嵌入式系统仿真实验第 12 讲

今天讲解嵌入式系统 ARM 环境 Linux 操作系统下的驱动程序、应用程序开发，实验做到现在，对于嵌入式系统原理这门课来说，基本就是最后一章的实验了。

1. 驱动程序开发

Linux 设备驱动程序为操作硬件提供良好内部接口，为应用程序提供了访问设备的机制。Linux 设备驱动分为：

字符设备：键盘、鼠标、串口

块设备：硬盘、Flash

网络接口：以太网

特定类型设备：audio 设备

系统调用和设备 I/O 关系见图 1 所示。

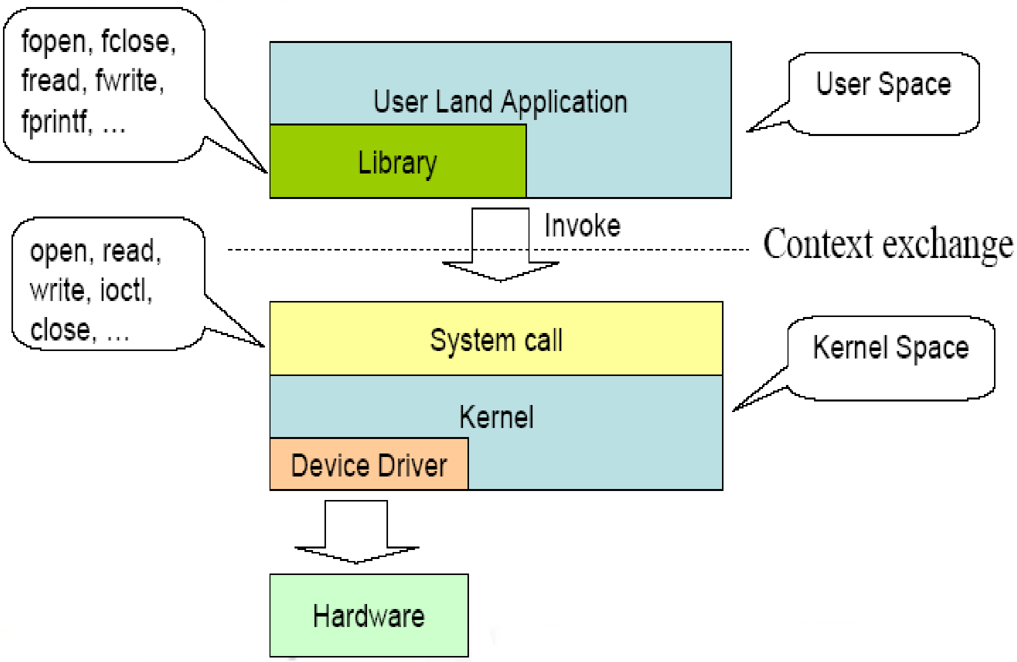

图 1 系统调用和设备 I/O

在做实验之前，请大家先回顾 insmod,rmmod,lsmod,modprobe,modinfo 这几个命令的用法，然后再弄明白驱动程序开发的步骤。这里我们以一个简单的字符

