁玉龙

2024年6月4日

目录

[课程实验(设计)报告 1](#_Toc168952341)

[一、功能描述 4](#_Toc168952342)

[二、设计方案 6](#_Toc168952343)

[1.实验环境 6](#_Toc168952344)

[2.硬件 6](#_Toc168952345)

[3.功能设计 6](#_Toc168952346)

[4.安全设计 6](#_Toc168952347)

[三、安全设计描述 8](#_Toc168952348)

[1.抗干扰设计 8](#_Toc168952349)

[1.1算术中值滤波 8](#_Toc168952350)

[1.2冷热启动区分与初始化 8](#_Toc168952351)

[1.3初始化 11](#_Toc168952352)

[1.4看门狗及运行监视程序 11](#_Toc168952353)

[1.5配置刷新 12](#_Toc168952354)

[1.6睡眠躲避干扰 13](#_Toc168952355)

[2.容错设计 13](#_Toc168952356)

[2.1数据校验与修复 13](#_Toc168952357)

[2.2数据备份与距离 14](#_Toc168952358)

[2.3系统恢复（复位检查） 14](#_Toc168952359)

[2.4硬件故障自检 15](#_Toc168952360)

[2.5用户输入容错设计 15](#_Toc168952361)

[3.抗攻击设计 16](#_Toc168952362)

[3.1随机顺序执行 16](#_Toc168952363)

[3.2随机延迟启动： 17](#_Toc168952364)

[3.3前序代码执行检查 17](#_Toc168952365)

[3.4加密算法实现 18](#_Toc168952366)

[3.5数据冗余 19](#_Toc168952367)

[四、软件设计 20](#_Toc168952368)

[1.功能设计 20](#_Toc168952369)

[2.程序流程图 24](#_Toc168952370)

[3.实现结果 25](#_Toc168952371)

[五、实验问题与解决 27](#_Toc168952372)

[六、空间优化 29](#_Toc168952373)

[1.实验目的 29](#_Toc168952374)

[2.实验源码分析 29](#_Toc168952375)

[2.1 定义变量 29](#_Toc168952376)

[2.2 main函数 30](#_Toc168952377)

[2.3 switch\_key 31](#_Toc168952378)

[2.4 switch\_flag 32](#_Toc168952379)

[3.空间优化 33](#_Toc168952380)

[3.1 合并switch\_flag和switch\_key函数 33](#_Toc168952381)

[3.2 删掉flag变量 33](#_Toc168952382)

[3.3 删掉printf函数 33](#_Toc168952383)

[3.4优化后程序源代码 33](#_Toc168952384)

[4.优化结果 35](#_Toc168952385)

[5.空间优化方法总结 36](#_Toc168952386)

[七、设计总结 39](#_Toc168952387)

# 一、功能描述

本次实验设计了一个火灾自动报警灭火系统，因为在实验过程中温度传感器的芯片裸漏在外，不方便进行温度的改变，所以将火灾报警器的温度探测模块转移到了按键数码管中，由此实现的功能是用户通过按下按键数码管上的按键输入当前的温度数值，然后按下‘\*’号键将输入的温度提交判断，如果输入的温度超过设置的温度域值则将触发蜂鸣器报警以及步进电机旋转打开水阀。

（1）在用户操作时可以

* 按下数字按键0-9来输入当前环境的温度值；
* 按下‘\*’将输入的温度值提交检测；
* 按下‘#’可以将当前保存的温度值清空重新输入。

（2）整个系统的运行逻辑为：

* 不断检测用户输入，如果没有用户输入则进入睡眠。
* 出现用户输入后读取输入进行转码等处理方式获取当前温度值。
* 如果温度值超过设置的域值则蜂鸣器报警，步进电机转动
* 如果温度值没有超过设置的域值则输出信息并且重新进入等待状态

（3）在抗干扰设计上我们实现了：

* 算术中值滤波
* 上电复位延时处理
* 看门狗及运行监视程序
* 冷热启动区分与初始化
* 数据单元结构体：数据+校验码
* 超时处理
* 定时配置刷新

（4）在容错设计上我们实现了：

* 用户输入的容错设计
* 数据校验与修复
* 数据备份与距离
* 系统恢复（复位检查）
* 硬件故障自检

（5）在抗攻击设计上我们实现了：

* 数据冗余（数据单元+多备份、等效多变量实现存储位置随机）
* 控制冗余：入口参数检查与前序代码执行检查
* 执行冗余：随机顺序执行
* 特定加密算法的安全实现
* 前序代码执行检查

# 二、设计方案

## 1.实验环境

FS-STM32F407开发平台

ST-Link仿真器

RealView MDK5.21 集成开发软件

PC 机 XP、Window7/8 (32/64bit)

串口调试助手

## 2.硬件

按键与数码管

步进电机

蜂鸣器

## 3.功能设计

整个火灾自动报警灭火系统依靠温度判断部分、报警灭火两部分组成，在温度判断部分中使用开放版上的按键输入来读取用户输入作为当前的温度进行判断，并且将用户输入的温度显示在数码管中；在报警灭火部分中，如果用户输入的温度超过了提前设置的温度域值则触发步进电机打开水阀，蜂鸣器持续报警，如果用户输入的温度抵低于设置的温度域值则蜂鸣器和步进电机不进行工作并且输出提示信息告知判断结果。

当下次用户输入时则终止本次工作状态进而跳转到协议工作状态进行循环；同时在系统处于等待用户输入状态时如果等待时间超过了设置的休眠域值则整个系统会进行休眠进而躲避干扰。

## 4.安全设计

**在抗干扰层面的设计为：**

（1）输入滤波：过滤输入信号，去除噪声或不相关的数据，确保输入数据的纯净和可用性。

（2）冷启动上电延迟：当系统冷启动时延迟一定时间，保护设备免受启动时可能出现的电气冲击和损坏，同时确保系统能够在适当的操作条件下稳定启动。

（3）热启动数据校验：确保在重启过程中，所有的数据和系统状态都能保持正确和完整。

（4）看门狗防止死循环：监控系统运行状态，防止系统陷入死循环或无响应状态，通过重置系统来恢复正常运行。

（5）配置刷新：维护系统安全、效率和兼容性

（6）睡眠躲避干扰：在系统进入睡眠模式时减少外部干扰，保护系统不受未授权访问或干扰。

**在容错层面的设计为：**

（1）数据校验与修复：检查数据完整性和一致性，必要时进行修复，以保证数据的准确性和完整性。

（2）系统恢复（复位检查）：确保设备能在遇到问题时最小化数据损失并快速恢复正常运行。

（3）数据备份与距离：定期备份数据，并存储在物理上分离的位置，以防单点故障影响所有数据。

（4）硬件故障自检：系统自检硬件组件的功能性和状态，及时发现并处理故障。

（5）用户输入容错设计：设计系统以能够处理用户输入中的错误或异常，确保系统稳定性和数据准确性。

**在抗攻击层面的设计为：**

（1）随机顺序执行：通过改变程序执行的顺序来减少预测性和避免特定的干扰或攻击模式。

（2）随机延迟启动：增加攻击者利用时间窗口进行攻击的难度。

（3）前序代码执行检查：在执行代码前检查其安全性和完整性，防止恶意代码执行。

（4）加密算法实现：在内存中保存的数据均经过加密处理，防止被攻击者窃取。

# 三、安全设计描述

## 1.抗干扰设计

### 1.1算术中值滤波

因为用户输入的温度在传输过程中可能发生错误从而导致我们的系统错误运行，因此我们对输入数据进行算数中值滤波，在读取温度时读取两次数码管中的温度值，在对读取两次的温度求算术平均数，将求得的结果作为最终的温度值。

        uint8\_t temp1 = ReadNumber();               //读取两次取个平均值

       uint8\_t temp2 = ReadNumber();

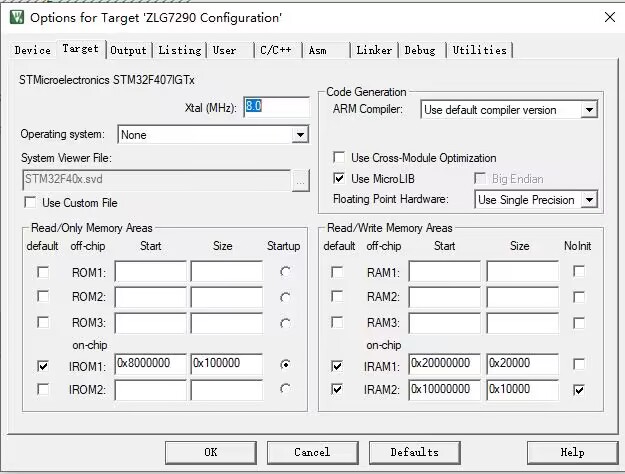
       uint8\_t ave = (temp1 + temp2)/2;

