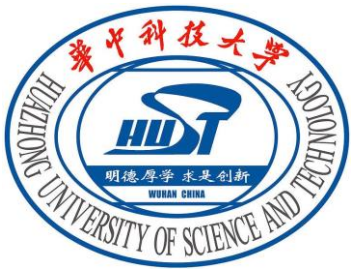

华中科技大学
实验报告
电子器件与电路（二）
嵌入式部分 4
走马灯的显示

学院	工程科学学院	
班级	工程科学学院（生医）1701 班	
小组成员	汪能志	U201713082
	周政宏	U201714460
指导老师	钟国辉	

2020 年 9 月 18 日

目录

1. 实验目的

1. 实现数字译码和动态显示（每一位数码管显示不同的内容）；
2. 实现数码管显示的滚动效果；
3. 构建显示模块，方便扩展显示不同的内容；
4. 提供译码组合，显示开机动画。

2. 实验原理

2.1. 多位数码管与扫描显示

可见，四位八段数码管中 A-G 和 DP 阳极引脚是各位数码管共用的，只能同时加载相同的信号。而每一位数码管中的八段 LED 是共阴极的。因此，让每一位数码管显示不同的内容的基本方法是扫描显示，即每次显示时在阴极选通一位数码管，并在阳极输入对应的信号；一定时间后，选通下一位数码管，输入下一位显示信号。当扫描周期足够快（每秒扫描 40 次以上）时，由于视觉暂留效应，人眼就会看到四位数码管在同时显示不同的内容。

2.2. 显示译码表

表 1：显示内容及对应的译码表

显示内容	8 位输出译码	显示内容	8 位输出译码
0	0xFC	A	0xEE
1	0x60	B	0xFE
2	0xDA	D	0xFC
3	0xF2	E	0x9E
4	0x66	L	0x1C
5	0xB6	O	0xFC
6	0xBE	S	0xB6
7	0xE0		
8	0xFE		
9	0xF6		

3. 实验实现

在每次触发中断时，对中断数量进行累加计数。计数值模 4 后作为显示的扫描序号，选通对应数码管，并将显示数据寄存器中对应字节的值送入相应的数码管。在扫描序号完成一次循环时，数码管 0-3 也会完成一次扫描，称为一帧。在这一帧中，四位数码管将按顺序依次显示 4byte 显示信号寄存器中的数据。

在缓冲显示模式下，每一组画面将显示预定的时间（帧数），因此还需要帧重复计数器、帧索引号和显示数据缓冲区。帧重复计数器用于记录画面重复显示的次数，帧索引号用于从显示数据缓冲区中读取正确的数据，显示数据缓冲区用于缓存之后需要使用的显示数据。

完成一帧的扫描显示后，帧重复数计数器累加。在帧重复计数器达到预定值之前，都将重复显示这一帧的内容，因此不会显示数据寄存器不会修改。在达到预设值后，就需要修改显示信号寄存

器中的内容，以显示更新后的内容。显示信号寄存器将根据对应的索引号，从显示数据缓冲区中读取 4byte 的数据。随后将更改显示索引号（循环显示模式），或者将显示数据缓冲区中被读取出的数据推出（一次性显示模式）。

在实时更新模式下，帧重复计数器、帧索引号和显示数据缓冲区将被停止使用。数码管将直接显示显示数据寄存器中的数据，在改写显示内容时，也将直接改写显示数据寄存器。

	帧（$5ms \times 4$）				帧					帧				帧					
触发中断计数	0	1	2	3	4	5	6	7	……	4N-4	4N-3	4N-2	4N-1	4N	4N+1	4N+2	4N+3		
扫描序号模4循环	0	1	2	3	0	1	2	3	……	0	1	2	3	0	1	2	3	……	0
帧重复计数	0				1				……	N-1				0				……	0
													载入下一组信号						