设备驱动程序为例来做实验。

```
/**
 * @file   ebbchar.c
 * @author Derek Molloy
 * @date   7 April 2015
 * @version 0.1
 * @brief   An introductory character driver to support the second article
of my series on
 * Linux loadable kernel module (LKM) development. This module maps to
/dev/ebbchar and
 * comes with a helper C program that can be run in Linux user space to
communicate with
 * this the LKM.
 * @see http://www.derekmolloy.ie/ for a full description and follow-up
descriptions.
 */

#include <linux/init.h>           // Macros used to mark up functions e.g.
__init __exit
#include <linux/module.h>         // Core header for loading LKMs into the
kernel
#include <linux/device.h>         // Header to support the kernel Driver
Model
#include <linux/kernel.h>         // Contains types, macros, functions for
the kernel
#include <linux/fs.h>             // Header for the Linux file system
support
#include <linux/uaccess.h>          // Required for the copy to user
function
#define  DEVICE_NAME "ebbchar"    ///< The device will appear at
/dev/ebbchar using this value
#define  CLASS_NAME  "ebb"        ///< The device class -- this is a
character device driver

MODULE_LICENSE("GPL");            ///< The license type -- this affects
available functionality
MODULE_AUTHOR("Derek Molloy");    ///< The author -- visible when you use
modinfo
MODULE_DESCRIPTION("A simple Linux char driver for the BBB");  ///< The
description -- see modinfo
MODULE_VERSION("0.1");            ///< A version number to inform users
```

```c
static int    majorNumber;                  ///< Stores the device number
-- determined automatically
static char   message[256] = {0};           ///< Memory for the string that
is passed from userspace
static short  size_of_message;              ///< Used to remember the size
of the string stored
static int    numberOpens = 0;              ///< Counts the number of times
the device is opened
static struct class*  ebbcharClass  = NULL; ///< The device-driver class
struct pointer
static struct device* ebbcharDevice = NULL; ///< The device-driver device
struct pointer

// The prototype functions for the character driver -- must come before
the struct definition
static int     dev_open(struct inode *, struct file *);
static int     dev_release(struct inode *, struct file *);
static ssize_t dev_read(struct file *, char *, size_t, loff_t *);
static ssize_t dev_write(struct file *, const char *, size_t, loff_t *);
static long     simple_ioctl(struct file *file, unsigned int cmd, unsigned
long num);

/** @brief Devices are represented as file structure in the kernel. The
file_operations structure from
 *  /linux/fs.h lists the callback functions that you wish to associated
with your file operations
 *  using a C99 syntax structure. char devices usually implement open,
read, write and release calls
 */
static struct file_operations fops =
{
   .open = dev_open,
   .read = dev_read,
   .write = dev_write,
   .unlocked_ioctl = simple_ioctl,
   .release = dev_release,
};

/** @brief The LKM initialization function
 *  The static keyword restricts the visibility of the function to within
this C file. The __init
 *  macro means that for a built-in driver (not a LKM) the function is
only used at initialization
```

```
 *  time and that it can be discarded and its memory freed up after that
point.
 *  @return returns 0 if successful
 */
static int __init ebbchar_init(void){
   printk(KERN_INFO "EBBChar: Initializing the EBBChar LKM\n");

   // Try to dynamically allocate a major number for the device -- more
difficult but worth it
   majorNumber = register_chrdev(0, DEVICE_NAME, &fops);
   if (majorNumber<0){
      printk(KERN_ALERT "EBBChar failed to register a major number\n");
      return majorNumber;
   }
   printk(KERN_INFO "EBBChar: registered correctly with major
number %d\n", majorNumber);

   // Register the device class
   ebbcharClass = class_create(THIS_MODULE, CLASS_NAME);
   if (IS_ERR(ebbcharClass)){             // Check for error and clean
up if there is
      unregister_chrdev(majorNumber, DEVICE_NAME);
      printk(KERN_ALERT "Failed to register device class\n");
      return PTR_ERR(ebbcharClass);        // Correct way to return an
error on a pointer
   }
   printk(KERN_INFO "EBBChar: device class registered correctly\n");

   // Register the device driver
   ebbcharDevice = device_create(ebbcharClass, NULL, MKDEV(majorNumber,
0), NULL, DEVICE_NAME);
   if (IS_ERR(ebbcharDevice)){            // Clean up if there is an
error
      class_destroy(ebbcharClass);         // Repeated code but the
alternative is goto statements