       //printf("--------------%d\r",ave);

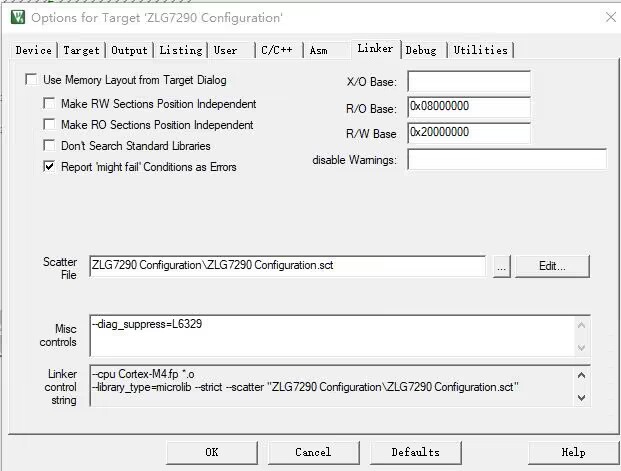
       TempUpdate(ave);

### 1.2冷热启动区分与初始化

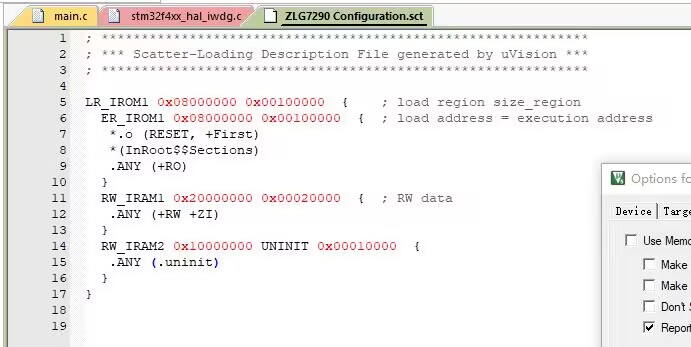
首先为了区分冷热启动，我们需要在编译器中修改芯片的配置，开启IRAM2芯片，并且将其No Init选项勾选将IRAM2设置为不可初始化，使其在每次reset不会被初始化，计划使用IRAM1保存程序代码，使用IRAM2保存程序关键数据



然后进入到设置的Linker模块中，点击Edit按钮编辑配置文件ZLG7290 Configuration.sct



将IRAM2内存块命名为.uninit



然后将关键变量设置为保存到.uninit区域中，并且设置一个HotStart变量用来标识程序是否为热启动

//该区域全局变量都会被0初始化

\_\_attribute\_\_((section(".uninit"),zero\_init)) Data Temperature;

//全局温度变量

\_\_attribute\_\_((section(".uninit"),zero\_init)) uint16\_t HotStart;

//是否热启动

\_\_attribute\_\_((section(".uninit"),zero\_init)) uint32\_t DelayTime;

//延迟启动时间

\_\_attribute\_\_((section(".uninit"),zero\_init)) uint8\_t InputLen;

//用于限制输入个数在两位数之内

\_\_attribute\_\_((section(".uninit"),zero\_init)) uint32\_t LastTime;

//用于超时处理，记录上一次get时间

\_\_attribute\_\_((section(".uninit"),zero\_init)) uint8\_t id;

//函数执行顺序

\_\_attribute\_\_((section(".uninit"),zero\_init)) uint8\_t first\_num;

//读取的第一位数字

\_\_attribute\_\_((section(".uninit"),zero\_init)) uint8\_t second\_num;

//读取的第二位数字

\_\_attribute\_\_((section(".uninit"),zero\_init)) uint8\_t num\_no;

//读取的数字序号

在程序的主函数中对HotStart变量进行判断，当冷启动时将其值设置为0x4514（该值为随机设置，需要确保0、1共存）即如果HotStart中的值为0x4514则视为冷启动，反之视为热启动

//冷热启动

    //printf("HotStart=%d\r\n",HotStart);

    if(HotStart != 0x4514){//冷启动初始化

        //启动与开始之间的休眠时间      delay(80~100)   //done

        DelayTime = rand() % 21 + 80;

        HAL\_Delay(DelayTime);

        Init();

        printf("Cold Start!!!!HotStart=%d\r\n",HotStart);

        HotStart = 0x4514;

        TempUpdate(25);

    }else{//热启动

        printf("\r\nhotStart,HoyStart=%d\r\n",HotStart);

        MX\_GPIO\_Init();

        int yes = CheckData();

        if(yes == 1){

            TempUpdate(Temperature.Odata - 1);

        }else{

            FaultDeal();

        }

}

### 1.3初始化

在初始化时首先初始化硬件抽象层（HAL）。确保微控制器的硬件在被任何设备特定的初始化代码使用前处于一个已知的状态。然后配置系统时钟。系统时钟决定了微控制器的运行频率，包括CPU时钟和外设时钟。之后初始化通用输入输出端口（GPIO）。之后初始化第一个I2C接口。允许微控制器与I2C兼容的设备（温度传感器、EEPROM、LCD显示器等）进行通信。之后初始化第一个串行通信接口（USART1）。

void Init(void){

    HAL\_Init();

    SystemClock\_Config();

    MX\_GPIO\_Init();

    MX\_I2C1\_Init();

    MX\_USART1\_UART\_Init();

}

### 1.4看门狗及运行监视程序

首先需要声明看门狗

IWDG\_HandleTypeDef \*hiwdg;

然后配置看门狗的具体信息并且调用库函数对看门狗进行初始化。

设置看门狗的实例，指定使用的是独立看门狗。

然后设置看门狗的预分频值。预分频器定义了系统时钟与看门狗时钟之间的分频比例，这里设置为4。这意味着看门狗计时器的时钟频率是系统时钟频率的1/4。

之后设置看门狗的重载值。这是看门狗计数器的初始值，每次“喂狗”操作都会将计数器重置到这个值。随着时间的推移，看门狗计数器递减，直到达到0。如果计数器达到0而没有被重置，看门狗硬件将触发系统复位。0x0ffff是一个较大的值，意味着在看门狗复位之前会有较长的时间窗口。

最后调用硬件抽象层（HAL）的函数来初始化看门狗。这个函数负责将前面设置的参数（实例、预分频值和重载值）应用到硬件中，并启动看门狗。一旦启动，除非系统重置，否则看门狗将持续运行。

void IWDG\_Init(void)

{

  hiwdg->Instance = IWDG;

  hiwdg->Init.Prescaler = IWDG\_PRESCALER\_4;

  hiwdg->Init.Reload = 0x0ffff;

  HAL\_IWDG\_Init(hiwdg);

}

然后在每一次检测到用户输入之后都进行喂狗操作

if(flag1 == 1){       //证明有按键进来，所以超时判断不应该在这里

            flag1 = 0;

            //喂狗

            \_\_HAL\_IWDG\_RELOAD\_COUNTER(hiwdg);

            //读输入板

            I2C\_ZLG7290\_Read(&hi2c1,0x71,0x01,Rx1\_Buffer,1);

            switch\_key();

            Rx2\_Buffer[1]=first\_num;

            //I2C\_ZLG7290\_Read(&hi2c1,0x71,0x10,Rx2\_Buffer,8);

            //来输入了，记录时间

            //LastTime = HAL\_GetTick();

            //输入检查

            switch\_flag();

            I2C\_ZLG7290\_Write(&hi2c1,0x70,ZLG\_WRITE\_ADDRESS1+2,Tx1\_Buffer+2,6);

            HAL\_Delay(20);

        }

### 1.5配置刷新

为了防止在程序运行的过程中硬件出现故障，所以在热启动时调用MX\_GPIO\_Init函数对硬件引脚进行初始化，实现在每一次热启动时都能够初始化硬件引脚，从而避免因为引脚错误而导致的程序错误

else{//热启动

        printf("\r\nhotStart,HoyStart=%d\r\n",HotStart);

        MX\_GPIO\_Init();

        int yes = CheckData();

        if(yes == 1){

            TempUpdate(Temperature.Odata - 1);

        }else{

            FaultDeal();

        }

}

### 1.6睡眠躲避干扰

在系统没有接受到用户输入时则进入睡眠模式，从而躲避来自外界的干扰

else{        //睡眠躲避干扰

            printf("HAL\_GetTICK:----->%d\r\n",HAL\_GetTick());

