

# 实验7指南

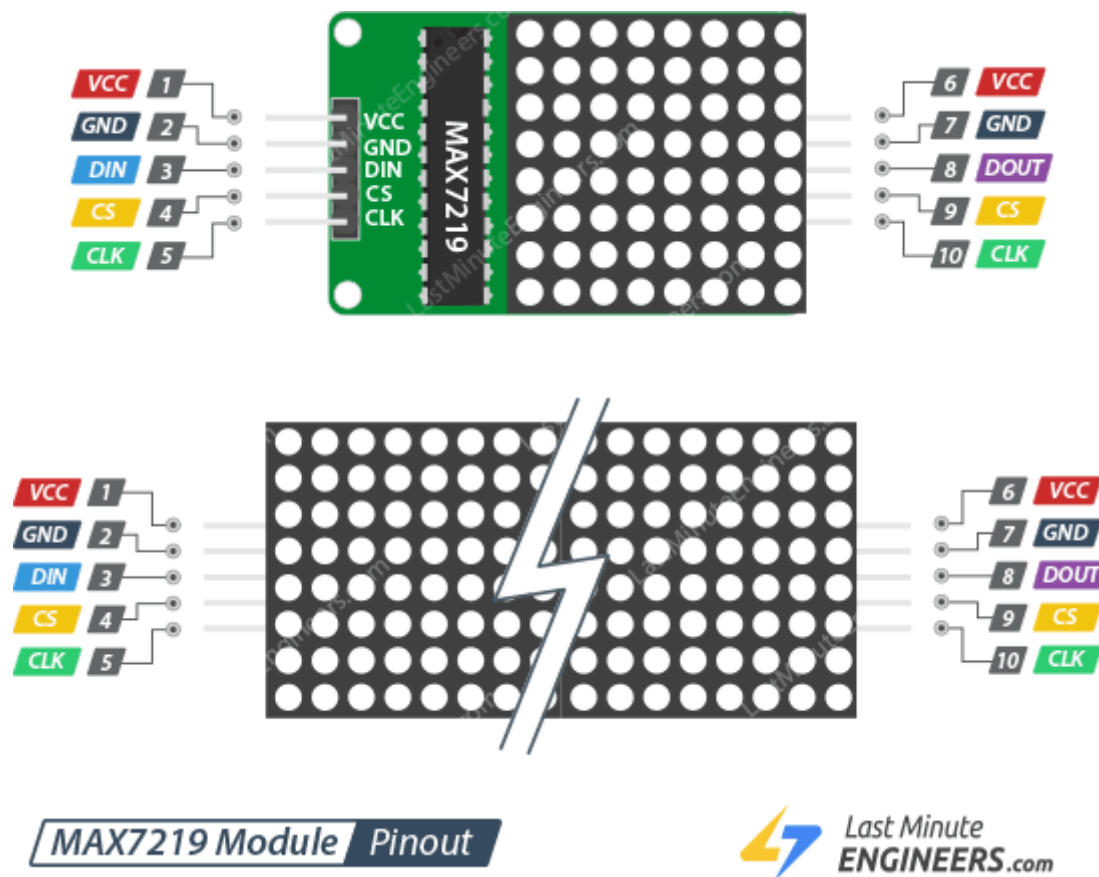
请跟随实验指南完成实验，完成文档中所有的 **TASK**。**BONUS** 部分的内容完成可作为加分，但报告的总分不应超过100分。请下载此指南作为实验报告模版，将填充完成的实验报告导出为PDF格式，并命名为“学号 姓名lab7.pdf”，上传至学在浙大平台。下载请点击 [这里](#)。

## 1 LED 矩阵

本实验中所使用的 LED 矩阵为 8x8 的点阵，使用 MAX7219 驱动。MAX7219 是一款集成了 8x8 LED 矩阵驱动电路的芯片，可以通过 SPI 接口控制。

### 1.1 引脚定义

MAX7219 的引脚定义如下图所示：



图片来自 <https://lastminuteengineers.com>

- 输入端接口
  - **VCC:** 电源。连接到5V。因为显示器需要很大的电流（在最大亮度时高达1A），最好使用外部电源而不是 3568 板的5V电源。如果你想使用 3568 板的 5V 电源，请将亮度保持在 25% 以下，以避免电压调节器过热。
  - **GND:** 地。连接到地。
  - **DIN:** 数据输入。将其连接到 3568 板的任一空闲 GPIO 引脚。DIN的数据仅在CLK处于上升沿的时候有效。

- **CS/LOAD:** 片选。将其连接到 3568 板的任一空闲 GPIO 引脚。作用为是控制串口通讯，其为低电平时，串行数据才被载入寄存器，高电平时会被锁存。
- **CLK:** 时钟。将其连接到 3568 板的任一空闲 GPIO 引脚。
- 输出端接口
  - **VCC:** 电源。连接到下一个 MAX7219 的 VCC 引脚。
  - **GND:** 地。连接到下一个 MAX7219 的 GND 引脚。
  - **DOUT:** 数据输出。将其连接到下一个 MAX7219 的 DIN 引脚。
  - **CS/LOAD:** 片选。将其连接到下一个 MAX7219 的 CS/LOAD 引脚。
  - **CLK:** 时钟。将其连接到下一个 MAX7219 的 CLK 引脚。

## 1.2 内置寄存器功能

以下内容摘录自 <https://juejin.cn/post/6976559353666994189>

通过查阅 max7219 的 [datasheet](#) 可以发现，max7219 芯片内不同的地址存储的值都有相应的作用。

0x01-0x08 对应的是 led 1 到 8 行的显示通过 8 位二进制数来控制这一行 led 的显示状况，0 led 熄灭，1 led 点亮。

比如说 0x3C 转换成二进制就是 00111100，该行的 led 的显示模式就是 ○○●●●●○○

地址	寄存器名称	作用
0x09	译码模式寄存器	0 为关闭译码模式
0x0b	扫描限制寄存器	设置扫描 led 的行数 1-8
0x0a	亮度调节寄存器	16 位调节亮度
0x0c	关断模式寄存器	0: 关断状态; 1: 正常操作状态
0x0f	显示测试寄存器	0: 正常模式; 1: 测试模式 (全部点亮)

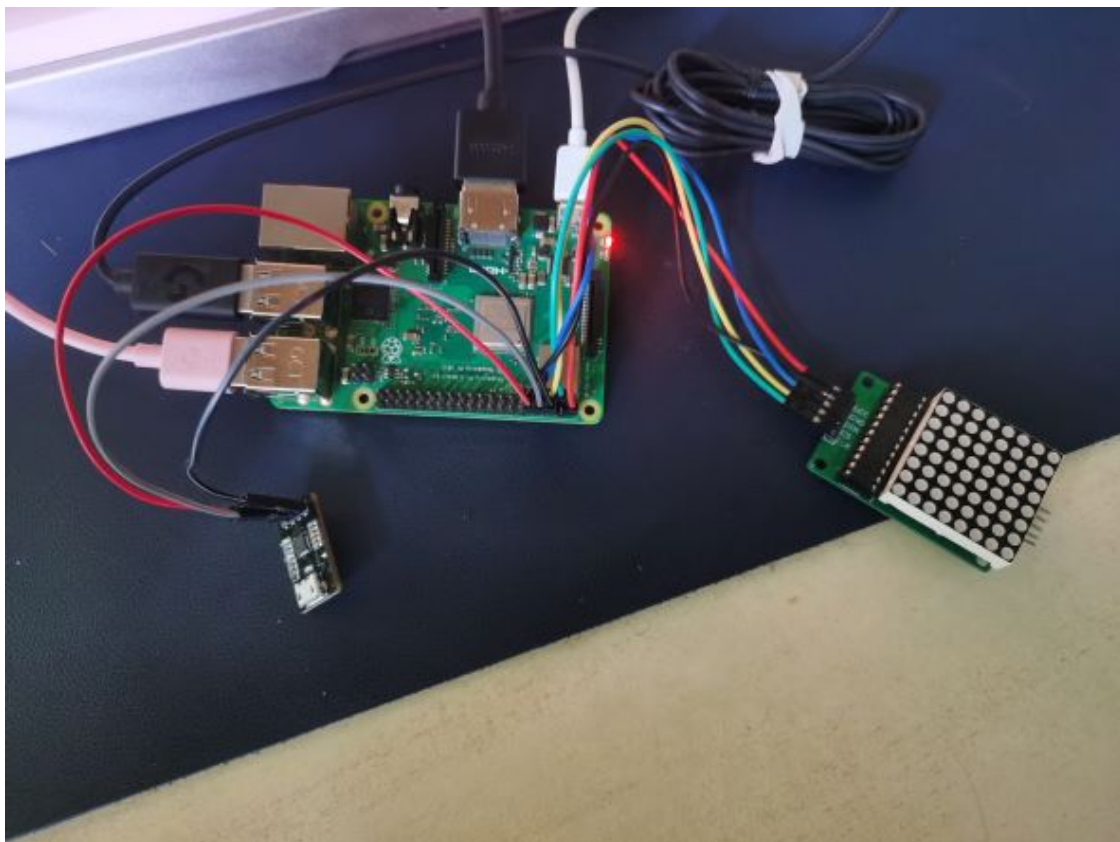
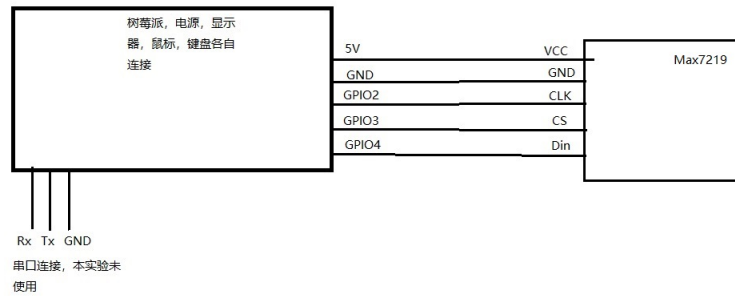
## 1.3 点亮 LED 矩阵

???+ tip "参考资料"

1. [第二个树莓派应用——LED点阵]  
(<https://juejin.cn/post/6976559353666994189#heading-9>)
2. [Interfacing MAX7219 LED Dot Matrix Display with Arduino]  
(<https://lastminuteengineers.com/max7219-dot-matrix-arduino-tutorial/>)
3. [树莓派学习笔记——使用文件IO操作GPIO SysFs方式]  
(<https://blog.csdn.net/xukai871105/article/details/38456801>)

**TASK1** 请设计接线方案，使得 3568 板能够在之后的实验中使用 LED 矩阵。画出你的连线示意图，并拍摄你实际连接的板卡照片。(5分)

## 连线示意图



**TASK2** 编写 C 程序, 采用 Arduino-ish 库或虚拟文件系统 (或其他你选择的库) 访问 GPIO, 实现在矩阵上显示文字或图案。请在下方给出你的源代码 (需有详细注释), 对你选用的方式 (库 / 虚拟文件系统) 做出阐释, 并对代码关键步骤进行解释。同时需附上 LED 矩阵成功显示文字或图案的照片。 (20分)

由于通过sysfs操作GPIO的代码比较庞大, 该项任务采用bcm2835库。

bcm2835库是一种C语言库, 由Broadcom开发, 用于控制树莓派上的硬件。它提供了一组函数, 可以直接访问GPIO引脚, 以及SPI和I2C总线。需要注意的是, bcm2835库需要在超级用户 (root) 权限下运行。

编译代码时需要通过 `-l` 选项链接bcm2835库, 如 `gcc task2.c -o task2 -lbcm2835`

运行 `sudo ./task2`

另外，虽然可以将bcm2835编译为静态库安装进系统中，也可以将bcm2835.h  
bcm2835.c 复制到代码文件夹中当成自定义文件参与编译，可以达到相同的效果，gcc  
task2.c bcm2835.c -o task2