图 1 显示信号刷新周期

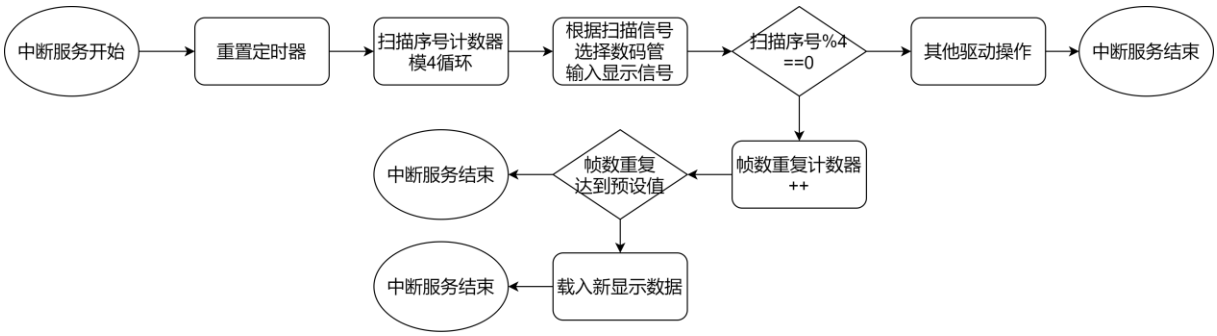

图 2 中断服务流程图

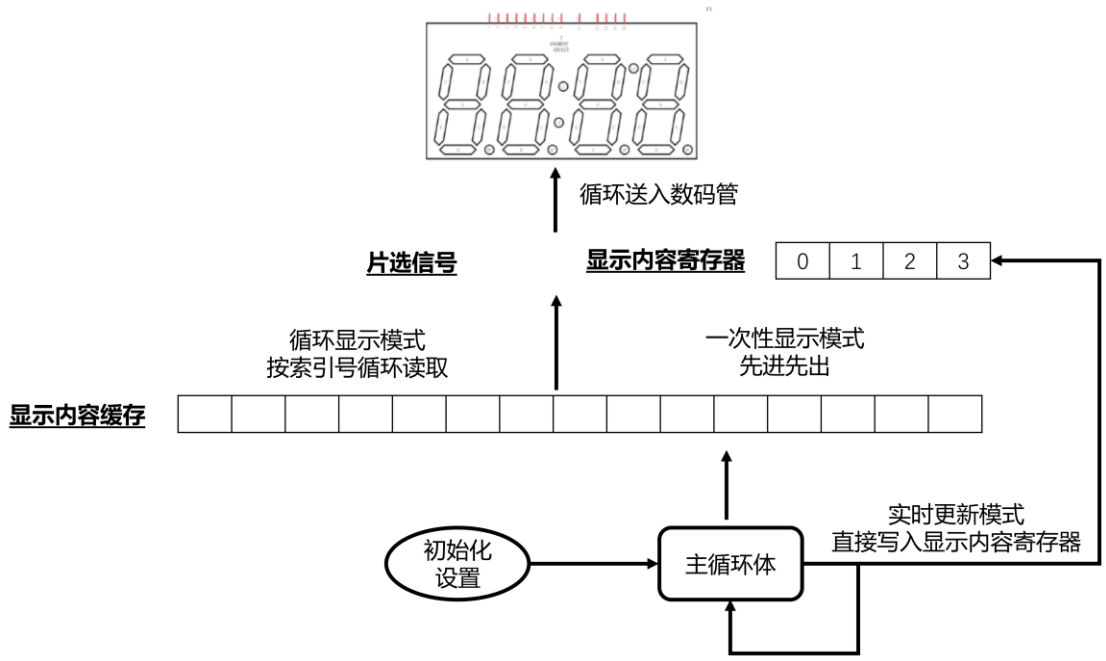

图 3 数码管驱动的缓存结构

程序中使用 16bit 定时器进行中断定时，不使用分频器，时钟信号源为指令时钟，TMR0H 和 TMR0L 分别设置为 0xEC 和 0x82，预设定时器溢出周期为 4990us。