            FlushBoard();

            EnterSleep();

睡眠函数内容如下使 STM32 微控制器进入低功耗模式

void EnterSleep(void){

    HAL\_PWR\_EnterSLEEPMode(PWR\_MAINREGULATOR\_ON, PWR\_SLEEPENTRY\_WFI);

}

## 2.容错设计

### 2.1数据校验与修复

为了防止温度等重要数据在传输的过程中发成错误，在这里将数据的低8位作为数据的校验和进行保存，如果数据的校验和不正确，则认为数据是错误数据。

在进行数据修复时首先判断原本的数据校验和是否正确，如果错误则验证备份1的校验和，如果备份1的校验和正确则将备份1赋值给数据变量并更改数据变量的校验和。

如果备份1的校验和也错误，则启用备份2，如果备份2校验和正确，则将备份2赋值给数据变量，更新数据变量的校验和；将备份2赋值给备份1，更新备份1的校验和。

如果数据变量和两个备份的校验和都发生错误，则返回0，反之返回1

uint8\_t CheckData(void){

    if((Temperature.Odata ^ 0xff) != Temperature.checksum){

        if((Temperature.back1 ^ 0xff) == Temperature.checksum){

            Temperature.Odata = Temperature.back1;

            Temperature.checksum = Temperature.Odata ^ 0xff;

        }else{

            if((Temperature.back2 ^ 0xff) == Temperature.checksum2){

                Temperature.Odata = Temperature.back2;

                Temperature.checksum = Temperature.Odata ^ 0xff;

                Temperature.back1 = Temperature.back2;

                Temperature.checksum1 = Temperature.back1 ^ 0xff;

            }else{

                return 0;

            }

        }

    }

    return 1;

}

### 2.2数据备份与距离

因为数据在传输的过程中会因为丢失等原因造成数据错误，所以需要对关键数据进行备份。保存数据备份的数据结构如下所示，同时因为数据损坏时往往是成片损坏，所以在数据结构中定义了blank1和blank2两个空内存块来实现内存数据备份时分开距离备份

typedef struct ValueAndBack{//数据及备份，检验和采用与0xff异或

    uint8\_t Odata;

    uint8\_t checksum;

    uint32\_t blank1;

    uint8\_t back1;

    uint8\_t checksum1;

    uint32\_t blank2;

    uint8\_t back2;

    uint8\_t checksum2;

}Data;

### 2.3系统恢复（复位检查）

在热启动对系统进行恢复时首先将硬件的引脚重新初始化，防止长期工作使得引脚发生错误；然后需要校验温度变量的正确性，入股温度变量的校验和正确则更新温度变量的数值，反之则进入错误处理模块

else{//热启动

        printf("\r\nhotStart,HoyStart=%d\r\n",HotStart);

        MX\_GPIO\_Init();

        int yes = CheckData();

        if(yes == 1){

            TempUpdate(Temperature.Odata - 1);

        }else{

            FaultDeal();

        }

}

### 2.4硬件故障自检

在系统的主函数中，为了防止硬件故障造成的系统错误，在判断是否有用户按键之前先将所有的硬件引脚重新初始化，从而规避掉因为硬件故障造成的程序错误

    while (1)

    {

        MX\_GPIO\_Init();

        //\_\_HAL\_IWDG\_RELOAD\_COUNTER(hiwdg);

        if(flag1 == 1){     //证明有按键进来，所以超时判断不应该在这里

            ...

        }

else{        //睡眠躲避干扰

....

}

}

### 2.5用户输入容错设计

在用户输入模块中，只读取数码管中的两位数据，限制用户的输入，从而防止用户输入过大的数值导致程序错误。

* 输入0位就回车，不会有任何功能
* 输入已经有2位，且还按0~9，则同上不做处理（也就不会显示）
* 其他情况下，进行显示数据或者清0，或回车

I2C\_ZLG7290\_Read(&hi2c1,0x71,0x01,Rx1\_Buffer,1);

            switch\_key();

            Rx2\_Buffer[1]=first\_num;

            //I2C\_ZLG7290\_Read(&hi2c1,0x71,0x10,Rx2\_Buffer,8);

            //来输入了，记录时间

            //LastTime = HAL\_GetTick();

            //输入检查

            switch\_flag();

## 3.抗攻击设计

### 3.1随机顺序执行

执行冗余是一种增加系统可靠性和容错性的技术，通过引入多个独立的冗余执行路径或等价实现方式来实现。这包括独立代码随机顺序执行、等价实现随机选择执行、延时和功耗插入，以及数据单元等价实现。这些技术的目标是通过多样性和随机性来减少潜在的错误或故障，并提高系统的稳定性和可靠性。执行冗余可以在设计和开发阶段实施，并可以通过软件或硬件方式实现。通过引入执行冗余，系统能够更好地适应不可预测的环境变化，提高系统的容错性，并增加系统的可靠性和性能。

具体而言我们在向数码管输出数据时采用随机乱序执行等价的代码，即设计随机种子，根据随机种子的值来选择要执行的等价的语句，各个等价的语句中添加了不同时间的HAL\_Delay来实现等价语句的差异性。通过多个实现之间的差异来减少潜在的错误或故障，并增加系统的可靠性。

同时这些差异性还实现了随机的延时和随机功耗的设计，这样可以有效地防止测信道攻击。

case 15://this is '\*' for enter number

rand\_func=rand()%3; //随机乱序执行躲避攻击

switch(rand\_func){

case 0:

HAL\_Delay(1);

EnterMain();

break;

case 1:

HAL\_Delay(2);

EnterMain();

break;

case 2:

HAL\_Delay(3);

EnterMain();

break;

}

### 3.2随机延迟启动：

在程序冷启动时为了防止侧信道攻击，设置了一个随机延迟，使得系统延迟随机一段时间之后再启动，同时也能够有随机功耗的作用

if(HotStart != 0x4514){//冷启动初始化

        //启动与开始之间的休眠时间      delay(80~100)   //done

        DelayTime = rand() % 21 + 80;

        HAL\_Delay(DelayTime);

        Init();

        printf("Cold Start!!!!HotStart=%d\r\n",HotStart);

        HotStart = 0x4514;

        TempUpdate(25);

    }

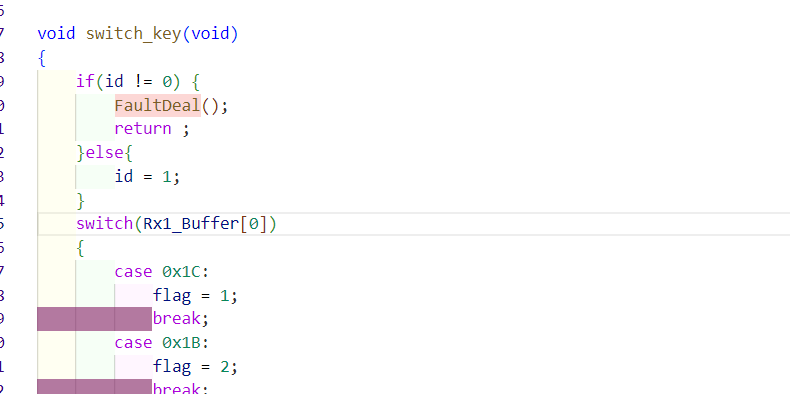
### 3.3前序代码执行检查

前序代码检查是确保代码连续性和正确性的过程。它包括检查上一条代码的返回值、处理异常情况、验证数据有效性以及检查相关依赖项。通过前序代码检查，可以确保每一条代码在执行之前，前一条代码已成功执行，并满足下一条代码的要求。这有助于提高代码的稳定性、可靠性和安全性，减少潜在的错误和故障的发生。

这里我们进行的简单实现是在解析用户按键和根据解析出的按键的编码进行写数码管这两处功能中对前序代码是否成功执行进行检查。即设置一个安全数据id，在初始化时将id置为0，在进入switch\_key()函数时，先检查id是否为0，当id等于0时将id置为1并且继续执行switch\_key()的功能代码；当id不等于0时调用错误处理函数FaultDeal()，在此函数中我们会处理错误，并将相关数据初始化，包括将id置为0，保证下一次正确读入数据。

而在进入switch\_flag()函数时，我们检查id是否为1，当id等于1时将id置为0并且继续执行switch\_flag()的功能代码；当id不等于1时，调用错误处理函数FaultDeal()。

