注：实验报告左侧装订

《计算机系统综合设计》报告

目录

1 设计目标

本设计旨在综合运用所学过的计算机硬件和软件知识，独立完成软件代码的编写，计算机系统外部接口硬件电路的设计，以及软件和硬件的协同调试，从而掌握计算机系统的硬件和软件的各个组成要素，主要包括以下内容:

使用汇编语言编写代码实现数据结构中的算法，并通过协同仿真和调试验证该算法的正确性

使用汇编语言编写 GPIO 驱动和控制的代码，实现检测按键的输入和对 LED 灯的控制

熟悉龙芯 1B 处理器的串口原理和功能，并使用 C 语言编写代码，实现 printf 和 scanf 的串口重定位

在龙芯 1B 处理器上部署 Linux 系统，并实现一个具体应用程序的开发

2 设计实现

2.0 分析启动引导代码

先在 Id.script 函数中找到入口

start.S × ld.script × cache.S × bsp_start.S ×

```
1  /*
2   * ld.script
3   */
4
5  _RamSize = DEFINED(_RamSize) ? _RamSize : 64M;
6  _StackSize = DEFINED(_StackSize) ? _StackSize : 0x4000; /* 16k */
7
8  OUTPUT_ARCH(mips)  //
9
10 ENTRY(start)        //入口，跳转到start.s
11
12 MEMORY
13 {
14     ram (wx) : ORIGIN = 0x80000000, LENGTH = 64M
15 }
16
```

然后看 Start.s

看 Start.s 的结构，主要有三个函数，分别是系统启动功能的 FRAME(start, sp, 0, ra),系统退出功能的 FRAME(_sys_exit,sp,0,ra)，获取内容大小功能的 FRAME(get_memory_size,sp,0,ra)。

先看系统启动函数 FRAME(start, sp, 0, ra)

```
//初始化C0
FRAME(start, sp, 0, ra)
    .set    noreorder

    mtc0    zero, C0_SR                     /* 清除 IntMsks/ 内核/禁用模式 */
    nop
    li      v0, CAUSE_DC
    mtc0    v0, C0_CAUSE                     /* 清除软件中断 */
    nop

    lui     v0, 0xFFFF
    mtc0    v0, C0_COMPARE                   /* 可能是 COUNT/COMPARE 中断，清除它。
    nop
```

然后是初始化一些寄存器

```
//初始化寄存器
#if (__mips_hard_float)
/* check to see if a fpu is really plugged in
 */
    li      t3,   0xAAAA5555                /* put a's and 5's in t3    */
    mtc1    t3,   fp0                       /* try to write them into fp0   */
    mtc1    zero, fp1                       /* try to write zero in fp  */
    mfc1    t0,   fp0
    mfc1    t1,   fp1
    nop
    bne     t0, t3, 1f                      /* branch if no match  */
    nop
    bne     t1, zero, 1f                    /* double check for positive id   */
    nop
/* We have a FPU. clear fcsr
 */
    ctc1    zero, fcr31
    j       2f                              /* status register already correct  */
    nop
1:
    li      v0, 0x0                         /* clear ERL and disable FPA */
    mtc0    v0, C0_SR                       /* reset status register */
    nop
2:
#endif
```

设置系统堆栈

```
108
109 /* Set system stack
110  */
111     la      t0, _stack_end
112     sub     t0, t0, (4 * 4)                 /* XXX overhead  */
113     move    sp, t0
114     nop
115
```

设置全局内存大小，会跳转到 set_memory_size()函数

```
/* Set memory size global
 */
    la      a0, _RamSize
    jal     set_memory_size
    nop
```

跳转到 config_cache 确定 D&I 缓存大小

```
/* determine size of D & I caches, In "Idtmem.S"
 */
    jal     config_cache
    nop
```

填充内存配置结构

```
/* fill memory config struct
 */
    la      a0, memory_cfg_struct
    jal     get_memory_conf
    nop
```

初始化 cache

```
/* Initialize cahce
 */
    jal     flush_cache
    nop
```

启用 cache

```
/* Enable cache
 */
    mfc0    v0, C0_CONFIG, 0                /* set K0 "cacheable noncoherent" mode */
    and     v0, v0, ~CFG0_K0
    or      v0, v0, CFG_C_CACHABLE          /* = 0x03 */
    mtc0    v0, C0_CONFIG, 0
    nop
```

清除 TLB

```
/* Clear Translation Lookaside Buffer (TLB)
 */
    jal     init_tlb                        /* clear the tlb */
    nop
```

CPU 初始化结束，准备启动内核 跳转到 bsp_start

```
/* End of CPU initialization, ready to start kernel
 */
    move    a0, zero                        /* Set argc passed to main */
    jal     bsp_start
    nop
```