      unregister_chrdev(majorNumber, DEVICE_NAME);
      printk(KERN_ALERT "Failed to create the device\n");
      return PTR_ERR(ebbcharDevice);
   }
   printk(KERN_INFO "EBBChar: device class created correctly\n"); // Made
it! device was initialized
   return 0;
}
```

```c
/** @brief The LKM cleanup function
 *  Similar to the initialization function, it is static. The __exit macro notifies that if this
 *  code is used for a built-in driver (not a LKM) that this function is not required.
 */
static void __exit ebbchar_exit(void){
   device_destroy(ebbcharClass, MKDEV(majorNumber, 0));     // remove the device
   class_unregister(ebbcharClass);                          // unregister the device class
   class_destroy(ebbcharClass);                             // remove the device class
   unregister_chrdev(majorNumber, DEVICE_NAME);             // unregister the major number
   printk(KERN_INFO "EBBChar: Goodbye from the LKM!\n");
}

/** @brief The device open function that is called each time the device is opened
 *  This will only increment the numberOpens counter in this case.
 *  @param inodep A pointer to an inode object (defined in linux/fs.h)
 *  @param filep A pointer to a file object (defined in linux/fs.h)
 */
static int dev_open(struct inode *inodep, struct file *filep){
   numberOpens++;
   printk(KERN_INFO "EBBChar: Device has been opened %d time(s)\n", numberOpens);
   return 0;
}

/** @brief This function is called whenever device is being read from user space i.e. data is
 *  being sent from the device to the user. In this case is uses the copy_to_user() function to
 *  send the buffer string to the user and captures any errors.
 *  @param filep A pointer to a file object (defined in linux/fs.h)
 *  @param buffer The pointer to the buffer to which this function writes the data
 *  @param len The length of the b
 *  @param offset The offset if required
 */
static ssize_t dev_read(struct file *filep, char *buffer, size_t len, loff_t *offset){
```

```
   int error_count = 0;
   // copy_to_user has the format ( * to, *from, size) and returns 0 on
success
   error_count = copy_to_user(buffer, message, size_of_message);

   if (error_count==0){           // if true then have success
      printk(KERN_INFO "EBBChar: Sent %d characters to the user\n",
size_of_message);
      return (size_of_message=0);  // clear the position to the start and
return 0
   }
   else {
      printk(KERN_INFO "EBBChar: Failed to send %d characters to the
user\n", error_count);
      return -EFAULT;            // Failed -- return a bad address message
(i.e. -14)
   }
}

/** @brief This function is called whenever the device is being written
to from user space i.e.
 *  data is sent to the device from the user. The data is copied to the
message[] array in this
 *  LKM using the sprintf() function along with the length of the string.
 *  @param filep A pointer to a file object
 *  @param buffer The buffer to that contains the string to write to the
device
 *  @param len The length of the array of data that is being passed in
the const char buffer
 *  @param offset The offset if required
 */
static ssize_t dev_write(struct file *filep, const char *buffer, size_t
len, loff_t *offset){
   sprintf(message, "%s(%zu letters)", buffer, len);   // appending
received string with its length
   size_of_message = strlen(message);               // store the length
of the stored message
   printk(KERN_INFO "EBBChar: Received %zu characters from the user\n",
len);
   return len;
}

/** Simple example on how to create a IOCTL by mhs 2019.4.9
 *
```

```
 *
 */
static long simple_ioctl(struct file *file, unsigned int cmd, unsigned
long num)
{
    printk("cmd=%d num=%d\n",cmd,num);
    switch(cmd){
        case 1:
            printk("demo for cmd = 1\n");
            break;
        case 2:
            printk("demo for cmd = 2\n");
            break;
    }
    return 0;
}

/** @brief The device release function that is called whenever the device
is closed/released by
 *  the userspace program
 *  @param inodep A pointer to an inode object (defined in linux/fs.h)
 *  @param filep A pointer to a file object (defined in linux/fs.h)
 */
static int dev_release(struct inode *inodep, struct file *filep){
   printk(KERN_INFO "EBBChar: Device successfully closed\n");
   return 0;
}

/** @brief A module must use the module_init() module_exit() macros from
linux/init.h, which
 *  identify the initialization function at insertion time and the cleanup
function (as
 *  listed above)
 */
module_init(ebbchar_init);
module_exit(ebbchar_exit);
```