这样可以保证switch\_flag的执行是在switch\_key执行成功的基础上的，并且保证了读入的按键的编码是在我们设置的范围中。





### 3.4加密算法实现

数据加密是一种重要的安全措施，它通过将数据转化为密文，以保护数据的机密性、完整性和可用性。加密防止未经授权的访问者获取敏感数据，防止数据泄露和窃取，确保数据在存储、传输和处理过程中的安全性。

我们的安全数据温度存储在安全数据区域，那么存储在此处的数据应该是密文而不是明文，在具体实现中，我们采用简易的加密方法，对原始数据进行加1处理，处理后的数据作为密文存储在内存中，并且对密文进行校验。那么在取出数据时要进行的解密操作就是对密文减1，即可得到明文。

这里采用加减一的加解密方法是因为简单，并且我们的安全数据的类型是无符号类型的数据，对其加1和减1不会造成数据溢出的危险。

void TempUpdate(uint8\_t temp){

Temperature.Odata = temp + 1;

Temperature.checksum = Temperature.Odata ^ 0xff;

Temperature.back1 = temp + 1;

Temperature.checksum1 = Temperature.back1 ^ 0xff;

Temperature.back2 = temp + 1;

Temperature.checksum2 = Temperature.back2 ^ 0xff;

}

### 3.5数据冗余

数据冗余的目的是提高数据的可靠性和可用性，防止数据丢失，并支持系统的高可用性和容错性。通过存储多个数据副本，数据冗余能够在某个副本出现故障或损坏时，仍然可以使用其他副本来恢复数据，保护数据的完整性。此外，数据冗余还能提高系统的性能和响应速度，通过就近获取数据副本减少数据访问的延迟。同时，数据冗余支持灾难恢复和备份策略，使系统能够在灾难性事件发生时快速恢复，并保障业务的连续性。综合而言，数据冗余旨在提供数据的可靠性、完整性、可用性和容错性，以应对各种风险和故障，并支持系统的高性能、高可用性和灾难恢复能力。

具体而言，我们利用结构体对原始数据进行了两次备份，并且在这三份数据之间设置了空白间隔来实现简易的异地存储。

typedef struct ValueAndBack{//数据及备份

uint8\_t Odata;

uint8\_t checksum;

uint32\_t blank1;

uint8\_t back1;

uint8\_t checksum1;

uint32\_t blank2;

uint8\_t back2;

uint8\_t checksum2;

}Data;

# 四、软件设计

## 1.功能设计

本次软件设计将数码管、舵机、蜂鸣器三个模块结合在一起，实现了通过按键输入当前温度，检查当前温度，在当前的温度高于系统设置的温度阈值时，启动蜂鸣器和舵机功能，即模拟火灾防御报警系统。

首先是输入当前温度的模块，这里我们设置的有效输入时1~9按键、\*按键、#按键，具体来说数字1~9按键是当前输入的最低位的温度的值，\*按键是确定当前输入的温度的值并将其发送给处理系统，#按键是清空当前所有输入的数字。具体的代码实现就是原始数码管项目中的switch\_key和switch\_flag函数，修改后的函数如下。

检测输入的按键的键值switch\_key

void switch\_key(void)

{

switch(Rx1\_Buffer[0])

{

case 0x1C:

flag = 1;

break;

case 0x1B:

flag = 2;

break;

case 0x1A:

flag = 3;

break;

case 0x14:

flag = 4;

break;

case 0x13:

flag = 5;

break;

case 0x12:

flag = 6;

break;

case 0x0C:

flag = 7;

break;

case 0x0B:

flag = 8;

break;

case 0x0A:

flag = 9;

break;

case 0x03:

flag = 0;

break;

case 0x02://# //'#'用于清空，'\*'用于回车

flag = 14;

break;

case 0x04://\*

flag = 15;

break;

default:

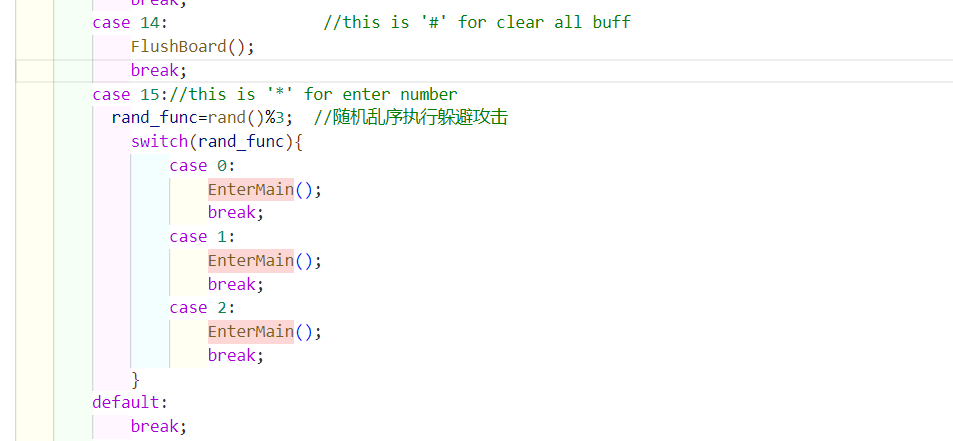
break;

}

}

根据键值flag来不断刷新数码管用来显示当前输入的具体数据，具体代码过长，这里不全部展示





并且在输入时我们设置了两位数据的的输入限制，那么在switch\_flag刷新完整个数码管模块后，我们将其他六位数据重新清零，即只显示当前的有效的两位数据，具体代码如下。

switch\_flag();

I2C\_ZLG7290\_Write(&hi2c1,0x70,ZLG\_WRITE\_ADDRESS1+2,Tx1\_Buffer+2,6);

HAL\_Delay(20);

这里在检测到用户输入\*按键时，调用EnterMain函数进行参数判断和输出，此函数读取两次的输入的数据，并对两次读入的数据进行中值滤波和校验数据的合法性，当解密后得到的当前温度大于设定的阈值40时，程序会启动蜂鸣器和舵机，即BeeWater函数，否则进行错误处理或者清空输入，等待下次的输入。

void EnterMain(void){

uint8\_t temp1 = ReadNumber(); //读取两次取个平均值

uint8\_t temp2 = ReadNumber();

uint8\_t ave = (temp1 + temp2)/2;

TempUpdate(ave);

uint8\_t yes = CheckData();

if(yes && Temperature.Odata - 1 > 40){

printf("Tempreture is ==>%d\r",Temperature.Odata - 1);

BeeWater();

}else if(!yes){

printf("check is error\r");

FaultDeal();//需要调用'#'功能并bee一下提示，清除并重新输入

}else{

printf("not so hot, I think there has no fire\r");

}

FlushBoard(); //输入0个数,超过2位数,回车之后，都会清空

}

启动蜂鸣器和舵机时是根据原始项目的启动方式设置相关引脚来实现的。

void BeeWater(void){

while(flag1 == 0){

//beep

HAL\_GPIO\_WritePin(GPIOG,GPIO\_PIN\_6,GPIO\_PIN\_SET);

HAL\_Delay(5);

HAL\_GPIO\_WritePin(GPIOG,GPIO\_PIN\_6,GPIO\_PIN\_RESET);

HAL\_Delay(5);

//step\_motor

DE\_A;

HAL\_Delay(3);

DE\_AB;

HAL\_Delay(3);

DE\_B;

HAL\_Delay(3);

DE\_BC;

HAL\_Delay(3);

DE\_C;

HAL\_Delay(3);

DE\_CD;

HAL\_Delay(3);

DE\_D;

HAL\_Delay(3);