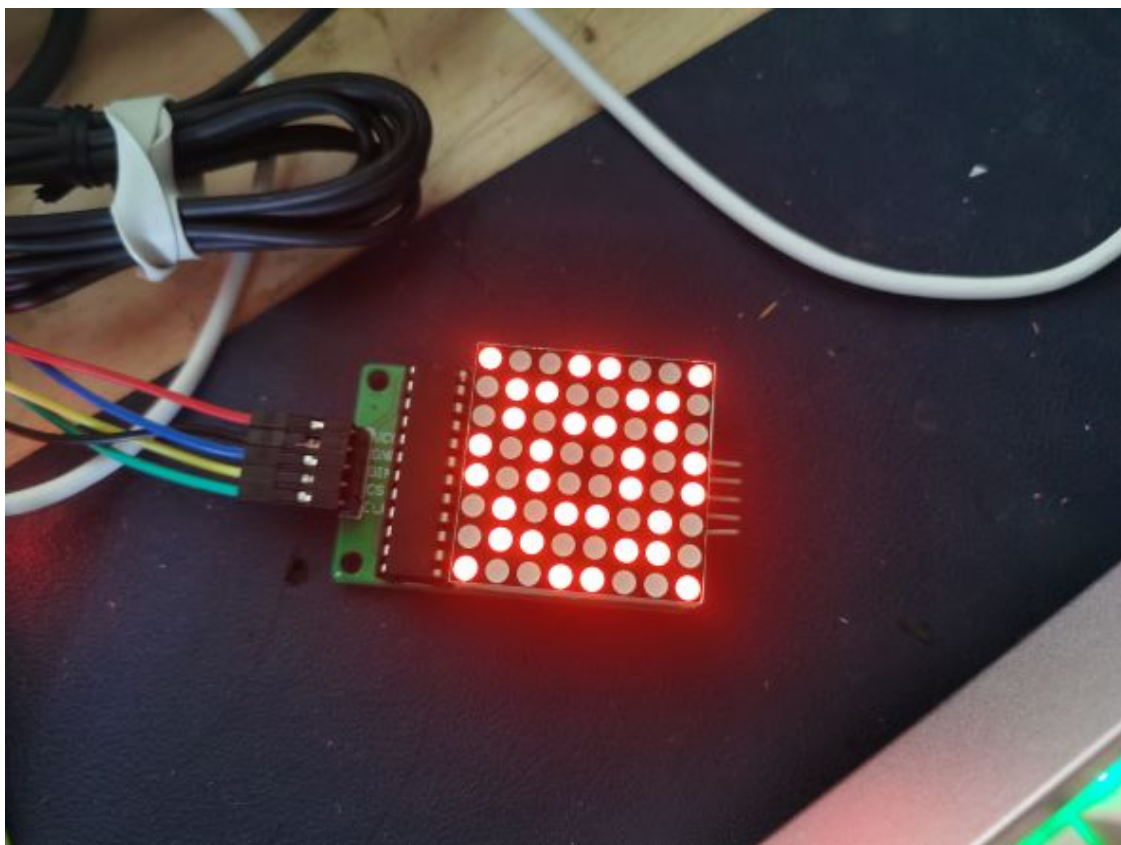
```
1  #include <bcm2835.h>
2  #include <stdio.h>
3  #include <time.h>
4
5  // 重新定义引脚名，注意右侧定义 _J8_03 说明是三号物理引脚，对应 GPIO2
6  #define CLK  RPI_BPLUS_GPIO_J8_03 // GPIO2
7  #define CS   RPI_BPLUS_GPIO_J8_05 // GPIO3
8  #define Din  RPI_BPLUS_GPIO_J8_07 // GPIO4
9
10 void setBitData(int data);
11 void write_Max7219_byte(int data);
12 void write_Max7219(int addr, int data);
13 int gpio_Max7219_init();
14 int heart[] = {0x00,0x66, 0xff, 0xff, 0xff, 0x7e, 0x3c, 0x18};
15
16 char matrix[] = {0x99,0x66,0x5a,0xa5,0xa5,0x5a,0x66,0x99};
17
18 int gpio_Max7219_init() //初始化bcm2835库并设置好对应的GPIO
19 {
20     if(!bcm2835_init()) //使用库函数前需要初始化
21     {
22         printf("BCM2835 initialization fails !!!");
23         return -1;
24     }
25
26     bcm2835_delay(800); //延时准备
27     bcm2835_gpio_fsel(CS, BCM2835_GPIO_FSEL_OUTP); //配置GPIO方向为输出
28     bcm2835_gpio_fsel(CLK, BCM2835_GPIO_FSEL_OUTP);
29     bcm2835_gpio_fsel(Din, BCM2835_GPIO_FSEL_OUTP);
30
31     // 设置Max7219
32     write_Max7219(0x09, 0x00); // 关闭译码模式
33     write_Max7219(0x0a, 0x00); // 设置led亮度 3 ,15最大
34     write_Max7219(0x0b, 0x07); // 设置扫描行数
35     write_Max7219(0x0c, 0x01); // 设置正常模式
36     write_Max7219(0x0f, 0x00); // 设置正常模式
37 }
38
39 void setBitData(int data)
40 {
41     // 传输1bit数据，需要先拉低时钟再拉高，数据才会传输出去
42     bcm2835_gpio_write(CLK, LOW);
43     bcm2835_gpio_write(Din, data);
44     bcm2835_gpio_write(CLK, HIGH);
45 }
46
47 void write_Max7219_byte(int data)
```

```

48 {
49     //传输一个字节的的数据
50     // 从高位到低位判断是否为1 or 0, 传输即可
51     int i;
52     for(i = 0 ; i < 8 ; i++){
53         if(((data <<i) & 0x80) == 0x80){//判断对应bit位是否为1
54             setBitData(HIGH);
55         }
56         else {
57             setBitData(LOW);
58         }
59     }
60 }
61
62 void write_Max7219(int addr, int data)
63 {
64     //传输一次数据，即两个字节
65     // 先传地址，再传数据
66     bcm2835_gpio_write(CS, LOW);// 将CS信号拉低才能写入数据
67     write_Max7219_byte(addr);//传输地址
68     write_Max7219_byte(data);//传输数据
69     bcm2835_gpio_write(CS, HIGH);//拉高CS，说明此次传输完成
70 }
71
72 void ShowDotMatrix()
73 {
74     int i=0;
75     for (i=0;i<8;i++){
76         write_Max7219(i+1,heart[i]);//将要显示的图案数据依次写入
77     }
78 }
79
80 int main()
81 {
82     gpio_Max7219_init();
83     ShowDotMatrix();
84     return 0;
85 }

```

显示图案：



## 2 字符设备驱动

### 2.1 Linux内核模块

此节内容摘录自 [Linux驱动基本知识 - Linux内核模块](#)

Linux是一个跨平台的操作系统，支持众多的设备，在Linux内核源码中有超过50%的代码都与设备驱动相关。Linux为宏内核架构，如果开启所有的功能，内核就会变得十分臃肿。内核模块就是实现了某个功能的一段内核代码，在内核运行过程，可以加载这部分代码到内核中，从而动态地增加了内核的功能。基于这种特性，我们进行设备驱动开发时，以内核模块的形式编写设备驱动，只需要编译相关的驱动代码即可，无需对整个内核进行编译。

模块是具有独立功能的程序，它可以被单独编译，但不能独立运行，在运行时它被链接到内核作为内核的一部分在内核空间运行，这与运行在用户空间的进程是不一样的。模块由一组函数和数据结构组成，用来实现一种文件系统、一个驱动程序和其他内核上层功能。因此内核模块具备如下特点：

- 模块本身不被编译入内核映像，这控制了内核的大小。
- 模块一旦被加载，它就和内核中的其它部分完全一样。

我们编写的内核模块，经过编译，最终形成.ko为后缀的ELF文件。我们可以使用file命令来查看它。

### 2.2 编译内核



### 2.3.1 交叉编译环境配置

此节内容参考自

[https://doc.embedfire.com/linux/imx6/driver/zh/latest/linux\\_driver/exper\\_env.html](https://doc.embedfire.com/linux/imx6/driver/zh/latest/linux_driver/exper_env.html)

设备驱动是具有独立功能的程序，它可以被单独编译，但不能独立运行，在运行时它被链接到内核作为内核的一部分在内核空间运行。也因此想要我们写的内核模块在某个版本的内核上运行，那么就必须在该内核版本上编译它，如果我们编译的内核与我们运行的内核具备不相同的特性，设备驱动则可能无法运行。

由于嵌入式 Linux 板卡的性能有限，编译内核需要较长时间，因此我们需要使用交叉编译的方式，在PC上编译内核，然后将编译好的内核模块传输到板卡上。

可以参考 [编译环境搭建](#) 下载并解压 [Firefly Linux SDK源码包](#)。

需要选择和你之前烧录固件相同版本的 Linux 内核，或者也可以选择编译好 SDK 以后重新烧录一次固件。

这里仍然只有百度网盘，且 SDK 源码很大（十几GB），超过了浙大云盘的限额，所以要提前做好心理准备。

同时配置好相关的交叉编译环境（推荐使用docker，并参考以上教程使用如下的 dockerfile）：

```
1 FROM ubuntu:18.04
2 MAINTAINER firefly "service@t-firefly.com"
3
4 ENV DEBIAN_FRONTEND=noninteractive
5
6 RUN cp -a /etc/apt/sources.list /etc/apt/sources.list.bak
7 RUN sed -i 's@http://.*ubuntu.com@http://repo.huaweicloud.com@g'
   /etc/apt/sources.list
8
9 RUN apt update
10
11 RUN apt install -y build-essential crossbuild-essential-arm64 \
12     bash-completion vim sudo locales time rsync bc python
13
14 RUN apt install -y repo git ssh libssl-dev liblz4-tool lib32stdc++6 \
15     expect patchelf chrpath gawk texinfo diffstat binfmt-support \
16     qemu-user-static live-build bison flex fakeroot cmake \
17     unzip device-tree-compiler python-pip ncurses-dev python-pyelftools
   \
18     subversion asciidoc w3m dblatex graphviz python-matplotlib cpio \
19     libparse-yapp-perl default-jre patchutils swig expect-dev u-boot-
   tools
20
21 RUN apt update && apt install -y -f
22
23 # language support
24 RUN locale-gen en_US.UTF-8
25 ENV LANG en_US.UTF-8
26
```

```
27 # switch to a no-root user
28 RUN useradd -c 'firefly user' -m -d /home/firefly -s /bin/bash firefly
29 RUN sed -i -e '/\%sudo/ c \%sudo ALL=(ALL) NOPASSWD: ALL' /etc/sudoers
30 RUN usermod -a -G sudo firefly
31
32 USER firefly
33 WORKDIR /home/firefly
```

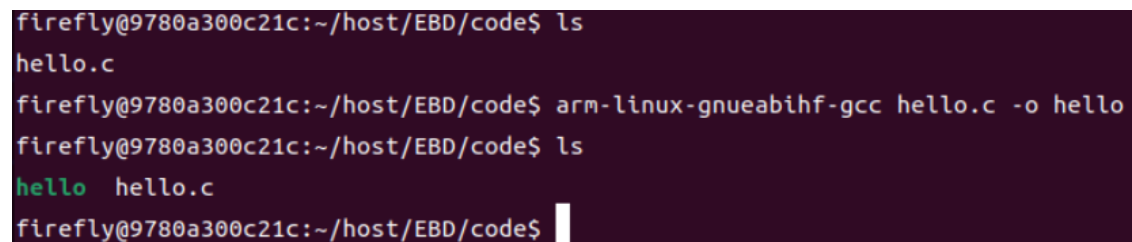
???+ warning "警告"