这是从网上直接找的一个经典的字符设备驱动程序：ebbchar.c。

驱动测试程序 testebbchar.c:

```
/**
 * @file   testebbchar.c
 * @author Derek Molloy
 * @date   7 April 2015
```

```
 * @version 0.1
 * @brief  A Linux user space program that communicates with the ebbchar.c
LKM. It passes a
 * string to the LKM and reads the response from the LKM. For this example
to work the device
 * must be called /dev/ebbchar.
 * @see http://www.derekmolloy.ie/ for a full description and follow-up
descriptions.
*/
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
#include<fcntl.h>
#include<string.h>
#include<unistd.h>

#define BUFFER_LENGTH 256             ///< The buffer length (crude but
fine)
static char receive[BUFFER_LENGTH];     ///< The receive buffer from the
LKM

int main(){
   int ret, fd;
   char stringToSend[BUFFER_LENGTH];
   printf("Starting device test code example...\n");
   fd = open("/dev/ebbchar", O_RDWR);            // Open the device with
read/write access
   if (fd < 0){
      perror("Failed to open the device...");
      return errno;
   }
   printf("Type in a short string to send to the kernel module:\n");
   scanf("%[^\n]%*c", stringToSend);                // Read in a string
(with spaces)
   printf("Writing message to the device [%s].\n", stringToSend);
   ret = write(fd, stringToSend, strlen(stringToSend)); // Send the string
to the LKM
   if (ret < 0){
      perror("Failed to write the message to the device.");
      return errno;
   }

   printf("Press ENTER to read back from the device...\n");
   getchar();
```

```c
  printf("Reading from the device...\n");
  ret = read(fd, receive, BUFFER_LENGTH);        // Read the response from the LKM
  if (ret < 0){
    perror("Failed to read the message from the device.");
    return errno;
  }
  printf("The received message is: [%s]\n", receive);
  printf("Test ioctl.\n");
  int i;
  for(i=0;i<2;i++){
  int cmd = i;
  printf("The received cmd is: [%d]\n", cmd);
  printf("The received cmd num is: [%d]\n", i+10);
  ioctl(fd,i,i+10);
  }

  close(fd);
  printf("End of the program\n");
  return 0;
}
```