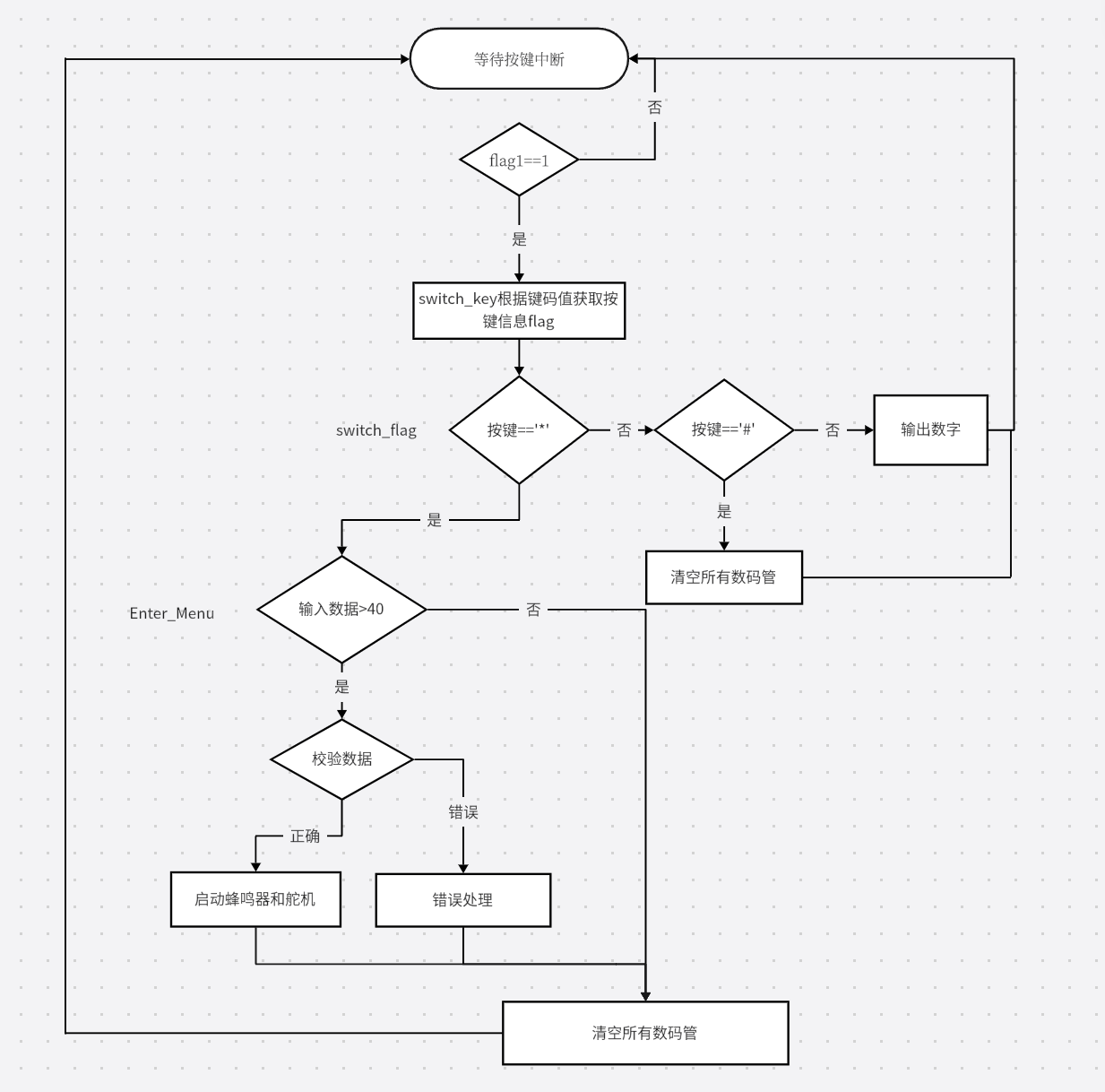
DE\_DA;

HAL\_Delay(3);