所给的参考资料使用的上位机架构为 x86\_64，如果你使用的上位机架构为 arm64（对说的就是你 Apple 的 M 系列芯片），那么就需要对以上教程中的相关配置进行修改。（实际上是更简单了，因为如果使用 docker 中的 Linux 进行编译，那么就不需要进行交叉编译，直接编译即可）

**TASK3** 请参考以上教程配置 docker 镜像，创建容器并启动。在 docker 中使用交叉编译工具编译 C 语言的 "Hello, world" 程序，并将编译后的二进制文件传输到 3568 板上运行。请给出你在 docker 中成功交叉编译产生二进制文件的截图以及在 3568 板上成功运行此程序的截图。（15 分）

```
1 #include <stdio.h>
2 int main()
3 {
4     printf("Hello, world!\n");
5     return 0;
6 }
```

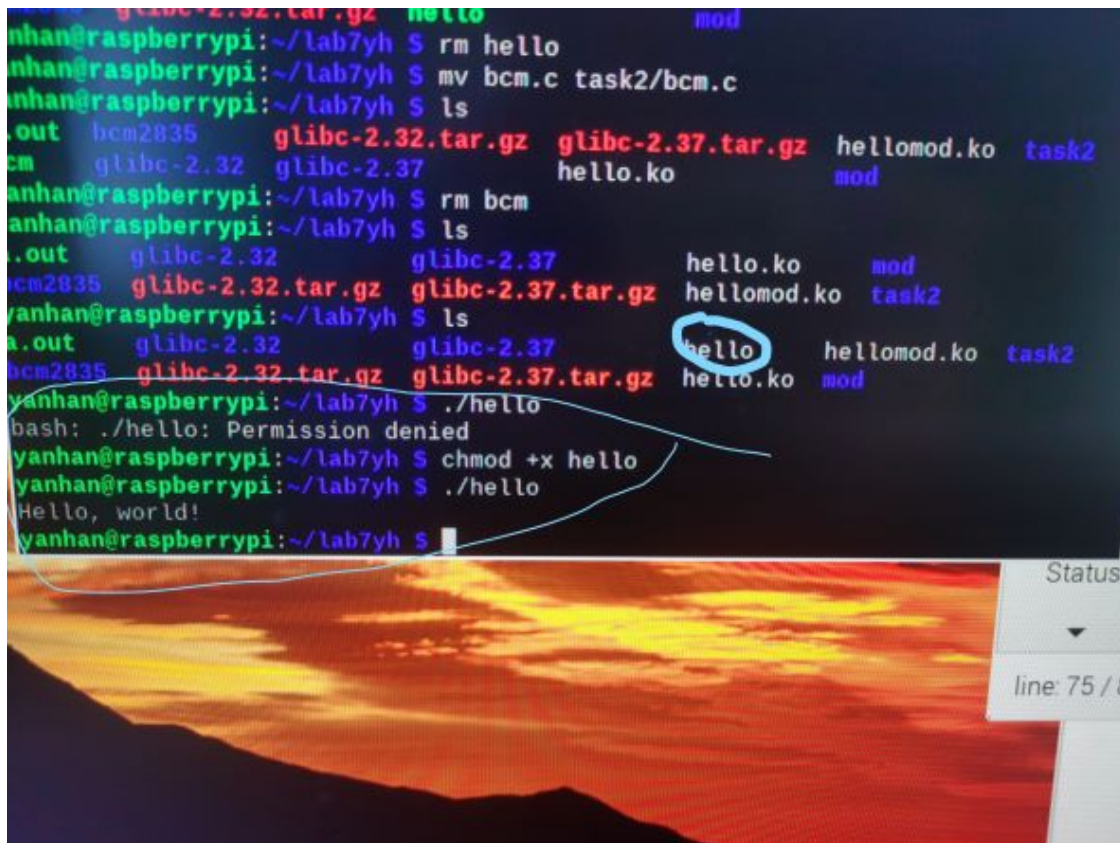
编译：



```
firefly@9780a300c21c:~/host/EBD/code$ ls
hello.c
firefly@9780a300c21c:~/host/EBD/code$ arm-linux-gnueabihf-gcc hello.c -o hello
firefly@9780a300c21c:~/host/EBD/code$ ls
hello hello.c
firefly@9780a300c21c:~/host/EBD/code$
```

运行（通过sftp传输，需要先添加执行权限）：





```
yanhan@raspberrypi:~/lab7yh $ rm hello
yanhan@raspberrypi:~/lab7yh $ mv bcm.c task2/bcm.c
yanhan@raspberrypi:~/lab7yh $ ls
.out      bcm2835      glibc-2.32.tar.gz  glibc-2.37.tar.gz  hellomod.ko  task2
bcm       glibc-2.32  glibc-2.37        hello.ko            mod
yanhan@raspberrypi:~/lab7yh $ rm bcm
yanhan@raspberrypi:~/lab7yh $ ls
.out      glibc-2.32      glibc-2.37        hello.ko            mod
bcm2835   glibc-2.32.tar.gz  glibc-2.37.tar.gz  hellomod.ko        task2
yanhan@raspberrypi:~/lab7yh $ ls
.out      glibc-2.32      glibc-2.37        hello               hellomod.ko        task2
bcm2835   glibc-2.32.tar.gz  glibc-2.37.tar.gz  hello.ko            mod
yanhan@raspberrypi:~/lab7yh $ ./hello
bash: ./hello: Permission denied
yanhan@raspberrypi:~/lab7yh $ chmod +x hello
yanhan@raspberrypi:~/lab7yh $ ./hello
Hello, world!
yanhan@raspberrypi:~/lab7yh $
```

### 2.3.2 开始编译内核

请参考 [Kernel 使用](#)，在前一节中配置好的环境中编译 Linux 内核。

**TASK4** 请参考以上教程，编译内核。请给出编译过程中的截图，以及编译完成后生成的内核镜像文件的截图。若选择与之前固件内核版本不同需要重新烧录，则需要同时放上烧录成功的截图。（15分）

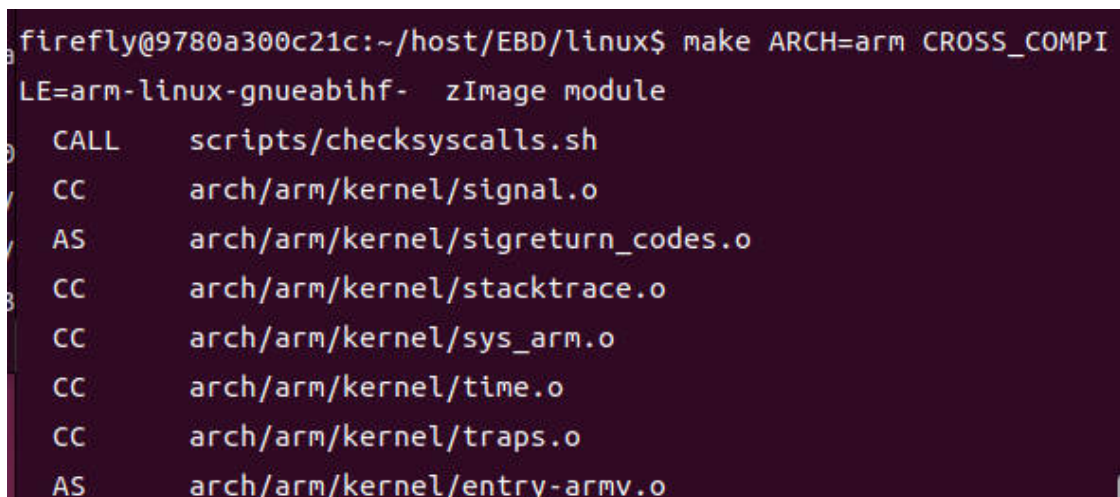
烧录的树莓派内核版本为 6.1.y，下载对应源码即可。

获取树莓派的内核配置 `sudo modprobe configs`

将该配置解压至内核源码根目录下，运行 `KERNEL=kernel7 make ARCH=arm`

`CROSS_COMPILE=arm-linux-gnueabihf- zImage modules`（使用自己的配置前最好先用默认配置配一次，再覆盖 `.config` 文件，否则会有文件缺失）

编译过程：



```
firefly@9780a300c21c:~/host/EBD/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage module
CALL scripts/checksyscalls.sh
CC      arch/arm/kernel/signal.o
AS      arch/arm/kernel/sigreturn_codes.o
CC      arch/arm/kernel/stacktrace.o
CC      arch/arm/kernel/sys_arm.o
CC      arch/arm/kernel/time.o
CC      arch/arm/kernel/traps.o
AS      arch/arm/kernel/entry-armv.o
```

(modules 编译选项错误, 后面重新编译 modules)

```
LD [M] net/ipv4/udp_tunnel.ko
CC [M] net/ipv4/xfrm4_tunnel.mod.o
LD [M] net/ipv4/xfrm4_tunnel.ko
CC [M] net/ipv6/ah6.mod.o
LD [M] net/ipv6/ah6.ko
CC [M] net/ipv6/esp6.mod.o
LD [M] net/ipv6/esp6.ko
CC [M] net/ipv6/esp6_offload.mod.o
LD [M] net/ipv6/esp6_offload.ko
CC [M] net/ipv6/fou6.mod.o
```

编译完成:

zImage:



```
AS      .tmp_vmlinux.kallsyms2.5
LD      vmlinux
NM      System.map
SORTTAB vmlinux
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
LDS     arch/arm/boot/compressed/vmlinux.lds
AS      arch/arm/boot/compressed/head.o
GZIP    arch/arm/boot/compressed/piggy_data
AS      arch/arm/boot/compressed/piggy.o
CC      arch/arm/boot/compressed/misc.o
CC      arch/arm/boot/compressed/decompress.o
CC      arch/arm/boot/compressed/string.o
AS      arch/arm/boot/compressed/hyp-stub.o
CC      arch/arm/boot/compressed/fdt_rw.o
CC      arch/arm/boot/compressed/fdt_ro.o
CC      arch/arm/boot/compressed/fdt_wip.o
CC      arch/arm/boot/compressed/fdt.o
CC      arch/arm/boot/compressed/fdt_check_mem_start.o
AS      arch/arm/boot/compressed/lib1funcs.o
AS      arch/arm/boot/compressed/ashldi3.o
AS      arch/arm/boot/compressed/bswapsdi2.o
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
```

modules:

```
LD [M] sound/usb/hiface/snd-usb-hiface.ko
CC [M] sound/usb/line6/snd-usb-line6.mod.o
LD [M] sound/usb/line6/snd-usb-line6.ko
CC [M] sound/usb/line6/snd-usb-toneport.mod.o
LD [M] sound/usb/line6/snd-usb-toneport.ko
CC [M] sound/usb/misc/snd-ua101.mod.o
LD [M] sound/usb/misc/snd-ua101.ko
CC [M] sound/usb/snd-usb-audio.mod.o
LD [M] sound/usb/snd-usb-audio.ko
CC [M] sound/usb/snd-usbmidi-lib.mod.o
LD [M] sound/usb/snd-usbmidi-lib.ko
```

## 2.3 驱动编写

### 2.3.1 第一个内核模块

请先参考 [第一个内核模块](#) 和 [Linux驱动实践：带你一步一步编译内核驱动程序](#)，尝试一个最简单 hello module 框架。需要注意 `kconfig` 和 `Makefile` 的编写。