程序很简单，所以我在实验课就不额外讲。大家对照我们课堂讲的原理，看看就很快知道了，这是一个很简单的例子。他原来的 Makefile 如下：

```makefile
obj-m+=ebbchar.o

all:
	make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
	$(CC) testebbchar.c -o test
clean:
	make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean
	rm test
```

这是在 PC 环境下写的驱动，我们可以直接在电脑上编译测试就可以，但现在我们要用 arm-none-linux-gnueabi-交叉工具链去编译，所以这个 Makefile 需要改，怎么改，我直接上代码：

```makefile
obj-m +=ebbchar.o

CC = arm-none-linux-gnueabi-gcc
LD = arm-none-linux-gnueabi-ld
EXEC = testebbchar
```

```
OBJS = testebbchar.o

CFLAGS +=
LDFLAGS +=

all:
    make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- -C
/home/mhs/linux-3.2/ M=$(PWD) modules
    $(CC) $(LDFLAGS) -o $(EXEC) $(EXEC).c $(LDLIBS$(LDLIBS_$@))  -static
    cp $(EXEC) /home/mhs/busybox-1.28.1/_install/home/examples/
    cp ebbchar.ko /home/mhs/busybox-1.28.1/_install/home/examples/

clean:
    make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- -C
/home/mhs/linux-3.2/ M=$(PWD) clean
    -rm -f $(EXEC)
```

大家看看这个区别在哪，区别在编译器是 arm，CPU 架构是 arm，以及链接需要包含的头文件、库文件在我们上节课下载解压的 linux-3.2 目录下。其他没有什么区别，编译完成后我把他们拷贝到上次课做的 busybox 根文件裁剪安装目录里面，我在根目录建了个 home 目录，home 目录下又建了个 examples 目录，准备把这次课的所有程序放这里去打包，生成根目录。

注意，这里开始我们需要把执行文件链接静态库-static，这是由于我们在裁剪内核的时候选的用户执行程序需要静态链接，否则，会在模拟运行的时候，出现找不到执行文件的错误。

2. 应用程序开发

我这次另外还写了三个实验：helloworld 不需要讲解，thread 线程实验需要注意链接-lpthread，framebuffer 实验，我们操作了/dev/fb0 这个 framebuffer，大家如果记得前面我们做过无操作系统时候的 LCD 驱动实验，，这里我们有操作系统了，所以我们只需对 fb0 这个字符设备进行读写和 IO 操作即可。上次我们显示的是 24 位 BMP 彩图，这次由于仿真的 fb0 设备是 640x480x16bpp，所以我们需要将 24 位位图数据转为 RGB565 格式，才能争取转换，格式转换部分代码如下：

```
b = *pp; g = *(pp+1); r = *(pp+2); // BRG values
```

```
        r = r >> 3; g = g >>2; b= b >> 3;
        pixel = (r<<11) | (g<<5) | b; // pixel value
        *(((unsigned short *)fbp)+i*WIDTH + j) = pixel; // write to frame
buffer
        //*((unsigned short int*)(fbp + +i*WIDTH + j)) = pixel;
        pp += 3; // advance pp to next pixel
```

保留 8 位红色的高 5 位，8 位绿色的高 6 位，8 位蓝色的高 5 位，组合成 RGB565 的格式即可正确显示。

我在源代码顶层目录里写个 makefile，一次性完成 make 工作：

```
all clean:
    @for subdir in helloworld thread framebuffer  ebbchar;\
    do\
        #echo $$subdir;\
        if test -d $$subdir;\
        then\
        echo making $@ in $$subdir;\
        (cd $$subdir && make $@) || exit 1;\
        fi;\
    done
```