}

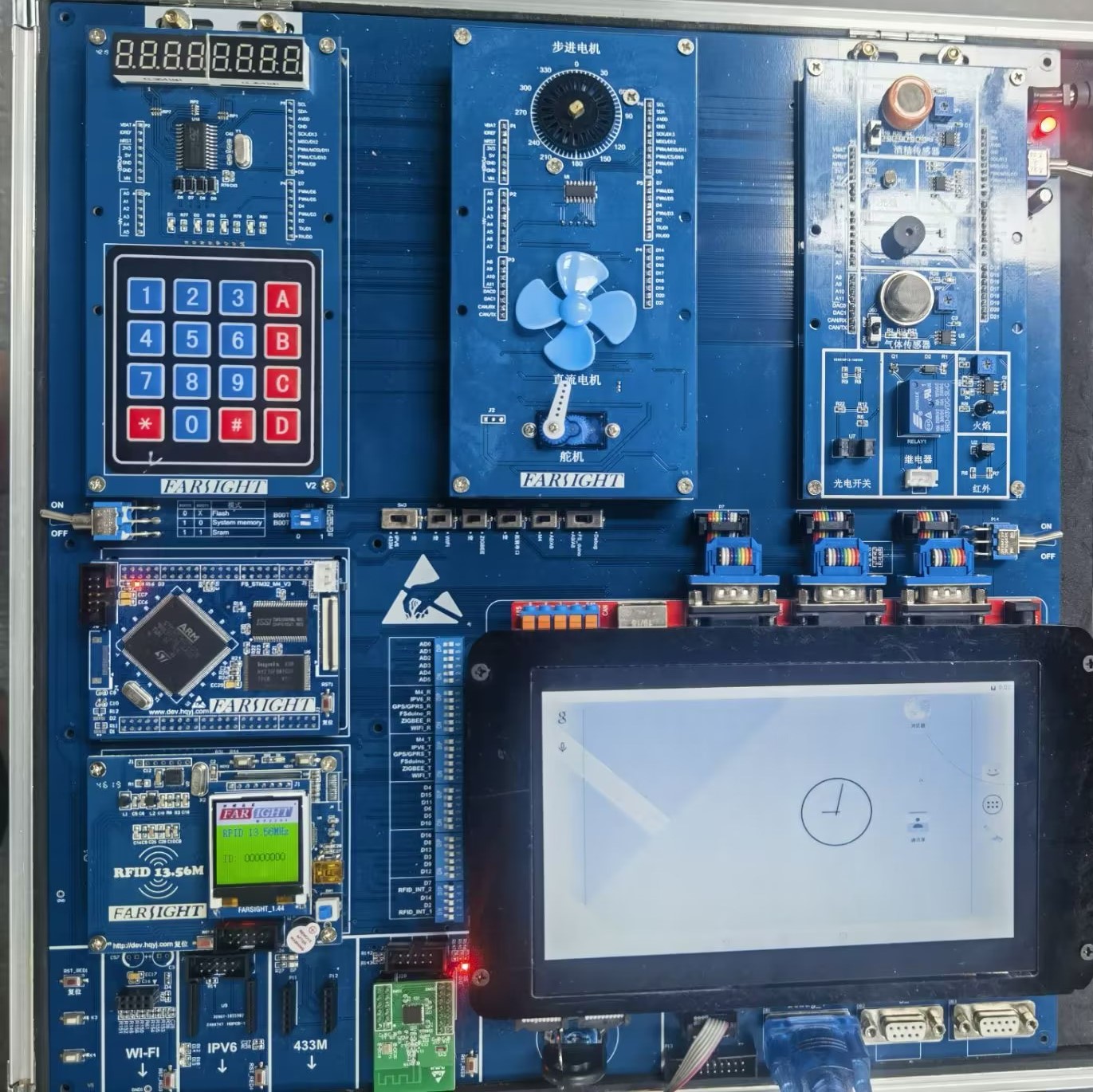
}

## 2.程序流程图

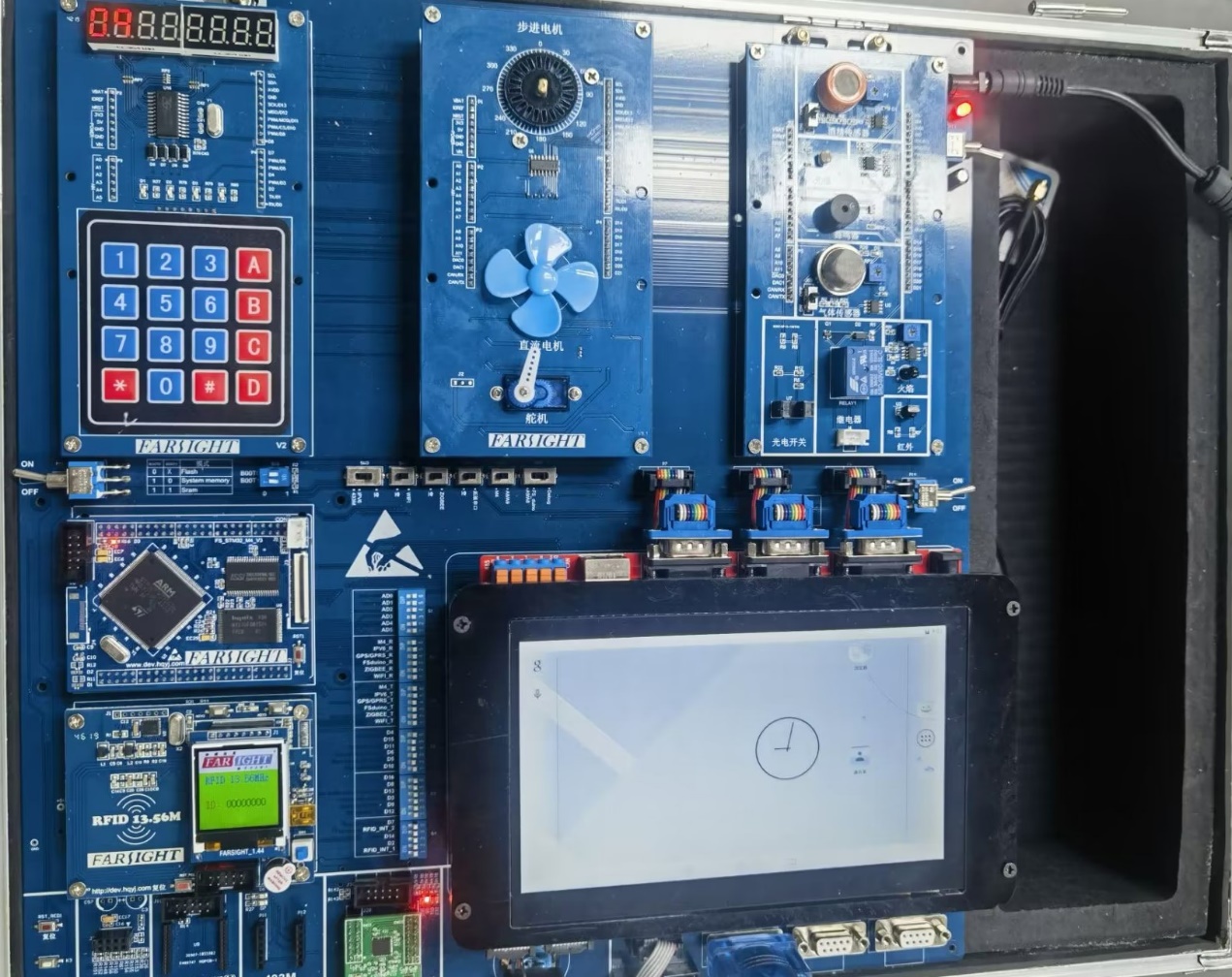


## 3.实现结果

初始状态（无输入）

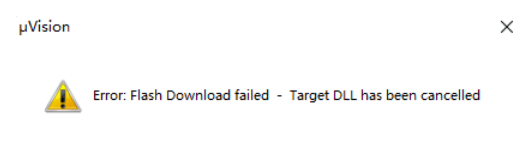


用户输入89作为当前温度，蜂鸣器和步进电机启动



# 五、实验问题与解决

1.在下载到开发板时下载错误



产生错误的原因是SWD的引脚被占用了，或者被禁用了。需要在下载时按住reset，再点击download，再松开reset。原理就是在程序运行到SWD引脚占用之前，把新的程序烧录进去。

2.睡眠躲避干扰

在添加睡眠躲避干扰时，我们查看延迟函数HAL\_Delay的定义如下，可以发现延迟函数只是空转了一定的时间，在while循环中并没有进行其他操作。

\_\_weak void HAL\_Delay(\_\_IO uint32\_t Delay)

{

  uint32\_t tickstart = 0;

  tickstart = HAL\_GetTick();

  while((HAL\_GetTick() - tickstart) < Delay)

  {

  }

}

因此，我们尝试修改HAL\_Delay函数的定义如下所示，将睡眠函数的核心功能加入到延迟函数中，使得程序在延迟过程中也可以进入睡眠状态从而躲避干扰

\_\_weak void HAL\_Delay(\_\_IO uint32\_t Delay)

{

  uint32\_t tickstart = 0;

  tickstart = HAL\_GetTick();

  while((HAL\_GetTick() - tickstart) < Delay)

  {

    //HAL\_SuspendTick();

    HAL\_PWR\_EnterSLEEPMode(PWR\_MAINREGULATOR\_ON, PWR\_SLEEPENTRY\_WFI);

    //HAL\_ResumeTick();

  }

}

3.冷启动上电延迟位置

在系统冷启动给程序上电时需要延迟一段时间等软硬件全部准备完毕，调用延迟的位置应该在冷启动初始化过程中。

4.看门狗时长

因为在编写程序初期设定的程序运行时常为5s，后期将程序的运行时间修改为1s，在修改程序运行时间之后需要同步修改看门狗的时常，将看门狗的时常修改为程序原本运行时间的120%。

# 六、空间优化

## 1.实验目的

在本次实验中选择对第9号实验：按键扫描数码管显示实验进行空间优化并总结在嵌入式系统编码过程中可以使用的空间优化方法

## 2.实验源码分析

### 2.1 定义变量

首先定义了ZLG\_READ\_ADDRESS1、ZLG\_READ\_ADDRESS2、ZLG\_WRITE\_ADDRESS1、ZLG\_WRITE\_ADDRESS2这些常量，意义是地址或标识符；BUFFER\_SIZE1、BUFFER\_SIZE2这两个常量定义了两个缓冲区的大小，它们分别是 Tx1\_Buffer 和 Rx2\_Buffer。

之后进行宏定义countof(a)用于计算数组的元素个数。接受一个数组作为参数，返回该数组的元素个数。

最后进行变量定义，flag用于标识不同的操作或状态；flag1用作检测标志位，每次进行一项操作后，它会被清零，准备获取8位数据的第一个字节；Rx2\_Buffer、Tx1\_Buffer、Rx1\_Buffer分别是用于存储数据的数组；Rx2\_Buffer 和 Tx1\_Buffer 的长度为8，而 Rx1\_Buffer 的长度为1。它们被初始化为全0。

#define ZLG\_READ\_ADDRESS1         0x01

#define ZLG\_READ\_ADDRESS2         0x10

#define ZLG\_WRITE\_ADDRESS1        0x10

#define ZLG\_WRITE\_ADDRESS2        0x11

#define BUFFER\_SIZE1              (countof(Tx1\_Buffer))

#define BUFFER\_SIZE2              (countof(Rx2\_Buffer))

#define countof(a) (sizeof(a) / sizeof(\*(a)))

uint8\_t flag;

uint8\_t flag1 = 0;

uint8\_t Rx2\_Buffer[8]={0};

uint8\_t Tx1\_Buffer[8]={0};

uint8\_t Rx1\_Buffer[1]={0};

### 2.2 main函数

(1)HAL\_Init()是HAL（Hardware Abstraction Layer）库的初始化函数，用于初始化STM32的HAL库，包括将系统时钟配置为默认时钟。

(2)SystemClock\_Config()用于配置系统时钟，具体设置了外部高速时钟（HSE），PLL倍频器以及AHB/APB时钟的分频器等参数，以确保MCU能够以预期的时钟频率运行。

(3)MX\_GPIO\_Init()、MX\_I2C1\_Init()、MX\_USART1\_UART\_Init()：这些函数是CubeMX代码生成器自动生成的，用于初始化GPIO、I2C1和USART1模块，以便后续的GPIO操作、I2C通信和串口通信。

(4)函数的主体是while(1) 无限循环，在 flag1 为1的情况下进行数据处理，否则会继续等待。

在数据处理部分，首先将 flag1 置为0，以便下一次循环时重新检测数据处理的条件。然后调用 I2C\_ZLG7290\_Read() 函数读取数据，这里假设是从某个I2C设备中读取数据。之后将读取的数据打印到终端。最后调用swtich\_key() 和 switch\_flag()）来处理读取的数据。

int main(void)

{

  HAL\_Init();

  SystemClock\_Config();

  MX\_GPIO\_Init();

  MX\_I2C1\_Init();

  MX\_USART1\_UART\_Init();

  printf("\n\r");

  printf("\n\r-------------------------------------------------\r\n");

  printf("\n\r FS-STM32������ ������оƬ��������\r\n");

  while (1)

  {

        if(flag1 == 1)

        {

            flag1 = 0;

            I2C\_ZLG7290\_Read(&hi2c1,0x71,0x01,Rx1\_Buffer,1);

            printf("\n\r������ֵ = %#x\r\n",Rx1\_Buffer[0]);

            swtich\_key();

            I2C\_ZLG7290\_Read(&hi2c1,0x71,0x10,Rx2\_Buffer,8);

            switch\_flag();