由于初始化定时器 TMR 时会清空数据，因此两次进入中断的周期恰好为 5000us。

4. 程序调试与实验结果

使用跑表进行调试，将光标置于中断服务函数中重置定时器处。可见，两次进入中断的周期为 5000us。

```
Target halted. Stopwatch cycle count = 5000 (5 ms)
Target halted. Stopwatch cycle count = 5000 (5 ms)
Target halted. Stopwatch cycle count = 5000 (5 ms)
Target halted. Stopwatch cycle count = 5000 (5 ms)
```

图 4 扫描周期调试结果

记录中断服务函数的时间，可见在不执行显示内容切换时，需要消耗 41 个指令周期；显示内容切换较为复杂，需要占用约 2000 个指令周期。为了方便调试，调试时将帧重复时间改为 1，即每一帧画面只重复显示一次。在正常显示的情况下，显示内容切换的频率较低（如果 1s 切换一次显示内容，则每 200 次中断才会进行一次显示内容切换程序），因此这一资源占用目前可以接受。

```
Target halted. Stopwatch cycle count = 4959 (4.959 ms)
Target halted. Stopwatch cycle count = 41 (41 µs)
Target halted. Stopwatch cycle count = 4959 (4.959 ms)
Target halted. Stopwatch cycle count = 2157 (2.157 ms)
Target halted. Stopwatch cycle count = 2843 (2.843 ms)
Target halted. Stopwatch cycle count = 41 (41 µs)
Target halted. Stopwatch cycle count = 4959 (4.959 ms)
Target halted. Stopwatch cycle count = 41 (41 µs)
Target halted. Stopwatch cycle count = 4959 (4.959 ms)
Target halted. Stopwatch cycle count = 41 (41 µs)
Target halted. Stopwatch cycle count = 4959 (4.959 ms)
Target halted. Stopwatch cycle count = 2005 (2.005 ms)
```