3. 测试运行

生成的文件拷贝到 busybox 的_install 目录，然后我们执行上一次生成 flash.bin 的过程，只是这一次考虑到根目录里我们加了很多文件，所以，flash.bin 我们预留 8M，然后重新生成 flash.bin:

```
mkimage -A arm -C none -O linux -T ramdisk -d rootfs.img.gz -a 0x00800000
-e 0x00800000 rootfs.uimg
dd if=/dev/zero of=flash.bin bs=1 count=8M
dd if=u-boot.bin of=flash.bin conv=notrunc bs=1
dd if=zImage.uimg of=flash.bin conv=notrunc bs=1 seek=2M
dd if=rootfs.uimg of=flash.bin conv=notrunc bs=1 seek=4M
```

然后我们上一次一样进行仿真运行。

qemu-system-arm -M versatilepb -serial stdio -kernel flash.bin

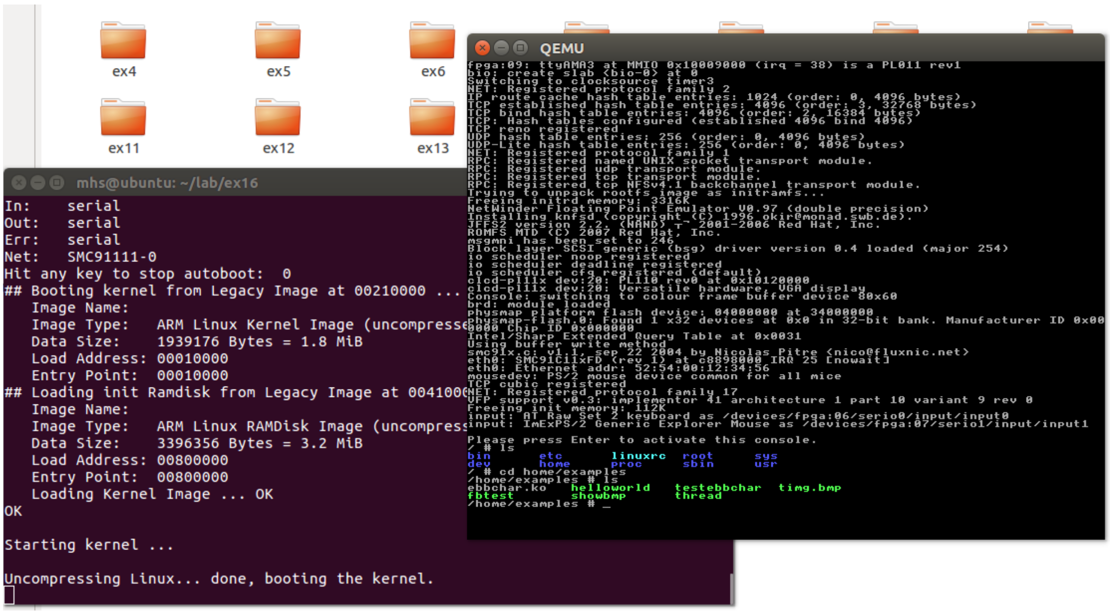

图 2 仿真运行