**TASK5** 将以上框架编译成 `.ko` 文件后，传输到 3568 板上，加载此驱动后再卸载此驱动，并在过程中观察串口输出，并使用如 `lsmod`、`dmesg` 等命令查看驱动加载状态。请给出相应的截图和对截图的解释。（10分）

单独编译模块，得到 `hellomod.ko`：

```
firefly@9780a300c21c:~/host/EBD/linux/drivers/hellomod$ ls
hellomod.c  Makefile
firefly@9780a300c21c:~/host/EBD/linux/drivers/hellomod$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf
-
make -C /home/firefly/host/EBD/linux M=/home/firefly/host/EBD/linux/drivers/hellomod modules
make[1]: Entering directory '/home/firefly/host/EBD/linux'
  CC [M] /home/firefly/host/EBD/linux/drivers/hellomod/hellomod.o
  MODPOST /home/firefly/host/EBD/linux/drivers/hellomod/Module.symvers
  CC [M] /home/firefly/host/EBD/linux/drivers/hellomod/hellomod.mod.o
  LD [M] /home/firefly/host/EBD/linux/drivers/hellomod/hellomod.ko
make[1]: Leaving directory '/home/firefly/host/EBD/linux'
firefly@9780a300c21c:~/host/EBD/linux/drivers/hellomod$ ls
hellomod.c  hellomod.mod  hellomod.mod.o  Makefile      Module.symvers
hellomod.ko hellomod.mod.c  hellomod.o      modules.order
firefly@9780a300c21c:~/host/EBD/linux/drivers/hellomod$
```

查看驱动加载状态：



```

yanhan@raspberrypi:~/lab7yh $ lsmod | grep hellomod  查询最初模块加载状态, 无输出, 说明未加载
yanhan@raspberrypi:~/lab7yh $ sudo insmod hellomod.ko  加载模块
yanhan@raspberrypi:~/lab7yh $ lsmod | grep hellomod
hellomod          16384  0  再次输出, 发现模块已加载
yanhan@raspberrypi:~/lab7yh $ dmesg | tail  输出内核信息
[ 31.848632] hwmmon hwmmon1: Voltage normalised
[ 93.540463] ICMPv6: process 'dhcpcd' is using deprecated sysctl (syscall) net.ipv6.neigh.wlan0.retrans_time - use net
.ipv6.neigh.wlan0.retrans_time_ms instead
[ 800.403694] hello: loading out-of-tree module taints kernel.
[ 800.404183] welcome, hello
[ 824.360369] bye, hello
[ 1687.274122] welcome, hello
[ 1696.229850] bye, hello
[ 5212.974140] welcome, hello
[ 5325.086127] bye, hello
[ 5435.747171] welcome, hello  加载模块的输出, 上面为之前的测试
yanhan@raspberrypi:~/lab7yh $ sudo rmmod hellomod.ko
yanhan@raspberrypi:~/lab7yh $ lsmod | grep hellomod  删除模块, 再次输出模块加载信息, 无输出信息, 模块已卸载
yanhan@raspberrypi:~/lab7yh $ dmesg | tail
[ 93.540463] ICMPv6: process 'dhcpcd' is using deprecated sysctl (syscall) net.ipv6.neigh.wlan0.retrans_time - use net
.ipv6.neigh.wlan0.retrans_time_ms instead
[ 800.403694] hello: loading out-of-tree module taints kernel.
[ 800.404183] welcome, hello
[ 824.360369] bye, hello
[ 1687.274122] welcome, hello
[ 1696.229850] bye, hello
[ 5212.974140] welcome, hello
[ 5325.086127] bye, hello
[ 5435.747171] welcome, hello
[ 5465.533193] bye, hello  模块卸载信息
yanhan@raspberrypi:~/lab7yh $

```

模块加载过程串口输出:

```

yanhan@raspberrypi:~/lab7yh $ sudo insmod hellomod.ko
yanhan@raspberrypi:~/lab7yh $ sudo rmmod hellomod.ko
yanhan@raspberrypi:~/lab7yh $ dmesg | tail
[ 800.404183] welcome, hello
[ 824.360369] bye, hello
[ 1687.274122] welcome, hello
[ 1696.229850] bye, hello
[ 5212.974140] welcome, hello
[ 5325.086127] bye, hello
[ 5435.747171] welcome, hello
[ 5465.533193] bye, hello
[ 5814.119744] welcome, hello
[ 5822.991958] bye, hello

```

## 2.3.2 编写字符设备驱动程序

完成以上内容后, 请编写字符设备驱动程序, 通过内核 GPIO 库访问引脚, 能将 `write()` 送来的单个字符在矩阵上显示出来。

以下内容摘录自 <https://zhuanlan.zhihu.com/p/137636768>

Linux 系统将设备分为三大类: 字符设备、块设备和网络设备。字符设备是其中较为基础的一类, 它的读写操作需要一个字节一个字节的进行, 不能随机读取设备中的某一数据, 即要按照先后顺序。举例来说, 比较常见的字符设备有鼠标、键盘、串口等。

字符设备驱动所做的工作主要是添加、初始化、删除 `cdev` 结构体, 申请、释放设备号, 填充 `file_operations` 结构体中的功能函数, 比如 `open()`、`read()`、`write()`、`close()` 等。当我们创建一个字符设备时, 一般会在 `/dev` 目录下生成一个设备文件, Linux 用户层的程序就可以通过这个设备文件来操作这个字符设备。

???+ tip "参考资料"

1. [字符设备驱动]  
([https://doc.embedfire.com/linux/imx6/ls1's's's's's's's's'sdriver/zh/latest/linux\\_driver/character\\_device.html](https://doc.embedfire.com/linux/imx6/ls1's's's's's's's's'sdriver/zh/latest/linux_driver/character_device.html))
2. [Linux驱动篇(五)--字符设备驱动(一)]  
(<https://zhuanlan.zhihu.com/p/137636768>)
3. [GPIO 使用]([https://wiki.t-firefly.com/zh\\_CN/ROC-RK3568-PC-SE/driver\\_gpio.html#faqs](https://wiki.t-firefly.com/zh_CN/ROC-RK3568-PC-SE/driver_gpio.html#faqs))

**TASK6 编写字符设备驱动程序，通过内核GPIO库访问引脚，能将 write() 送来的单个字符在矩阵上显示出来。请给出源代码（需有详细注释）以及对关键部分的解释。将驱动编译并加载后，请编写C语言程序（或直接使用 shell 脚本）测试此驱动，并给出测试使用的程序、终端中的截图以及 LED 板显示的照片。**（20分）

由于完成了Bonus，此处一同展示对应代码。但是不对bonus的部分着重解释。

```
1 // 驱动程序
2 #include <linux/kernel.h>
3 #include <linux/module.h>
4 #include <linux/device.h>
5 #include <linux/stddef.h>
6 #include <linux/types.h>
7 #include <linux/fs.h>
8 #include <linux/cdev.h>
9 #include <linux/delay.h>
10 #include <linux/uaccess.h>
11 #include <linux/init.h>
12 #include <linux/gpio.h>
13 #include <linux/string.h>
14 #include <linux/slab.h>
15 #include <linux/time.h>
16 #include <linux/param.h>
17
18 // 定义设备名称
19 #define DEVICE_NAME "led_matrix"
20
21 //class声明内核模块驱动信息,使UDEV能够自动生成/dev下相应文件
22 static dev_t led_matrix_devno; //设备号
23 static struct class *led_matrix_class; // cdev所需结构
24 static struct cdev led_matrix_class_dev; // cdev用于在系统中注册字符设备
25
26 // 对应的引脚号
27 #define Din 4
28 #define CS 3
29 #define CLK 2
30 // 定义电平信号值
31 #define HIGH 1
32 #define LOW 0
33
34 // 定义led显示形状对应的各寄存器值，matrix为初始化默认形状
35 // 数字和字母对应表示来自 https://xantorohara.github.io/led-matrix-editor/ 给出的默认库
36 int matrix[8] = {0x99,0x66,0x5a,0xa5,0xa5,0x5a,0x66,0x99};
37 int digits[][8]={
38 {0x1c, 0x22, 0x22, 0x22, 0x22, 0x22, 0x22, 0x1c}, // 0
39 {0x08, 0x18, 0x08, 0x08, 0x08, 0x08, 0x08, 0x1c}, // 1
40 {0x1c, 0x22, 0x22, 0x04, 0x08, 0x10, 0x20, 0x3e}, // 2
41 {0x1c, 0x22, 0x02, 0x0c, 0x02, 0x02, 0x22, 0x1c}, // 3
42 {0x04, 0x0c, 0x14, 0x14, 0x24, 0x1e, 0x04, 0x04}, // 4
```



```

43 {0x3e, 0x20, 0x20, 0x3c, 0x02, 0x02, 0x22, 0x1c}, // 5
44 {0x1c, 0x22, 0x20, 0x3c, 0x22, 0x22, 0x22, 0x1c}, // 6
45 {0x3e, 0x24, 0x04, 0x08, 0x08, 0x08, 0x08, 0x08}, // 7
46 {0x1c, 0x22, 0x22, 0x1c, 0x22, 0x22, 0x22, 0x1c}, // 8
47 {0x1c, 0x22, 0x22, 0x22, 0x1e, 0x02, 0x22, 0x1c}, // 9
48 };
49 int letters[][8] = {
50 {0x3c, 0x42, 0x02, 0x3e, 0x42, 0x46, 0x3a, 0x00}, // a
51 {0xe0, 0x60, 0x60, 0x7c, 0x66, 0x66, 0xdc, 0x00}, // b
52 {0x00, 0x00, 0x78, 0xcc, 0xc0, 0xcc, 0x78, 0x00}, // c
53 {0x1c, 0x0c, 0x0c, 0x7c, 0xcc, 0xcc, 0x76, 0x00}, // d
54 {0x00, 0x00, 0x78, 0xcc, 0xfc, 0xc0, 0x78, 0x00}, // e
55 {0x38, 0x6c, 0x60, 0xf0, 0x60, 0x60, 0xf0, 0x00}, // f
56 {0x00, 0x00, 0x76, 0xcc, 0xcc, 0x7c, 0x0c, 0xf8}, // g
57 {0xe0, 0x60, 0x6c, 0x76, 0x66, 0x66, 0xe6, 0x00}, // h
58 {0x30, 0x00, 0x70, 0x30, 0x30, 0x30, 0x78, 0x00}, // i
59 {0x0c, 0x00, 0x0c, 0x0c, 0x0c, 0xcc, 0xcc, 0x78}, // j
60 {0xe0, 0x60, 0x66, 0x6c, 0x78, 0x6c, 0xe6, 0x00}, // k
61 {0x70, 0x30, 0x30, 0x30, 0x30, 0x30, 0x78, 0x00}, // l
62 {0x00, 0x00, 0xcc, 0xfe, 0xfe, 0xd6, 0xc6, 0x00}, // m
63 {0x00, 0x00, 0xf8, 0xcc, 0xcc, 0xcc, 0xcc, 0x00}, // n
64 {0x00, 0x78, 0xcc, 0xcc, 0xcc, 0xcc, 0x78, 0x00}, // o
65 {0x00, 0xdc, 0x66, 0x66, 0x7c, 0x60, 0xf0, 0x00}, // p
66 {0x00, 0x76, 0xcc, 0xcc, 0x7c, 0x0c, 0x1e, 0x00}, // q
67 {0x00, 0xdc, 0x76, 0x66, 0x60, 0xf0, 0x00, 0x00}, // r
68 {0x00, 0x7c, 0xc0, 0x78, 0x0c, 0xf8, 0x00, 0x00}, // s
69 {0x10, 0x30, 0x7c, 0x30, 0x30, 0x34, 0x18, 0x00}, // t
70 {0x00, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc, 0x76, 0x00}, // u
71 {0x00, 0xcc, 0xcc, 0xcc, 0xcc, 0x78, 0x30, 0x00}, // v
72 {0x00, 0xc6, 0xd6, 0xfe, 0xfe, 0x6c, 0x00, 0x00}, // w
73 {0x00, 0xc6, 0x6c, 0x38, 0x6c, 0xc6, 0x00, 0x00}, // x
74 {0x00, 0xcc, 0xcc, 0xcc, 0xcc, 0x7c, 0x0c, 0xf8}, // y
75 {0x00, 0xfc, 0x98, 0x30, 0x64, 0xfc, 0x00, 0x00}, // z
76 };
77
78 static struct timer_list timer; // 定时器指针
79
80 void setBitData(int data); // 传输1bit数据
81 void write_Max7219_byte(int data); // 传输一字节数据
82 void write_Max7219(int addr, int data); // 传输 两字节数据，即设置一个寄存器
83 void ShowLEDMatrix(int ch[]); // 根据传入的数组显示对应形状
84 void init_GPIO_Max7219(void); // 初始化LED矩阵
85
86 int checkQueue(void); // 检查数据队列是否为空
87 int addToQueue(char one); // 向队列添加数据
88 char PopFront(void); // 将队列的第一个取出
89 void HandleTimer(struct timer_list * name); // 定时器回调函数，定时显示传入数据
90
91 // 数据队列的数据结构，单个字符进行存储。