图 5 中断服务程序耗时调试结果

根据以上调试内容，程序可以实现预设的显示效果。

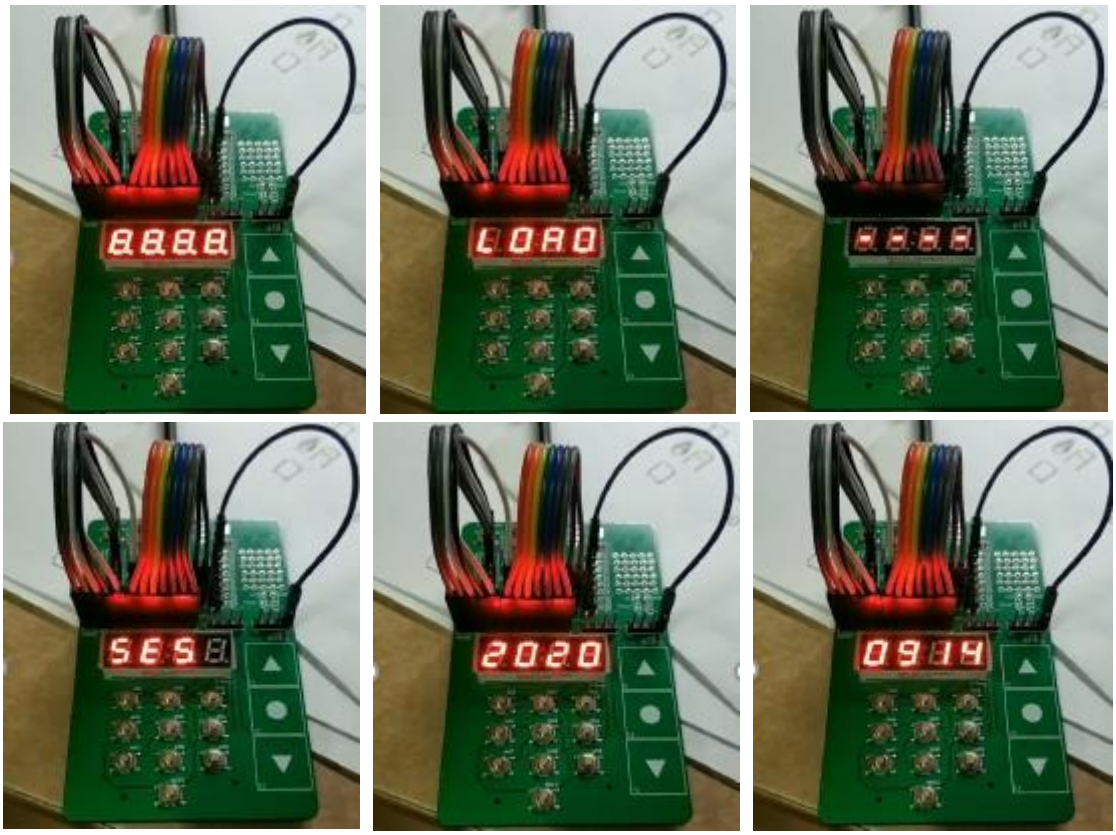

图 6 开机动画显示

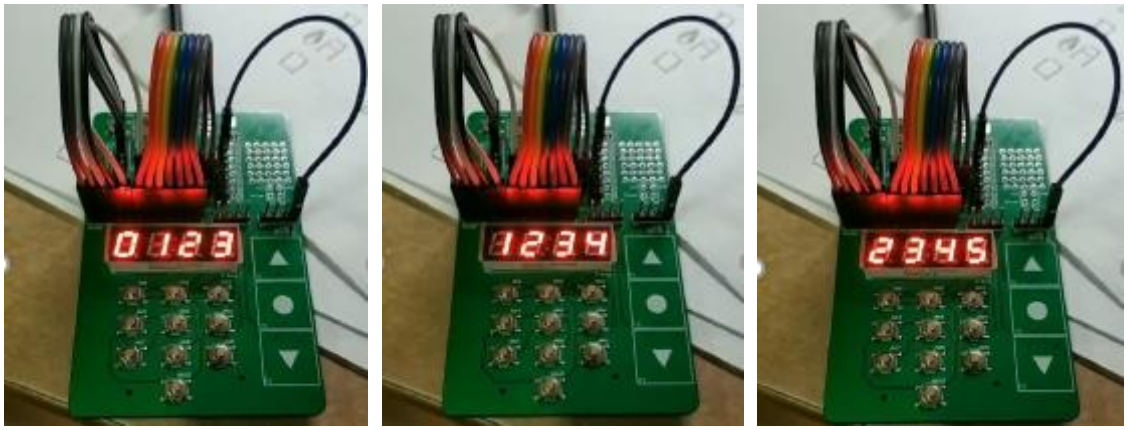

图 7 流水灯显示

实验结果均实现了非常完美的效果，如图所示。具体而言，动态显示可以非常清晰的看见 1、2、3、4 四个“静态数字”；走马灯动态显示则可以看见数字 1 至 9 在不断轮回变化；拓展实验中，开机动画包括“LOAD”，小数点接着闪，“SES”，“2020”，“0914”然后开始走马灯展示。

5. 实验小结

5.1. 遇到的问题

首先是显示乱码的问题，程序似乎都是对的，但是在板子上运行起来就是显示乱码，经过一段时间后的调试，我在中断前加入了对端口清 0 的操作，问题就解决了；此外在实验过程中我的板子坏了，因此还需要利用他人与队友的板子继续开展实验；其他关于整个实验流程与思路我们的进展挺顺利的。

5.2. 实验总结

5.2.1. 周政宏

本次实验基于上次中断加入了数码管显示，并完成了动态显示、走马灯、开机动画等实验。实验整体并不难，因此我们在非常快的完成了基本实验了开始着手思考如何实现更加复杂的功能，我们对我们最后实现的功能非常满意，当然也花费了非常多的时间调试。总的来说实验完成后，收获非常大，对数码管有了进一步的理解，对嵌入式系统相关代码的编写也更加熟练。