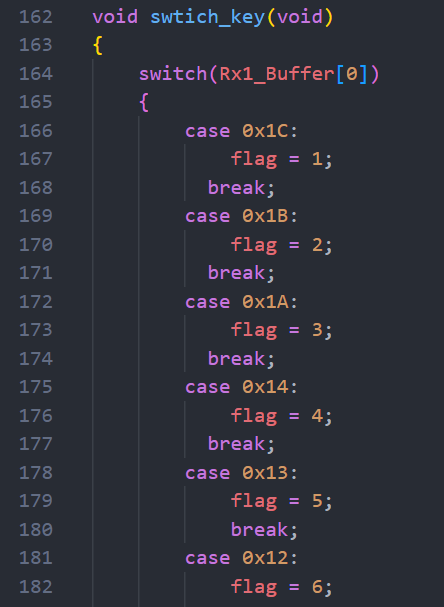
        }

  }

}

### 2.3 switch\_key

该函数根据Rx1\_Buffer[0]的值去修改flag的值



该函数所对应的变换关系如下表所示

|  |  |
| --- | --- |
| Rx1\_Buffer[0] | flag |
| 0x01 | 13 |
| 0x02 | 14 |
| 0x03 | 15 |
| 0x09 | 12 |
| 0x0A | 9 |
| 0x0B | 8 |
| 0x0C | 7 |
| 0x11 | 11 |
| 0x12 | 6 |
| 0x13 | 5 |
| 0x14 | 4 |
| 0x19 | 10 |
| 0x1A | 3 |
| 0x1B | 2 |
| 0x1C | 1 |

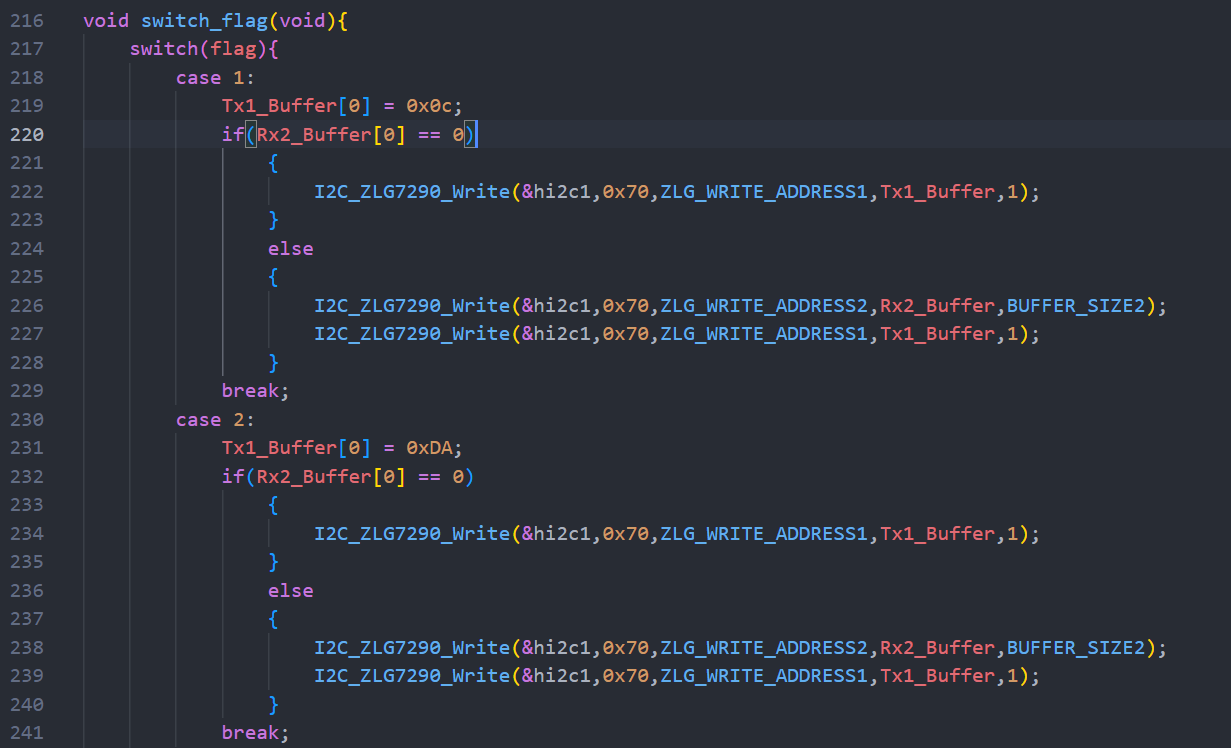
### 2.4 switch\_flag

根据flag的值不同，该函数会首先设置 Tx1\_Buffer[0] 的值，作为数码管显示的输出

之后会检查 Rx2\_Buffer[0] 的值是否为0，如果是，则执行一个I2C写操作，将数据发送到 ZLG\_WRITE\_ADDRESS1。

如果 Rx2\_Buffer[0] 不为0，则执行另一个I2C写操作，将数据发送到 ZLG\_WRITE\_ADDRESS2，然后再发送数据到 ZLG\_WRITE\_ADDRESS1。

需要注意的是当flag的值为14的时候会将Tx1\_Buffer[0]即数码管的回显值设置为全0，并且将数据发送八个字节到ZLG\_WRITE\_ADDRESS1



## 3.空间优化

### 3.1 合并switch\_flag和switch\_key函数

因为在函数的主体结构中我们知道两个函数之间通过flag变量来进行信息的传递，但是我们可以直接将两个函数进行合并，使得Rx1\_Buffer[0]能够直接与Tx1\_Buffer[0]进行交互。因为两个函数中都使用了大量的if-else语句，将两个函数合并之后可以减少掉if-else判断的数量，需要注意的是清零功能需要特殊处理。

### 3.2 删掉flag变量

因为在实验中要求我们只优化空间，并没有对程序运行的时间做要求，并且删去一个变量之后会为程序节省时间，但是代价是会使得程序在获取flag的值时每次都需要访问Rx1\_Buffer数组的值，会增加程序的时间，因此为了最大化利用程序的空间，可以将flag变量直接删去

### 3.3 删掉printf函数

因为printf的作用只是为了方便在编码的过程中进行调试，在程序运行的过程中并不会起到实际作用，所以可以将printf函数直接删去

### 3.4优化后程序源代码

main函数主体部分

int main(void)

{

  HAL\_Init();

  SystemClock\_Config();

  MX\_GPIO\_Init();

  MX\_I2C1\_Init();

  MX\_USART1\_UART\_Init();

  while (1)

  {

        if(flag1 == 1)

        {

            flag1 = 0;

            I2C\_ZLG7290\_Read(&hi2c1,0x71,0x01,Rx1\_Buffer,1);

            I2C\_ZLG7290\_Read(&hi2c1,0x71,0x10,Rx2\_Buffer,8);

            switch\_flag();

        }

  }

}

switch\_flag函数

void switch\_flag(void){

    switch(Rx1\_Buffer[0]){

        case 0x1C://按键1 输出1

            Tx1\_Buffer[0] = 0x0c; break;

        case 0x1B://按键2 输出2

            Tx1\_Buffer[0] = 0xDA; break;

        case 0x1A://按键3 输出3

            Tx1\_Buffer[0] = 0xF2; break;

        case 0x14://按键4 输出4

            Tx1\_Buffer[0] = 0x66; break;

        case 0x13://按键5 输出5

            Tx1\_Buffer[0] = 0xB6; break;

        case 0x12://按键6 输出6

            Tx1\_Buffer[0] = 0xBE; break;

        case 0x0C://按键7 输出7

            Tx1\_Buffer[0] = 0xE0; break;

        case 0x0B://按键8 输出8

            Tx1\_Buffer[0] = 0xFE; break;

        case 0x0A://按键9 输出9

            Tx1\_Buffer[0] = 0xE6; break;

        case 0x19://按键a 输出a

            Tx1\_Buffer[0] = 0xEE; break;

        case 0x11://按键b 输出b

            Tx1\_Buffer[0] = 0x3E; break;

        case 0x09://按键c 输出c

            Tx1\_Buffer[0] = 0x9C; break;

        case 0x01://按键d 输出d

            Tx1\_Buffer[0] = 0x7A; break;

        case 0x03://按键0 输出0

            Tx1\_Buffer[0] = 0xFC; break;

        case 0x02://

            Tx1\_Buffer[0] = 0x00;

            I2C\_ZLG7290\_Write(&hi2c1,0x70,ZLG\_WRITE\_ADDRESS1,Tx1\_Buffer,8);

            break;

        default:

            break;

    }

    if(Rx1\_Buffer[0]!=0x02){

        if(Rx2\_Buffer[0] != 0)