可以看到根目录下有 home/examples 目录，里面有我们生成的几个测试程序，我们来一一测试运行：

./helloworld

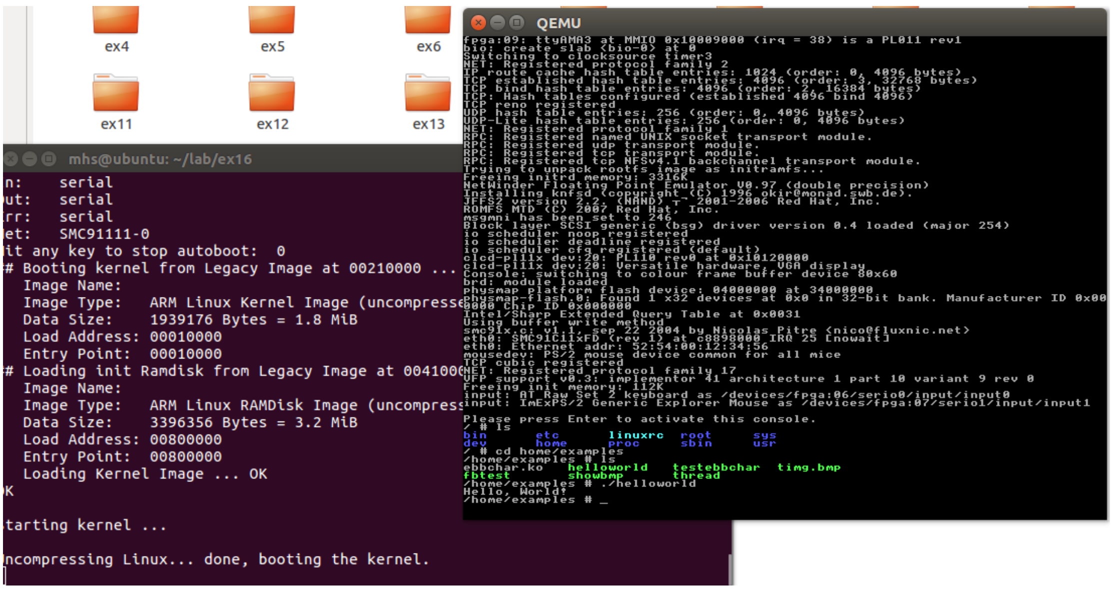

图 3 打印 helloworld

./thread

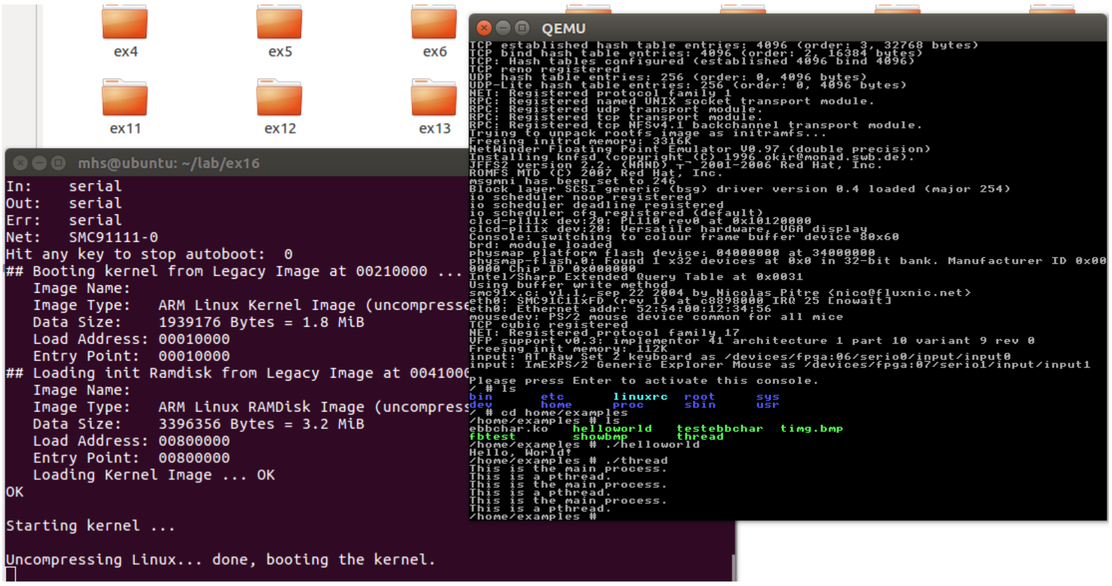

图 4 线程实验

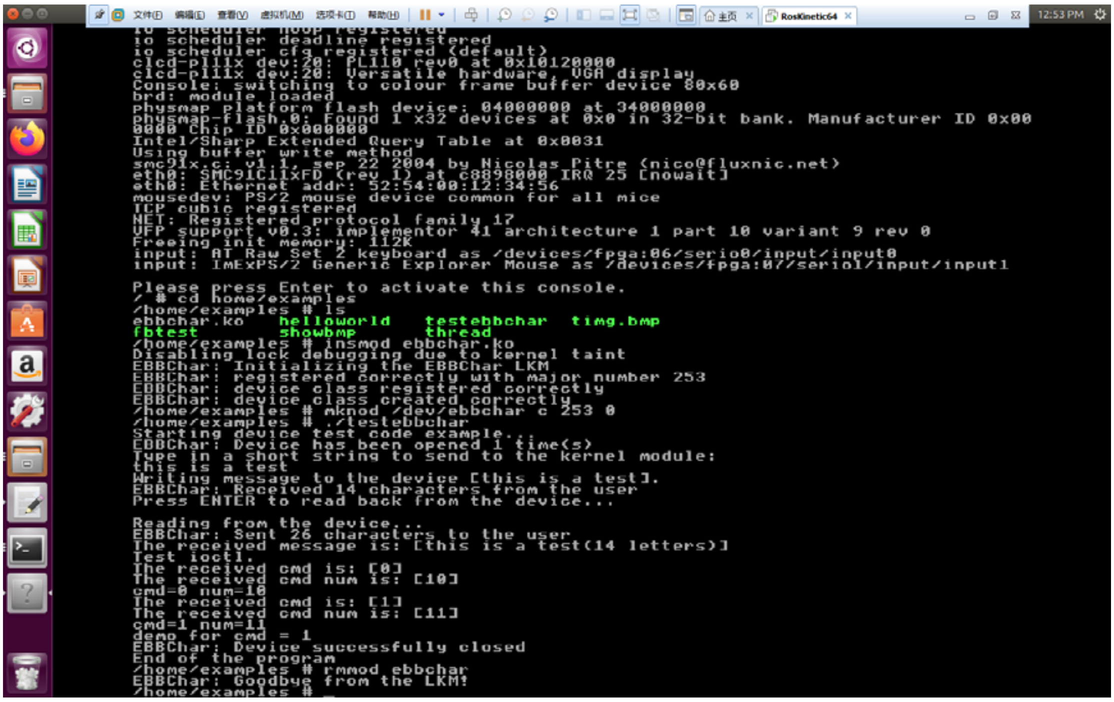

图 5 字符驱动实验

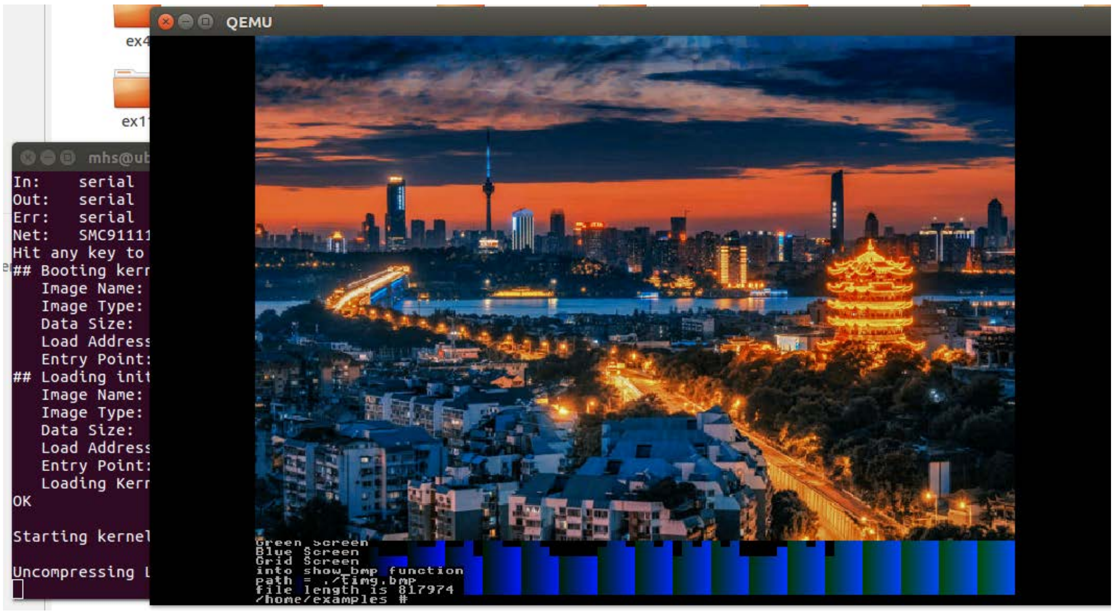

图 6 framebuffer 实验

到了这里，我们可以体会到现在我们有操作系统了，事情变得好办了很多，库也可以用了，能编出什么样的应用现在就看软件工程师了。

武汉快解封了，我们这门课的仿真实验也写完了，期望大家能掌握嵌入式系统开发的基本原理与技能，返校后再进行实物实验。

References

- Using U-Boot and Flash emulation in QEMU
- U-boot for ARM on QEMU
- Debugging Linux systems using GDB and QEMU
- Virtual Development Board
- Busybox for ARM on QEMU
- Part A. QEMU 使用手册