5.2.2. 汪能志

本次实验主要是数码管显示。多位数码管的扫描显示模式在某种程度上和 CRT 显像管显示器较为类似。CRT 显示器需要以此在扫描线上显示对应的内容，多位数码管也需要以此在每一位上输出对应的信号。在刷新速度较高的情况下，人眼就可以看到四位数码管在显示不同的内容。之后，为了方便显示不同的内容，特别是在主程序中实时改写显示的内容，我又对显示驱动的内容进行了大幅度的改写。先引入了显示信号寄存器，但是又出现了需要判断何时可以改写显示信号寄存器的内容，随后又引入了显示信号缓存。通过这两级缓存结构，最终完成了有多种显示模式的显示模块。通过实验，我对于动态显示有了进一步的了解，对于嵌入式代码的编写中需要注意的问题，例如：充分利用语言特性，在保持效果不变的情况下，尽可能缩短语句编译后占用的指令周期（如通过移位运算符进行对 2 的次幂的乘除法，通过位运算符取模等）。

附录：程序代码

```
#define TMR0_rst TMR0H = 0xEC, TMR0L = 0x82, PIR0bits.TMR0IF = 0x00

const unsigned char digital_decode[10] = {0xFC, 0x60, 0xDA, 0xF2, 0x66,
0xB6, 0xBE, 0xE0, 0xFE, 0xF6};
const unsigned char LED_select_signal[4] = {0xEE, 0xDD, 0xBB, 0x77};
// Display LEDs Select decode
unsigned char repeat_num;      //
unsigned char frame_repeat_num; // max number of a frame repeat
unsigned char interrupt_count = 0;

unsigned char display_signal[4] = {0xFF, 0xFF, 0xFF, 0xFF};

unsigned char display_cache[128] = {
    28, 0xFC, 238, 0xFC, // LOAD
    29, 0xFC, 238, 0xFC, // LOAD
    29, 0xFD, 238, 0xFC, // LOAD
    29, 0xFD, 239, 0xFC, // LOAD
    29, 0xFD, 239, 0xFD, // LOAD
    182, 158, 182, 0,    // SES

    0xDA, 0xFC, 0xDA, 0xFC,
    0xDA, 0xFC, 0xDA, 0xFC,
    0xFC, 0xF7, 0x60, 0xE0,
    0xFC, 0xF7, 0x60, 0xE0,
    0x02, 0x02, 0x02, 0x02,
    0x00, 0x00, 0x00, 0x00};
unsigned char dis_cache_size = 52; // byte saved in cache
unsigned char sum_frame_num = 13;  // number of frame, in the display
signal cache
//unsigned char display_ctrl = 0b00000001;
unsigned char display_ctrl;
/*
    bit 0
        1   shift 4
        0   shift 1
    bit 1
        1   loop
        0   clear
    bit 2
        0   frame
        1   real time
 */
```

```
#define display_clear_4 display_ctrl = 3, repeat_num = 0
#define display_clear_1 display_ctrl = 2, repeat_num = 0
#define display_loop_4 display_ctrl = 1, repeat_num = 0, frame_num = 0,
frame_start = 0, frame_cache_num = (dis_cache_size >> 2)
#define display_loop_1 display_ctrl = 0, repeat_num = 0, frame_num = 0,
frame_start = 0, frame_cache_num = dis_cache_size
#define display_real_time display_ctrl = 4
unsigned char frame_cache_num;
unsigned char frame_num;   // index of displaying frame
unsigned char frame_start; // start index in display signal cache
// used when loading data from display signal cache to register

unsigned char frame_refresh_enable = 0;
unsigned char display_write_in = 0x00;

void frame_switch(void)
{
    if (display_ctrl != 4)
    {
        if (repeat_num == frame_repeat_num)
        {
            repeat_num = 0;

            if (display_ctrl & 0x02)
            {

                if (display_ctrl & 0x01) // shift 4 byte once
                {
                    display_signal[0] = display_cache[0];
                    display_signal[1] = display_cache[1];
                    display_signal[2] = display_cache[2];
                    display_signal[3] = display_cache[3];
                    for (unsigned char j = 0; j < dis_cache_size; j++)
                    {
                        display_cache[j] = display_cache[j + 4];
                    }
                    display_cache[dis_cache_size - 1] = 0x00;
                    display_cache[dis_cache_size - 2] = 0x00;
                    display_cache[dis_cache_size - 3] = 0x00;
                    display_cache[dis_cache_size - 4] = 0x00;
                    dis_cache_size -= 4;
                }
                else // shift 1 byte once
```

```
            {