```

```

92 typedef struct charq* charqPtr;
93 struct charq {
94     char ch;
95     struct charq* next;
96 };
97 charqPtr ToShow = NULL;
98 charqPtr Tail = NULL;
99
100
101 int checkQueue(void)
102 {
103     return ToShow != NULL; // 返回数据是否不为空，为空返回0，不为空返回1
104 }
105 int addToQueue(char one)
106 {
107     charqPtr tmp = (charqPtr)kmalloc(sizeof(struct
108     charq), GFP_KERNEL); // 为结构申请空间
109     tmp->ch = one; //赋值
110     tmp->next = NULL;
111     if(ToShow == NULL){ //队列为空和不为空时的添加操作有所不同，为空添加要
        //把队头和队尾同时赋新值，
112         ToShow = tmp;
113         Tail = tmp;
114     }
115     else { // 不为空时仅需操作队尾。
116         Tail->next = tmp;
117         Tail = Tail->next;
118     }
119     return 0;
120 }
121 char PopFront(void)
122 {
123     charqPtr tmp = ToShow;
124     char tmpch = ToShow->ch;
125
126     if(ToShow->next != NULL){ // 不断将队首显示并移向下一个数据，
127         ToShow = ToShow->next;
128     }
129     else { // 其实队尾可以不管，此处怕有危险还是处理了
130         ToShow = NULL;
131         Tail = NULL;
132     }
133
134     kfree(tmp); // 释放已显示的数据空间。
135     return tmpch;
136 }
137 void HandleTimer(struct timer_list *name)
138 {
139     if(checkQueue()){ //检查是否为空
140         char ch = PopFront(); // 获取显示字符，对数字和字母分别处理

```

```

141     if(ch >= '0' && ch <= '9'){
142         ShowLEDMatrix(digits[ch-'0']);
143     }
144     else if(ch >= 'a' && ch <= 'z'){
145         ShowLEDMatrix(letters[ch-'a']);
146     }
147 }
148     mod_timer(&timer,jiffies + msecs_to_jiffies(500/*ms*/));// 设置
    下一次显示定时器。
149 }
150
151
152
153 // Max7219 引脚操作相关在上面有注释，在编写驱动时我们使用linux内核的
    引脚库对引脚进行操作，只是函数名换了，逻辑基本没变。
154 void setBitData(int data)
155 {
156     gpio_set_value(CLK, LOW);
157     gpio_set_value(Din, data);
158     gpio_set_value(CLK, HIGH);
159 }
160 void write_Max7219_byte(int data)
161 {
162     int i;
163     for(i = 0 ; i < 8 ; i++){
164         if(((data <<i) & 0x80) == 0x80){
165             setBitData(HIGH);
166         }
167         else {
168             setBitData(LOW);
169         }
170     }
171 }
172 }
173 void write_Max7219(int addr, int data)
174 {
175     gpio_set_value(CS, LOW);
176     write_Max7219_byte(addr);
177     write_Max7219_byte(data);
178     gpio_set_value(CS, HIGH);
179 }
180
181 void ShowLEDMatrix(int ch[])
182 {
183     int i=0;
184     for (i=0;i<8;i++){
185         write_Max7219(i+1,ch[i]);
186     }
187 }
188 void init_GPIO_Max7219(void)
189 {

```

```

190 int ret ;
191 // gpio_request()函数使用引脚，这是独占的，不允许其余程序再使用该引
    脚，实际只是进行了标记
192 ret = gpio_request(Din, "Din");
193 if (ret) {
194     printk(KERN_ERR "module: Failed to request GPIO %d, error
        %d\n", Din, ret);
195     return ;
196 }
197 ret = gpio_request(CS, "CS");
198 if (ret) {
199     printk(KERN_ERR "module: Failed to request GPIO %d, error
        %d\n", CS, ret);
200     return ;
201 }
202 ret = gpio_request(CLK, "Din");
203 if (ret) {
204     printk(KERN_ERR "module: Failed to request GPIO %d, error
        %d\n", CLK, ret);
205     return ;
206 }
207 gpio_direction_output(Din, 1); // 设置引脚方向 ，默认输出1.
208 gpio_direction_output(CS, 1); // 设置引脚方向 ，默认输出1.
209 gpio_direction_output(CLK, 1); // 设置引脚方向 ，默认输出1.
210
211 write_Max7219(0x09, 0x00); // 关闭译码模式
212 write_Max7219(0x0a, 0x00); // 设置led亮度 3 ,15最大
213 write_Max7219(0x0b, 0x07); // 设置扫描行数
214 write_Max7219(0x0c, 0x01); // 设置正常模式
215 write_Max7219(0x0f, 0x00); // 设置正常模式
216
217 ShowLEDMatrix(matrix);
218
219 }
220 static int dev_write(struct file *file, const char __user
    *buffer, size_t count, loff_t *ppos)
221 {
222
223     char ch[100]={0};
224     copy_from_user(ch,buffer,count); // 从用户空间将写入的字符拷贝下来
225     printk("get %s from user ,count %d ok\n",ch,count); // 输出得到的
        字符串
226     int i;
227     for(i=0;i<count;i++){
228         if ((ch[i] >= '0' && ch[i]<='9') || (ch[i] >= 'a' && ch[i]
            <='z') || (ch[i] >= 'A' && ch[i]<='Z')){
229             // ShowLEDMatrix(digits[(ch[i]-'0')]);
230             addToQueue(ch[i]); // 将得到的字符逐一加入队列中，不再直接显示。
231         }
232     }
233     return count;

```

```

234 }
235
236 //内核调用后的open操作
237 static int dev_open(struct inode *inode, struct file *filp)
238     // 由于设备并不独占，设备的打开操作不需要处理，此处仅显示信息的不同。
239 {
240     static int open_flag = 0;
241     printk("Open LED matrix .....!\n");
242     if(open_flag ==0){
243         open_flag =1;
244         printk("Open LED matrix success!\n");
245         return 0;
246     }
247     else{
248         printk("LED Matrix has opened!\n");
249     }
250     return 0;
251 }
252
253 //内核调用后的release操作
254 static int dev_release(struct inode *inode,struct file *file){
255     printk("LED Matrix has release!\n");
256     return 0;
257 }
258
259 //file_operations使系统的open,write等函数指针指向我们所写的dev_open
    等函数，
260 //这样系统才能够调用
261 static struct file_operations led_matrix_dev_fops = {
262     .owner      =    THIS_MODULE,
263     .write      =    dev_write,
264     .open       =    dev_open,
265     .release    =    dev_release,
266 };
267
268 //内核加载后的初始化函数.
269 static int __init led_matrix_init(void)
270 {
271     struct device *dev;
272     int major; //自动分配主设备号
273     major = alloc_chrdev_region(&led_matrix_devno,0,1,DEVICE_NAME);
274     // 内核注册设备
275     cdev_init(&led_matrix_class_dev, &led_matrix_dev_fops);
276     major = cdev_add(&led_matrix_class_dev,led_matrix_devno,1);
277     //注册class
278     led_matrix_class = class_create(THIS_MODULE,DEVICE_NAME);
279     // 创建设备，暴露至用户空间
280     dev = device_create(led_matrix_class, NULL, led_matrix_devno,
        NULL, DEVICE_NAME);
281
282     init_GPIO_Max7219();

```

```

283
284 timer_setup(&timer,HandleTimer,0);          /* 初始化定时
器 */
285 mod_timer(&timer,jiffies + msecs_to_jiffies(500/*ms*/)); /* 添加
并启动定时器 */
286
287 printk("led matrix module init ok!\n");
288
289 return 0;
290 }
291 //内核卸载后的销毁函数
292 void led_matrix_exit(void)
293 {
294     // 释放引脚
295     gpio_free(Din);
296     gpio_free(CS);
297     gpio_free(CLK);
298     device_destroy(led_matrix_class,led_matrix_devno); // 销毁设备
299     class_destroy(led_matrix_class); // 销毁class
300     cdev_del(&led_matrix_class_dev); // 销毁内核注册信息
301     unregister_chrdev_region(led_matrix_devno, 1);
302     printk("led matrix exit ok!\n");
303
304 }
305
306 module_init(led_matrix_init);
307 module_exit(led_matrix_exit);
308
309 MODULE_DESCRIPTION("Rasp Matrix Driver");
310 MODULE_LICENSE("GPL");
311

```

```

1 // 测试用程序。
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <sys/ioctl.h>
6 #include <sys/types.h>
7 #include <sys/stat.h>
8 #include <fcntl.h>
9 #include <sys/select.h>
10 #include <errno.h>
11 #include <string.h>
12 #define BUFSIZE 1
13 int main(void)
14 {
15     int fd,ret;
16     char ch[1];
17     char stdch[100];
18
19     fd = open("/dev/led_matrix",O_RDWR );//增加写权限

```



```

20     if(fd < 0)
21     {
22         printf("Error: open /dev/led_matrix error,%d !\n",fd);
23         return(1);
24     }
25
26     int stdinfd;
27     stdinfd = open("/dev/tty",O_RDONLY | O_NONBLOCK);//增加写权限
28     if(stdinfd < 0)
29     {
30         printf("Error: open /dev/tty error,%d !\n",stdinfd);
31         return(1);
32     }
33
34     printf("open ok!\n");
35     while(1){//循环接受字符
36         //ch[0] = getchar();
37         //ret = write(fd, ch, 1);
38
39
40         // scanf("%s",stdch);
41         read(stdinfd, stdch, 100);
42         ret = write(fd, stdch, strlen(stdch));
43         memset(stdch,0,100);
44         if(ret < 0)
45         {
46             printf("error\n");
47         }
48     }
49     return 0;
50
51 }