            I2C\_ZLG7290\_Write(&hi2c1,0x70,ZLG\_WRITE\_ADDRESS2,Rx2\_Buffer,BUFFER\_SIZE2);

        I2C\_ZLG7290\_Write(&hi2c1,0x70,ZLG\_WRITE\_ADDRESS1,Tx1\_Buffer,1);

    }

}

## 4.优化结果

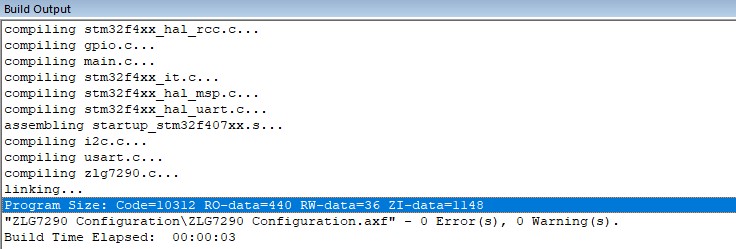
在程序编译之后会给出如下几个度量程序空间占用的值

Code：代表程序中执行的代码，即所有的函数占用空间

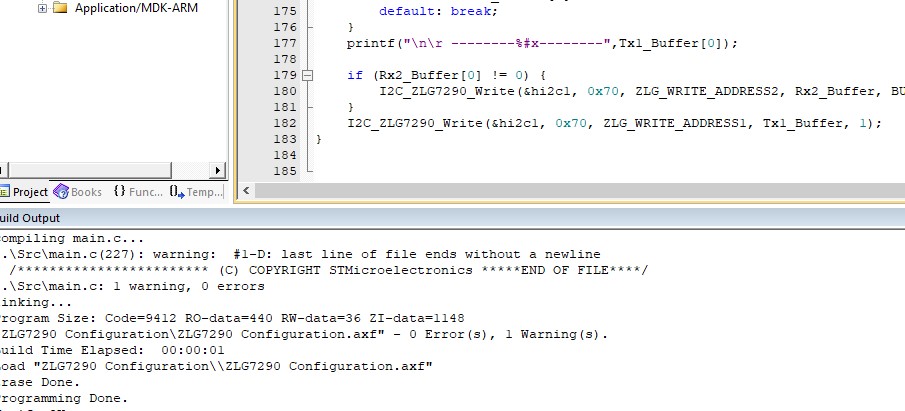
RO-data：代表程序中定义的全局常量与字符串形成的只读数据

RW-data：代表已经初始化完成的全局变量和静态常量的读写数据

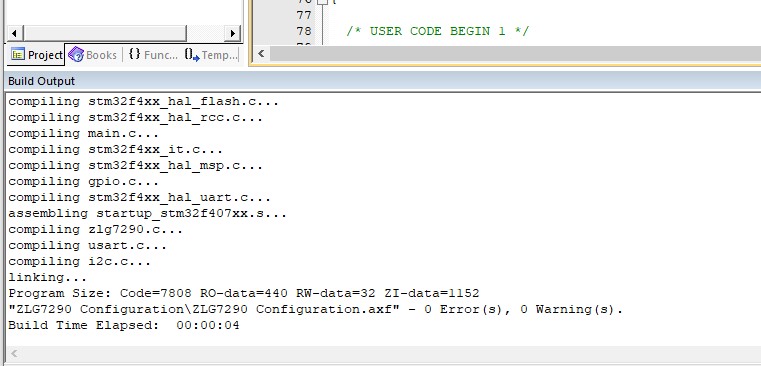
源码在lev0优化的情况下的空间占有情况



合并函数与删去变量之后的空间占有情况



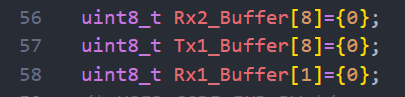
删去printf函数后的空间占有情况



## 5.空间优化方法总结

（1）数据结构优化

在不影响程序功能的前提下可以将程序中定义的变量的数据结构采用更小的数据类型，例如在本次实验的代码中均使用占有空间较小的uint8\_t数据结构



（2）变量优化

在程序执行的过程中可以将一些中间变量删去，让后面的变量直接调用前面变量的值来减少变量的使用从而减少空间的占用，但是这样做的代价是有可能将算法的运行时间提高

（3）运算方法的使用

在进行运算的过程中可以多使用复合运算来减少程序的代码量以及临时变量，对于乘法和除法运算可以选择占用空间更小的移位运算方式，但是这样做的代价是可能会将算法的复杂度提高，不利于后期的调试

（4）循环结构的使用

在程序编写的过程中循环结构也会占有过多的空间，因此应当尽量缩短循环次数，并且尽量提早使用break跳出循环，需要注意的是do-while语句的编译后代码长度要小于while语句

（5）逻辑结构优化

在编写if-else或者switch-case语句的过程中可以将共同的语句进行提取来简化代码量。

（6）去除不必要函数

在编写程序时可能会使用到printf等函数来显示程序输出方便在编写代码的过程中进行调试，在优化空间结构的时候可以将printf等在程序运行过程中不会起到实际作用的函数进行删去来保留程序的空间

（7）算法优化

选择合适的算法对性能有决定性影响。例如，快速排序比冒泡排序效率更高。可以使用选择排序和堆排序。这些算法在排序过程中不需要额外的存储空间，只通过交换元素在数组内部排序。

多使用原地操作，例如逆转数组或链表，可以通过修改原数据结构内部的链接或索引来实现，而不需要额外的存储空间。

许多动态规划问题可以通过仅保留必要的状态来优化空间，如使用滚动数组或压缩状态表格。

（8）压缩技术

可以通过进行数据压缩，对于大量数据，如图像、视频或大型数据集，使用压缩技术可以减少存储需求。例如使用数据压缩算法如Huffman编码、LZ77等来减少数据占用的空间。

对稀疏矩阵优化，对于包含大量零值的矩阵，可以使用特殊的数据结构如压缩稀疏行（CSR）或压缩稀疏列（CSC）格式存储非零元素和它们的索引。

（9）延迟计算和生成

在需要处理大量数据时，考虑使用延迟计算（Lazy Evaluation）技术，即仅在实际需要数据时才进行计算。这种方式可以减少同时存储在内存中的数据量。

（10）内存池

在频繁创建和销毁大量小对象的场景中，使用内存池技术可以减少内存分配和回收的开销，并减少内存碎片。

（11）结构共享

在数据结构如树或图中，共享不变的部分可以减少内存使用。例如，在实现函数式编程语言中的持久数据结构时常用到结构共享技术。

# 七、设计总结

在本次实验中合力设计出了一个火灾自动报警灭火系统，该系统在实现基本功能的基础上兼顾了抗干扰、容错和抗攻击等安全设计，设计的重心偏向在安全设计上，所以学习到了各种各样的嵌入式安全方面的知识。

在对整个系统进行设计的过程中我深刻的体会到了老师对于“功能性的实现是用户提出来的，安全性的实现才是我们的设计”这一观点。功能的实现只要用户提出要求，任何人设计出的程序在效果上可能都不会有太大的区别，并不能呢体现出我们的能力，而在安全设计上实现的各种各样的功能虽然在用户使用时不会看出来，但是在实际执行过程中确实在整个系统的安全方面起到了至关重要的作用。这也起始我在以后进行项目设计时在实现程序的功能的同时也要记住安全方面的要求，防止程序被攻击或者干扰。

老师在讲解安全原理时非常浅显易懂，使用我们能够理解的逻辑为我们介绍每一个安全设计的原因与作用，使得我们虽然没有太多的嵌入式实验基础但仍然能够理解每一个安全设计存在的意义。在最后对我们编写的程序进行优化时，也是从实际应用中出发，通过告诉我们如果某个位置坏掉之后的后果，从而让我们理解每一个安全设计对于错误的防范的原因。这让我能够很快速的理解每一个安全设计。

在本学期的嵌入式系统实践实验课中我学到了非常多的嵌入式安全方面的知识，也明白了在嵌入式开发过程中的艰难，但是我认为这门课的时间太过于紧凑了，致使我们在实现很多功能时并不能实现完整的功能，比如在设计加密模块时，为了节约时间，我们将保存到内存中的数据加1，在从内存中取出时减1，来实现加密功能。并没有太多的时间与我们之前学习的密码学结合使用更加复杂的加密算法。从而在进行安全性设计的很多地方只能够等价实现或者对设计步骤进行简化，没有从零开始设计出一整个完备的系统，因为这门实验真的很有趣，所以这一点我认为是比较遗憾的。