                display_signal[0] = display_cache[0];
                display_signal[1] = display_cache[1];
                display_signal[2] = display_cache[2];
                display_signal[3] = display_cache[3];

                for (unsigned char j = 0; j < dis_cache_size; j++)
                {
                    display_cache[j] = display_cache[j + 1];
                }
                display_cache[dis_cache_size - 1] = 0x00;

                dis_cache_size--;
            }
        }
        else
        {
            if (frame_num == sum_frame_num)
            {
                frame_num = 0;
                frame_start = 0;
            }
            if (display_ctrl & 0x01)
            {
                display_signal[0] = display_cache[frame_start];
                display_signal[1] = display_cache[frame_start + 1];
                display_signal[2] = display_cache[frame_start + 2];
                display_signal[3] = display_cache[frame_start + 3];
                frame_start += 4;
            }
            else
            {
                frame_start = frame_num;
                for (unsigned char j = 0; j < 4; j++)
                {

                    if (frame_num + j > frame_cache_num - 1)
                    {
                        display_signal[j] = display_cache[frame_start + j
- frame_cache_num];
                    }
                    else
                    {
```

```
                        display_signal[j] = display_cache[frame_start +
j];
                    }
                }
            }
        }
        frame_num++;
    }
  }
}

void __interrupt() isr(void)
{
    // reset TMR0
    TMR0_rst;
    interrupt_count++;
    // clear PORTC
    PORTC = 0x00;
    /************display part************/
    PORTA = LED_select_signal[interrupt_count & 0x03];
    PORTC = display_signal[interrupt_count & 0x03];

    if (!(interrupt_count & 0x03))
    {
        repeat_num++;
        frame_switch();
    }
}

void display_cache_push_back()
{
    //   if (display_ctrl & 0x02)
    // can only push display data in clear mode

    display_cache[dis_cache_size] = display_write_in;
    dis_cache_size++;
    display_write_in = 0x00;
}

void display_cache_clear(void)
{
    for (unsigned short j = 0; j < 128; j++)
    {
        display_cache[j] = 0x00;
```

```
    }
    dis_cache_size = 0;
}

void port_init(void)
{
    // init PORTC
    ANSELA = 0x00;
    LATA = 0x00;
    TRISA = 0x00;

    ANSELC = 0x00;
    LATC = 0x00;
    TRISC = 0x00;
}

void int_tmr_init(void)
{
    // init interrupt
    INTCONbits.GIE = 1;
    // global interrupt    enable
    INTCONbits.PEIE = 0;
    // peripheral interrupt disable
    INTCONbits.INTEDG = 1;
    // interrupt          rising edge
    PIE0bits.TMR0IE = 1;
    // Timer0 interrupt    enable

    // init TMR0
    T0CON0 = 0xD0;
    T0CON1 = 0x40;
    TMR0_rst;
}

void start_disp(void)
{
    frame_repeat_num = 1;
    display_clear_4;
    while (dis_cache_size)
    {
    };
    display_cache_clear();
}
```

```
void digital_num_loop(void)
{
    frame_repeat_num = 1;
    for (unsigned char i = 0; i < 10; i++)
    {
        display_write_in = digital_decode[i];
        display_cache_push_back();
    }
    display_loop_1;
}

void setup(void)
{
    port_init();
    int_tmr_init();
    start_disp();
    digital_num_loop();
}

void loop(void)
{
}

void main(void)
{
    setup();
    while (1)
    {
        loop();
    }
    return;
}
```