```

由于直接使用 `read()` 读取缓冲区，实际上字符后面还跟了 `\n`，但是经过驱动筛选不会显示。下面摘录部分显示

左边为测试代码输入，右边为dmsg内核信息输出。

```

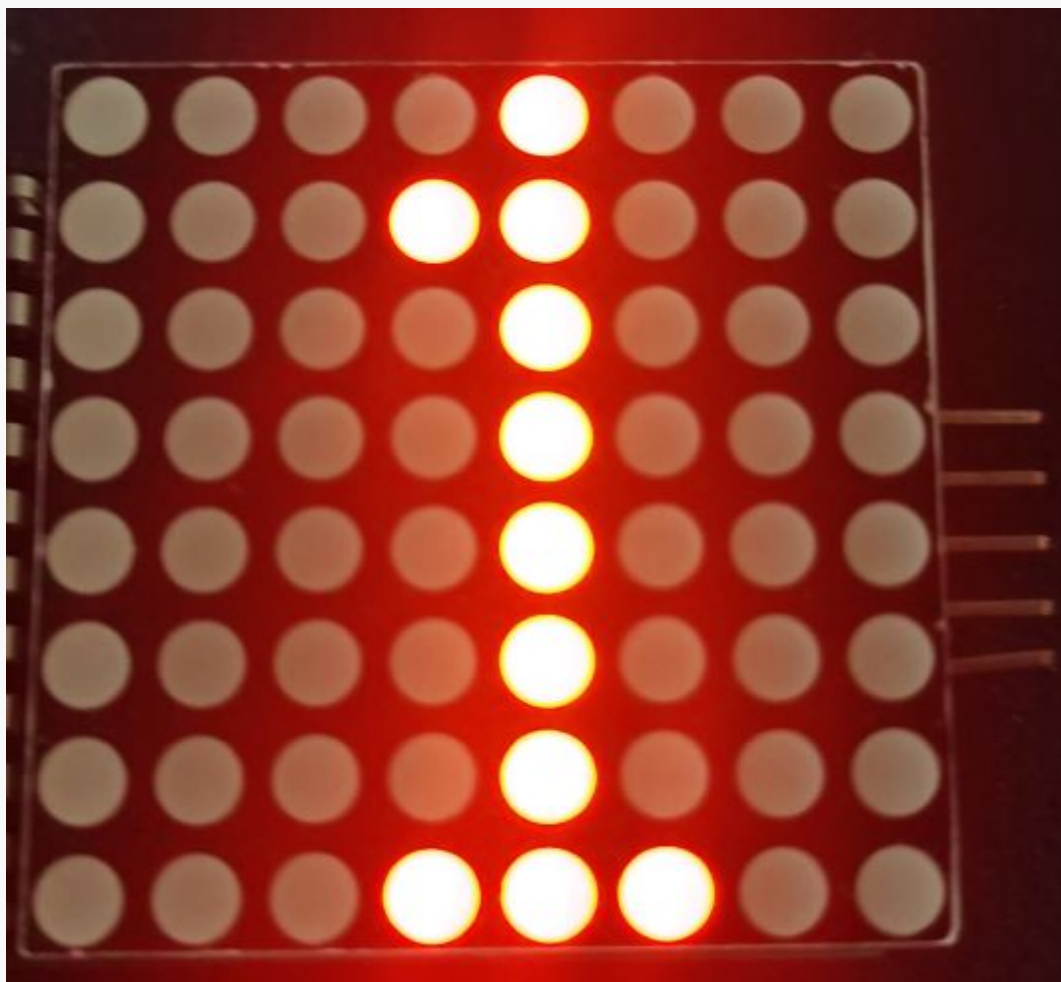
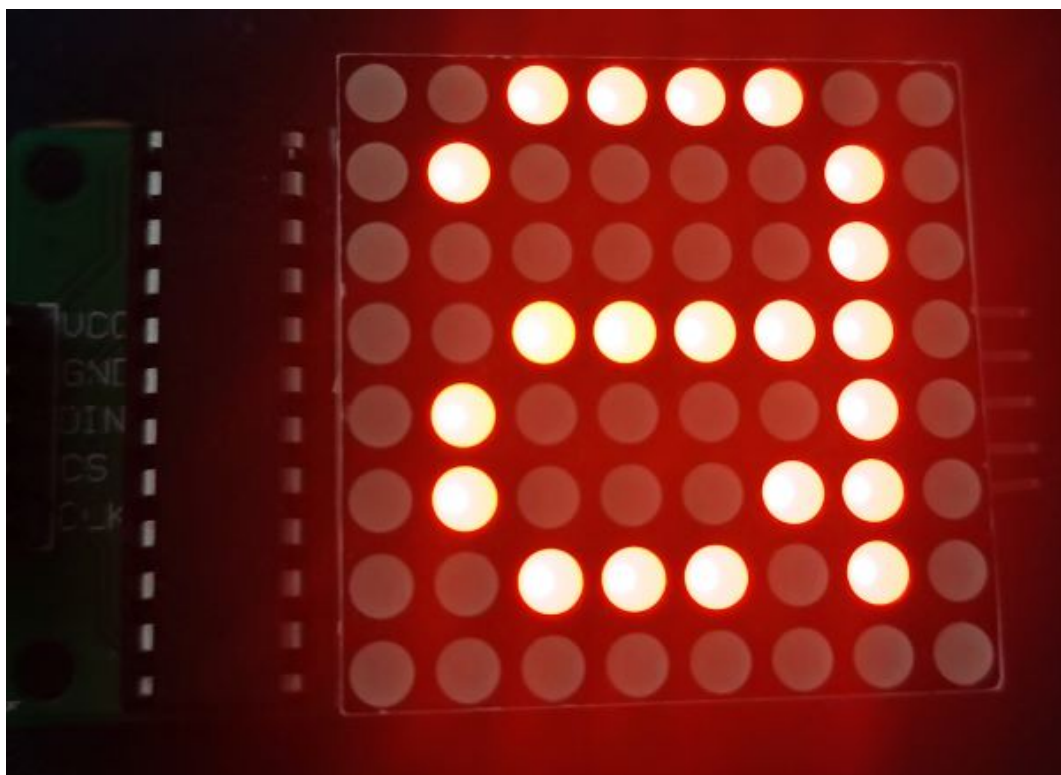
^C
yanhan@raspberrypi:~/Lab7yh/mod $ sudo ./a.out
open ok!
a
b
1
5
t
[ +0.000018] LED Matrix has opened!
[ +3.307701] get a
           from user ,count 2 ok
[Jun19 14:31] get b
           from user ,count 2 ok
[ +1.821893] get 1
           from user ,count 2 ok
[ +17.429500] get 5
           from user ,count 2 ok
[Jun19 14:32] get t
           from user ,count 2 ok

```

左边为测试代码输入，右边为dmsg内核信息输出。

```
^C
yanhan@raspberrypi:~/Lab7yh/mod $ sudo ./a.out
open ok!
a
b
1
5
t
```

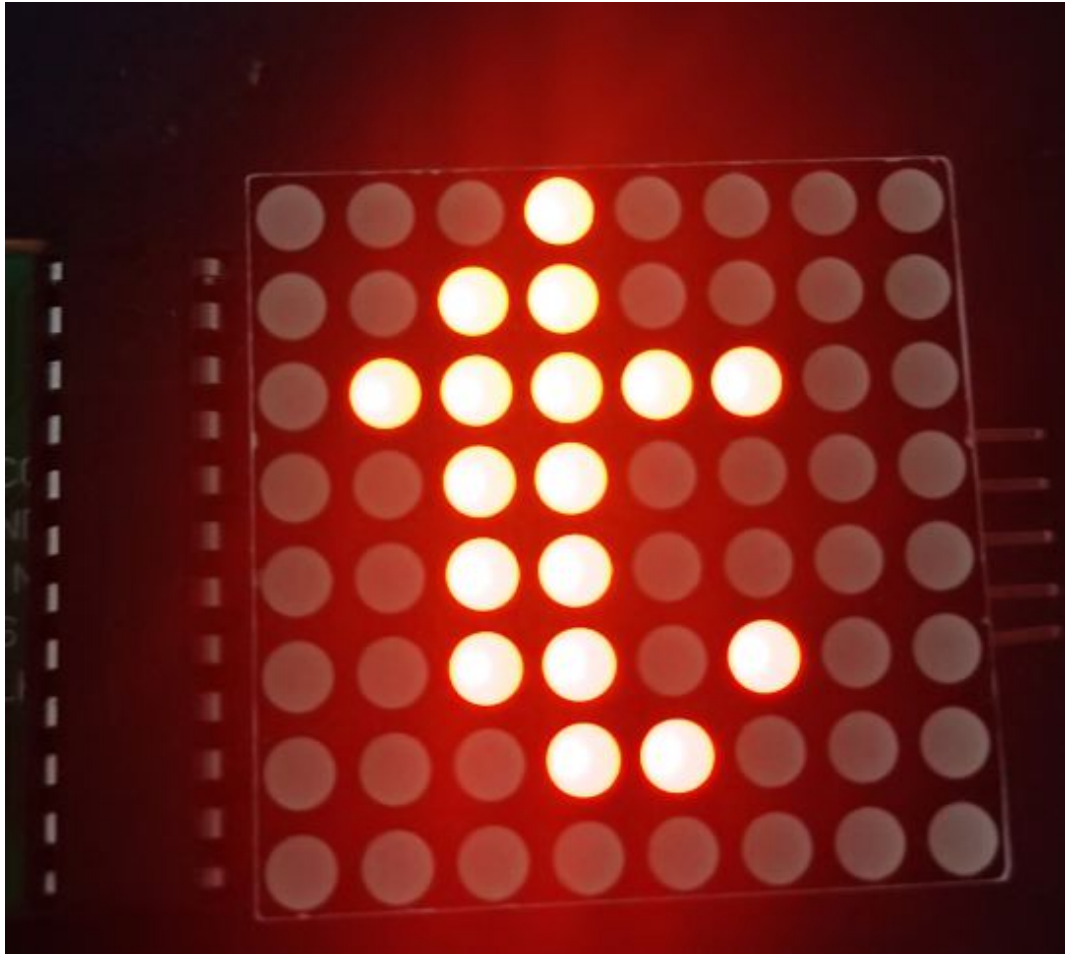
```
[ +0.000018] LED Matrix has opened!
[ +3.307701] get a
             from user ,count 2 ok
[Jun19 14:31] get b
             from user ,count 2 ok
[ +1.821893] get 1
             from user ,count 2 ok
[ +17.429500] get 5
             from user ,count 2 ok
[Jun19 14:32] get t
             from user ,count 2 ok
```



左边为测试代码输入，右边为dmsg内核信息输出。

```
^C
yanhan@raspberrypi:~/Lab7yh/mod $ sudo ./a.out
open ok!
a
b
1
5
t
```

```
[ +0.000018] LED Matrix has opened!
[ +3.307701] get a
              from user ,count 2 ok
[Jun19 14:31] get b
              from user ,count 2 ok
[ +1.821893] get 1
              from user ,count 2 ok
[ +17.429500] get 5
              from user ,count 2 ok
[Jun19 14:32] get t
              from user ,count 2 ok
```



### 3 驱动的应用

**TASK7** 编写 Linux 应用程序，能通过 MQTT 协议连接自己的 MQTT broker，将订阅收到的文字在 LED 矩阵上流动显示出来。要求给出源代码（有详细注释）以及对关键部分的解释。同时给出 LED 矩阵成功显示的现象照片。此 TASK 要求使用 TASK6 所编写的字符设备驱动程序，如果是通过 TASK2 的方式控制 LED，则此 TASK 分数减半。（10分）

由于使用的MQTT服务器软件为 `mosquitto`，因此此处我们同样使用 `mosquitto` 对应的C语言API。需要通过 `sudo apt-get install libmosquitto-dev` 安装对应的库。

