# x、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();

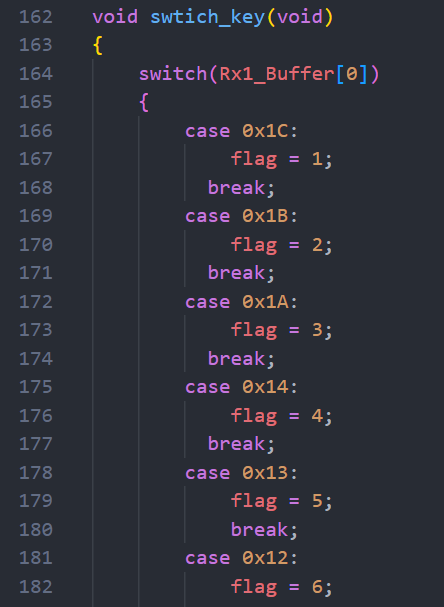
        }

  }

}

### 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 |

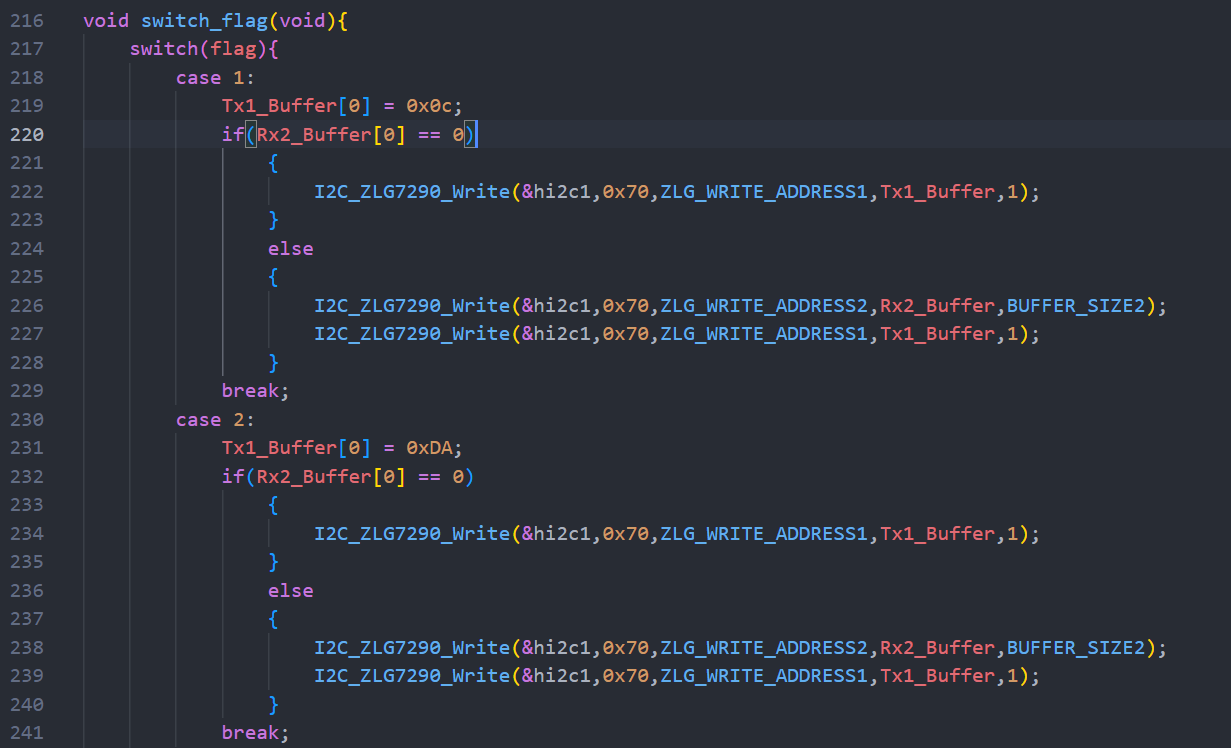
### 2.4 switch\_flag

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

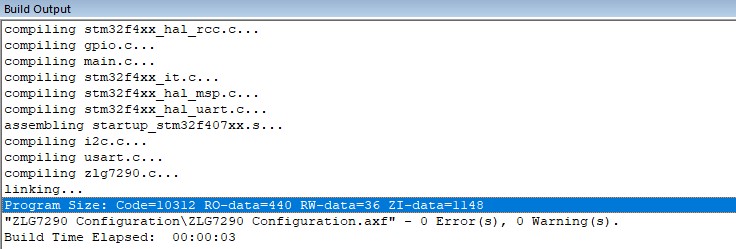
在程序编译之后会给出如下几个度量程序空间占用的值

Code：代表程序中执行的代码，即所有的函数占用空间

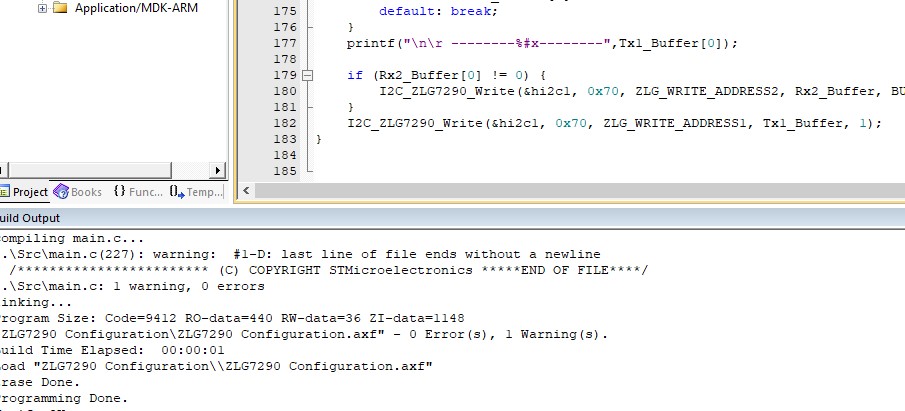
RO-data：代表程序中定义的全局常量与字符串形成的只读数据

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

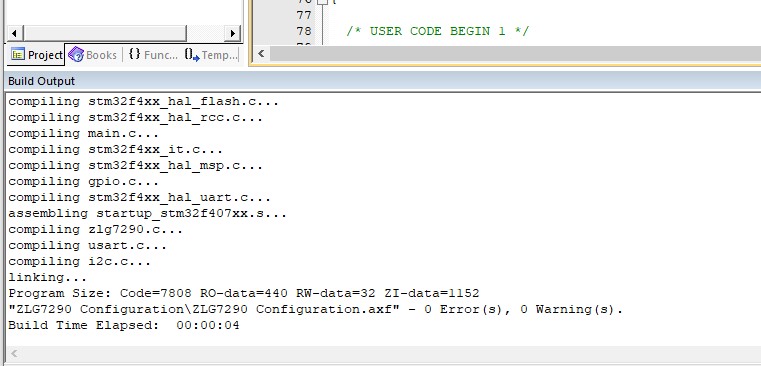
源码在lev0优化的情况下的空间占有情况



合并函数与删去变量之后的空间占有情况



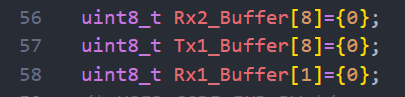
删去printf函数后的空间占有情况



## 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。