方便使用多线程中断回调来接收信息，可在主线程进行管理（当然本程序仅）

```
1 #include <mosquitto.h>
2 #include <stdio.h>
3 #include <stdlib.h>
```

```

4  #include <unistd.h>
5  #include <fcntl.h>
6  #include <string.h>
7
8  static int running = 1;
9
10 void ShowOnLED(char * msg, int len);
11
12 void ConnectCallback(struct mosquitto *mosq, void* obj, int rc)
13 {
14     printf("Connect: %s\n", mosquitto_connack_string(rc));
15     if(rc != 0){ // 连接失败
16         fprintf(stderr, "Error Connect: Connect fail !!!\n");
17         mosquitto_disconnect(mosq); // 释放该连接
18     }
19
20     rc = mosquitto_subscribe(mosq, NULL, "DHT11", 0); // 订阅主题: mosq
    结构体, id, 主题, Qos
21     if(rc != MOSQ_ERR_SUCCESS){
22         fprintf(stderr, "Error subscribe: %s\n",
mosquitto_strerror(rc));
23         mosquitto_disconnect(mosq);
24     }
25     running = 1;
26 }
27
28 void Subscribecallback(struct mosquitto *mosq, void* obj, int mid,
int qos_count, const int *granted_qos)
29 {
30     int i;
31     bool have_subscription = false; // 订阅是否成功 flag.
32
33     for(i=0; i<qos_count; i++){ // 申请Qos
34         printf("on_subscribe: %d:granted qos = %d\n", i,
granted_qos[i]);
35         if(granted_qos[i] <= 2){
36             have_subscription = true;
37         }
38     }
39     if(have_subscription == false){
40         fprintf(stderr, "Error: All subscriptions
rejected.\n");
41         mosquitto_disconnect(mosq); // 订阅失败, 释放连接
42     }
43 }
44
45 void MessageCallback(struct mosquitto *mosq, void *obj, const
struct mosquitto_message *msg)
46 { // 接收消息回调
47
48     // 输出对应信息并显示LED字符

```

```

49     printf("***** Message Begin *****\n");
50     printf("Topic: %s  Qos%d  len:%d\n",msg->topic, msg->qos,msg-
>payloadlen);// 输出相关信息, 主题, Qos, 信息
51     printf("Msg: %s\n", (char*)msg->payload);
52
53     ShowOnLED((char*) msg->payload,msg->payloadlen);// 显示字符
54
55     printf("***** Message END *****\n\n");
56
57
58 }
59
60 void DisconnectCallback(struct mosquitto *mosq, void *obj, int
rc)
61 {
62     printf("Disconnect: disconnect_callback\n");
63     running = 0;
64 }
65
66 void ShowOnLED(char * msg, int len)
67 {
68     int fd, ret;
69
70     fd = open("/dev/led_matrix",O_RDWR | O_NONBLOCK);//增加写权限 ,
不阻塞
71     if(fd < 0)
72     {
73         fprintf(stderr, "Error Open: open /dev/led_matrix error!
fd:%d\n",fd);
74         return ;
75     }
76
77     ret = write(fd, msg, len); // 写入LED设备
78     if(ret < 0)
79     {
80         fprintf(stderr,"error to write msg to LED\n");
81         return ;
82     }
83     close(fd);
84 }
85
86
87
88
89 int main()
90 {
91     struct mosquitto *mosq;
92     int rc;
93
94     // 初始化mosquitto库
95     mosquitto_lib_init();

```

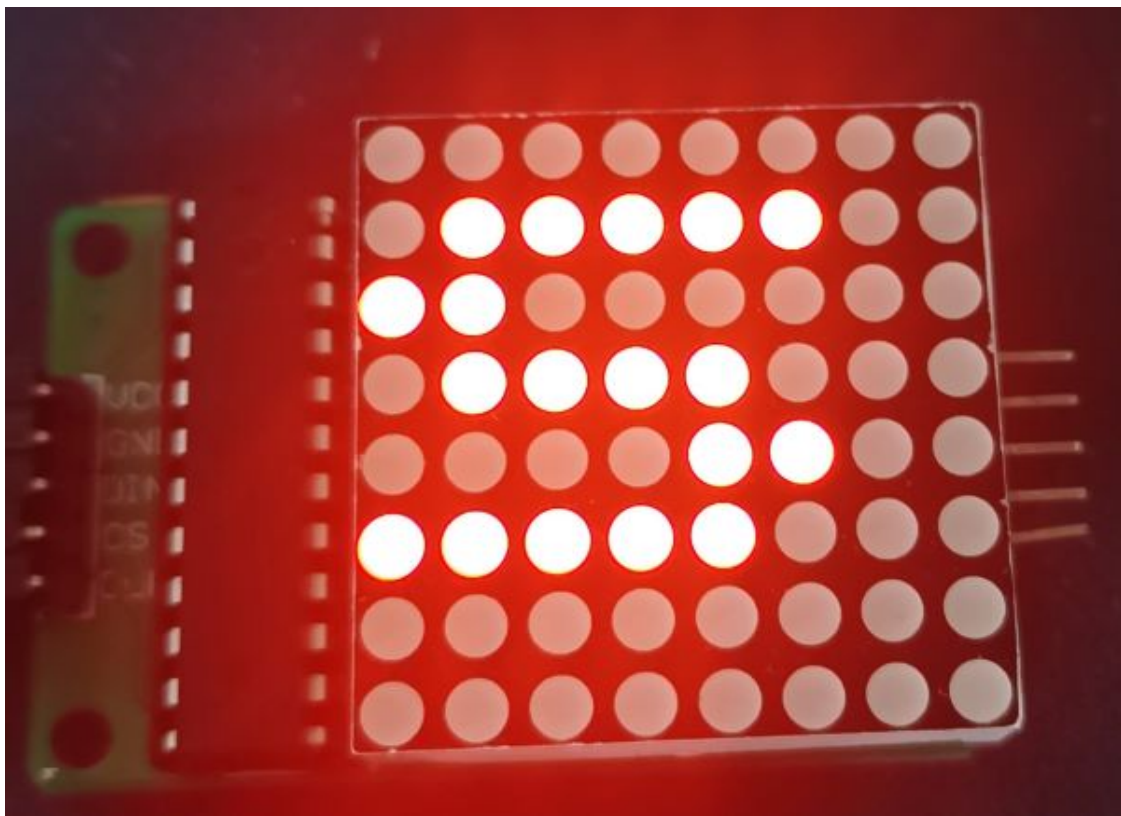
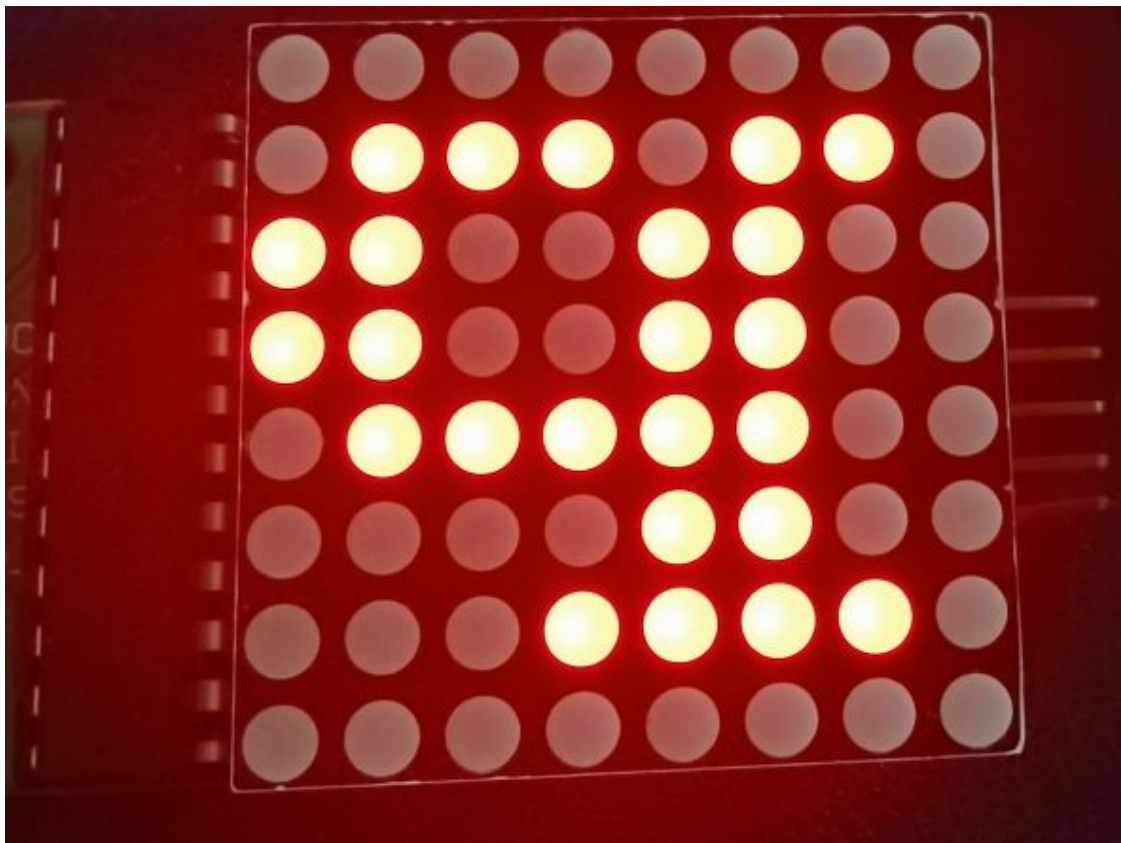
```
96
97 // 创建新的客户端实例。
98 mosq = mosquitto_new(NULL, true, NULL);
99 if(mosq == NULL){
100     fprintf(stderr, "Error: Not able to create mosquitto.\n");
101     return 1;
102 }
103
104 // 设置各操作对应回调函数。
105 mosquitto_connect_callback_set(mosq, ConnectCallback); // 连接回
    调
106 mosquitto_message_callback_set(mosq, MessageCallback); // 订阅回
    调
107 mosquitto_subscribe_callback_set(mosq, SubscribeCallback); // 接收
    消息回调
108 mosquitto_disconnect_callback_set(mosq, DisconnectCallback); //
    释放连接回调
109
110 mosquitto_username_pw_set(mosq, "yanhan", "yanhan"); // 设置服务器用
    户名和密码，否则不允许访问
111 rc = mosquitto_connect_async (mosq, "192.168.43.145", 1883,
    60); // 连接
112 if(rc != MOSQ_ERR_SUCCESS){
113     mosquitto_destroy(mosq);
114     fprintf(stderr, "Error connect main: %s\n",
    mosquitto_strerror(rc));
115     return 1;
116 }
117
118 // 异步循环
119 rc = mosquitto_loop_start(mosq); // 开启循环接收线程
120 if(rc != MOSQ_ERR_SUCCESS)
121 {
122     mosquitto_destroy(mosq);
123     fprintf(stderr, "Error loop main: %s\n",
    mosquitto_strerror(rc));
124     return 1;
125 }
126
127 // 开始循环
128 printf("Start mosquitto loop!\n");
129 while(running) // 此处仅管理程序结束
130 {
131     char ch;
132     scanf("%c", ch);
133     if(ch == 'q') break;
134 }
135 mosquitto_disconnect(mosq); // 释放该连接
136 mosquitto_loop_stop(mosq, 1) // 结束循环接收线程
137 // 结束后的清理工作
138 mosquitto_destroy(mosq);
```



手机发送信息:



部分显示:



**TASK8** 在 TASK7 的基础上，实现 Lab6 的 DHT-11 的数据发布到自己的 MQTT broker 上后显示在 LED 矩阵上。要求给出源代码（有详细注释）以及对关键部分的解释。同时给出 LED 矩阵成功显示的现象照片。（5分）

实际上同时运行task7 中代码和lab6对于dht的代码即可。

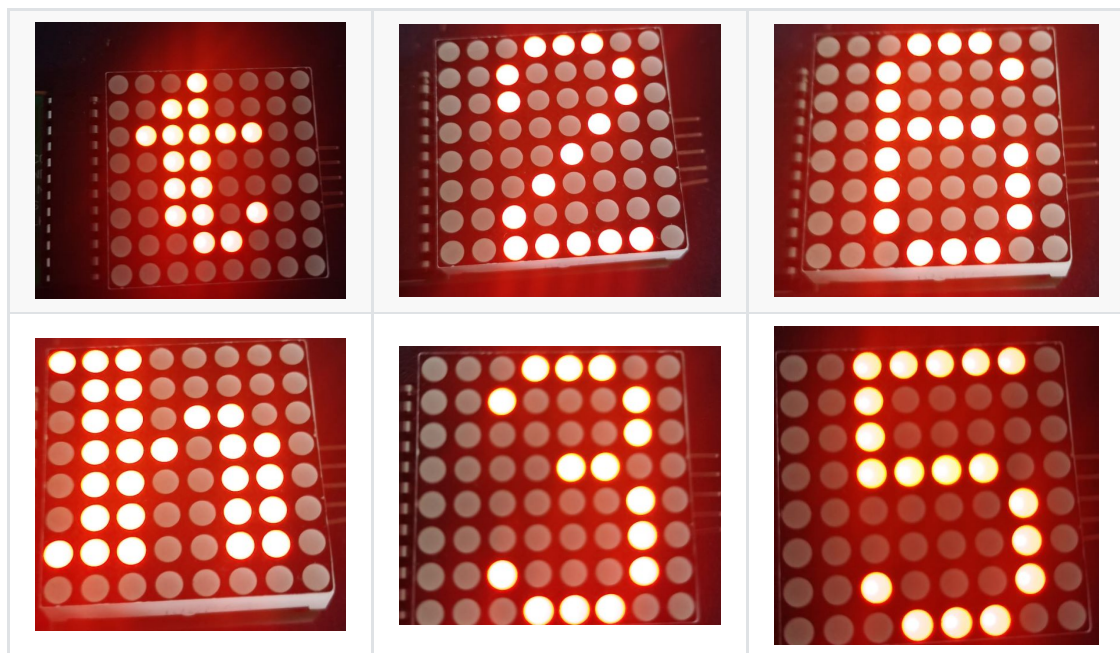
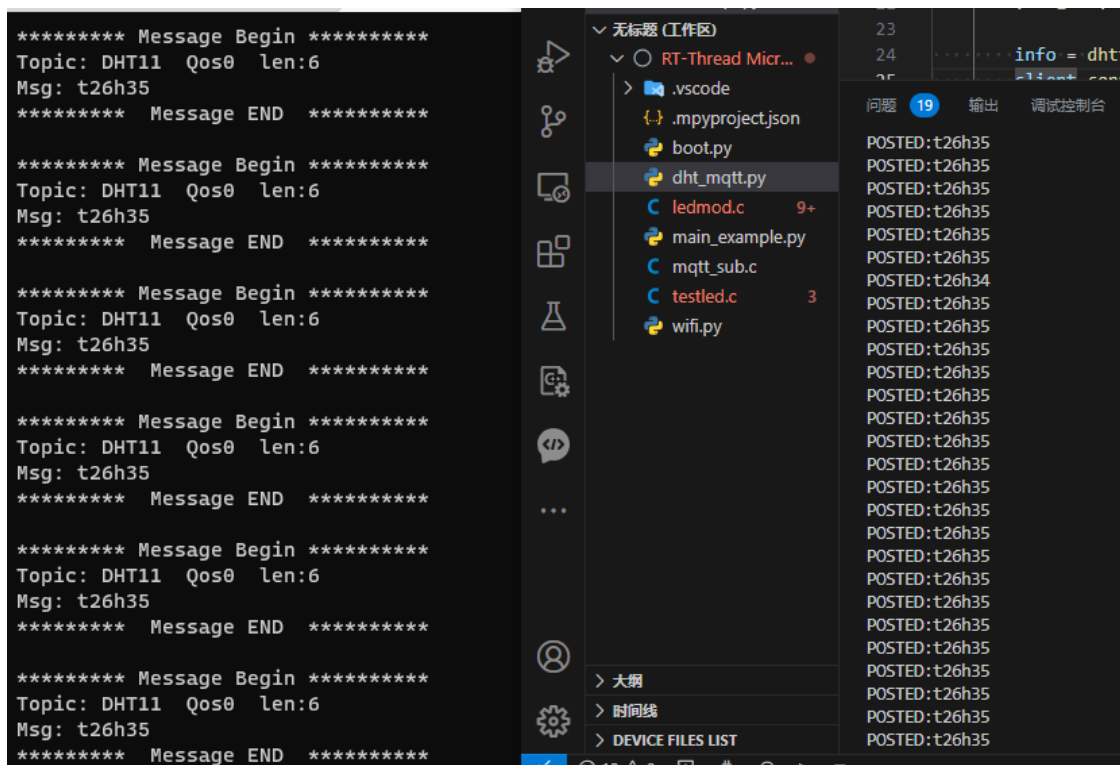
```
1 # lab6dht 发布代码，为了简化将发送的字符串缩小并仅支持字母和数字
2 # 导入模块
3 from machine import Pin
```

```

4  from umqtt.simple import MQTTClient
5  import time
6  import dht
7
8  def dhttest():
9      d = dht.DHT11(Pin(4))
10     d.measure()
11     return "t"+str(d.temperature())+" h"+str(d.humidity())# 返回温度和
    湿度信息字符串
12
13 def main():
14     print("welcome to RT-Thread MicroPython!")
15
16     led = Pin(2, Pin.OUT)
17     v = 1 # state of led, 根据led灯得知消息发送
18
19     client = MQTTClient("DHT11","192.168.43.145") # 设置mqtt服务器地址
    和客户端名字
20     topic = b"DHT11" # 设置订阅主题
21
22     while True:
23         led.value(v) # change state of led
24         v = 1 - v
25
26         info = dhttest()
27         client.connect() # 连接
28         client.publish(topic,info) # 发布消息
29         client.disconnect() # 断开连接
30         print("POSTED:"+info) # 终端打印输出信息
31
32         led.value(v)
33         v = 1 - v
34         time.sleep(5) # 延时
35
36 if __name__ == '__main__':
37     main()

```

终端显示:



## 4 拓展内容

拓展内容要求对驱动进行进一步细化，满足以下要求：

**BONUS1** 设备驱动程序能将 `write()` 送来的字符串以每个字母停留 500ms 的速度依次显示。  
(5分Bonus)

```

1 // 最初直接在write处理函数中显示，在显示完后添加睡眠500ms即可。
2
3 for(i=0;i<count;i++){
4     if ((ch[i] >= '0' && ch[i]<='9') || (ch[i] >= 'a' && ch[i]<='z')
5         || (ch[i] >= 'A' && ch[i]<='Z')){
6         ShowLEDMatrix(digits[(ch[i]-'0')]);
7         msleep(500);
8     }
9 }

```

**BONUS2** BONUS1 中的 `write()` 函数是非阻塞类型的，每500ms一个字符的显示是由内核定时器队列实现的。（5分Bonus）

直接在write中延时会导致多个程序写入时write无法及时响应，将write改为非阻塞有可能导致显示的错乱。因此我们使用定时器来完成字符的延时显示。

write不再直接显示字符，而是将获取的字符添加进一个待显示的队列中，这样大大加快了write的速度。

```

1 // 数据队列的数据结构，单个字符进行存储。
2 typedef struct charq* charqPtr;
3 struct charq {
4     char ch;
5     struct charq* next;
6 };
7 charqPtr ToShow = NULL;
8 charqPtr Tail = NULL;
9
10
11 int checkQueue(void)
12 {
13     return ToShow != NULL; // 返回数据是否不为空，为空返回0，不为空返回1
14 }
15 int addToQueue(char one)
16 {
17     charqPtr tmp = (charqPtr)kmalloc(sizeof(struct
18 charq), GFP_KERNEL); // 为结构申请空间
19     tmp->ch = one; //赋值
20     tmp->next = NULL;
21
22     if(ToShow == NULL){//队列为空和不为空时的添加操作有所不同，为空添加
23 要把队头和队尾同时赋新值，
24         ToShow = tmp;
25         Tail = tmp;
26     }
27     else { // 不为空时仅需操作队尾。
28         Tail->next = tmp;
29         Tail = Tail->next;
30     }
31 }

```

```

28     }
29     return 0;
30 }
31
32 char PopFront(void)
33 {
34     charqPtr tmp = ToShow;
35     char tmpch = ToShow->ch;
36
37     if(ToShow->next != NULL){ // 不断将队首显示并移向下一个数据，
38         ToShow = ToShow->next;
39     }
40     else { // 其实队尾可以不管，此处怕有危险还是处理了
41         ToShow = NULL;
42         Tail = NULL;
43     }
44
45     kfree(tmp); // 释放已显示的数据空间。
46     return tmpch;
47 }
48
49 // write
50 for(i=0;i<count;i++){
51     if ((ch[i] >= '0' && ch[i]<='9') || (ch[i] >= 'a' && ch[i]<='z')
52     || (ch[i] >= 'A' && ch[i]<='Z')){
53         // ShowLEDMatrix(digits[(ch[i]-'0')]);
54         addToQueue(ch[i]); // 将得到的字符逐一加入队列中，不再直接显示。
55     }
56 }

```

而在模块初始化同时我们也要初始化定时器。`timer`为全局变量。`HandleTimer`为回调函数。

```

1 timer_setup(&timer,HandleTimer,0); /* 初始化定时器
   */
2 mod_timer(&timer,jiffies + msecs_to_jiffies(500/*ms*/)); /* 添加并
   启动定时器 */

```

在回调函数中我们将需要显示的字符从队列中取出并显示，如果为空则不操作。

最后为了能够不断显示队列中的字符，回调函数最后我们需要通过`mod_timer`修改定时器的触发时间后再次添加进队列等待延时触发。



```

1 void HandleTimer(struct timer_list *name)
2 {
3     if(checkQueue()){ //检查是否为空
4         char ch = PopFront();// 获取显示字符，对数字和字母分别处理
5         if(ch >= '0' && ch <= '9'){
6             ShowLEDMatrix(digits[ch-'0']);
7         }
8         else if(ch >= 'a' && ch <= 'z'){
9             ShowLEDMatrix(letters[ch-'a']);
10        }
11    }
12    mod_timer(&timer,jiffies + msecs_to_jiffies(500/*ms*/)); // 设置
    下一次显示定时器。
13 }

```

## 5 讨论和心得

请认真填写本模块，若不填写或胡乱填写将酌情扣分，写明白真实情况即可。

请在此处填写实验过程中遇到的问题及相应的解决方式。

有关树莓派编译内核的相关博客大都比较遥远了，甄别信息花费了一点时间。

另外，由于docker突然无法连接网络，最初直接使用ubuntu 22.04 虚拟机进行交叉编译，一开始还是正常编译成功，程序也可以放到树莓派跑，后来不知道更新了什么，突然有 glibc 的版本问题，高版本虚拟机的 glibc 版本太高了导致树莓派无法运行，花大量时间更新 glibc 后也没解决问题。只能转而解决docker网络问题，使用firefly的ubuntu18.04容器来进行交叉编译。

另外，写驱动只能使用内核中的库时，关于 linux 内核的 gpio 库在网上的资料也良莠不齐，大都过时，因此探索 gpio 的操作也花了不少时间。

最后是关于 bonus 的非阻塞 write，一开始不太理解非阻塞和阻塞的区别，因为不断输入虽然由内核信息write不会一次性接收到，但是最终都会一一输出（这个是因为用户输入缓冲区的管理，多次回车输入都会一一响应），后来同时开启两个程序并非阻塞地从标准输入读取发现非阻塞写会导致不同程序的显示错乱甚至LED点阵熄灭。使用定时器显示后就没有这样的情况了。