内核已关闭，跳转至_sys_exit()

```
/* Kernel has been shutdown, jump to the "exit" routine
 */
    jal     _sys_exit
    move    a0, v0                          # pass through the exit code

1:
    beq     zero, zero, 1b
    nop

    .set    reorder
ENDFRAME(start)
```

最后是 bsp_start.c

CPU 初始化后跳转到 bsp_start，再进行初始化 isr 表，控制台，代码，启用所有中断，FPU

```
/
void bsp_start(void)
{
    mips_interrupt_disable();

    /*
     *安装 exec vec。 数据<==>指令必须使用K1地址
     */
    memcpy((void *)K0_TO_K1(T_VEC), except_common_entry, 40);
    memcpy((void *)K0_TO_K1(C_VEC), except_common_entry, 40);
    memcpy((void *)K0_TO_K1(E_VEC), except_common_entry, 40);

    mips_init_isr_table();          /* 初始化isr表 */

    console_init(115200);           /* 初始化控制台 */

    Clock_initialize();             /* 初始化代码 */

    /*
     * 启用所有中断，FPU
     */
#if __mips_hard_float
    mips_unmask_interrupt(SR_CU1 | SR_IMASK | SR_IE);
#else
    mips_unmask_interrupt(SR_IMASK | SR_IE);
#endif

    /* 转到主函数 */
    main();
```

最后跳转到主函数 main()。

2.1 算法的汇编语言实现

2.1.1 设计方案

本实验编写汇编语言对 10 个数进行冒泡排序。

设计思路为将数组放入 v0 寄存器中，通过 v0 的头地址和偏移量来实现寻址。根据 C 语言冒泡排序算法(见下图)来进行两层循环，将当前 str[i]与在它之后的数进行比较，如果当前值比下一个值大则进行交换，两层循环完成后，排序结果在 v0 寄存器中查看。

```
/*
int str[10] = {23,84,13,92,32,64,57,99,74,19};
    for (i = 0; i < 10 ; i++)
    {
        for (j = i + 1; j < 10; j++)
        {
            if (str[i] > str[j])
            {
                t = str[i];
                str[i] = str[j];
                str[j] = t;
            }

        }
    }
/*
```

2.1.2 设计实现

首先，创建一个 10 个数的数组

```
1.  array: .byte 23,84,13,92,32,64,57,99,74,19  //数组赋值
```

接着，将数组存入 v0 寄存器中，将外循环次数和内循环次数分别储存到 s0,s1 中

```
1.  .text
2.      la v0,array      //将数组头地址放进 v0
3.      addi s0,zero,10  //外循环次数为 10 储存到 s0
4.      addi s1,zero,9   //内循环次数为 9 储存到 s1
5.      move a1,s1       //将 s1->a1
```

然后将编写冒泡排序算法的核心，先找到当前相邻的两数，然后比较大小，如果前者比后者大则两数交换。

```
1:
    lbu t1,0(v0)    //提取 array[0] 加载到 t1
    lbu t2,1(v0)    //提取 array[1] 加载到 t2
    sltu a0,t1,t2   //如何 t1 < t3 则为 1 反之
    bgtz a0,2f      //如果 a0 大于 1 则跳转 2 继续  反之进行交换
    nop
    sb t1,1(v0)    //将 array[1]移到 t1 中
    sb t2,0(v0)    //将 array[0]移到 t2 中
```

编写两层循环的运行过程，即算法执行的过程

```
2:
    addi v0,v0,1    //移到下一个  即 array[i+1]
    addi s1,s1,-1   //同时内循环次数-1
    bgtz s1,1b      //如果 s1 > 0 则跳转到 1
    nop
    addi s0,s0,-1   //外循环次数减 1
    la v0,array     //将数组头地址保存在 v0 准备下次排序
    addi a1,a1,-1   //a1-1
    move s1,a1      //a1->s1
    bgtz s0,1b      //如果 s0 > 0 则跳转到 1
    nop
```

完整代码如下图

```
.data

array: .byte 23,84,13,92,32,64,57,99,74,19  //数组赋值


.text
    la v0,array     //将数组头地址放进v0
    addi s0,zero,10  //外循环次数为10 储存到s0
    addi s1,zero,9   //内循环次数为9 储存到s1
    move a1,s1      //将s1->a1

1:
    lbu t1,0(v0)    //提取array[0] 加载到t1
    lbu t2,1(v0)    //提取array[1] 加载到t2
    sltu a0,t1,t2   //如何t1 < t3 则为1 反之
    bgtz a0,2f      //如果a0大于1 则跳转2 继续  反之进行交换
    nop
    sb t1,1(v0)    //将array[1]移到t1中
    sb t2,0(v0)    //将array[0]移到t2中

2:
    addi v0,v0,1    //移到下一个  即array[i+1]
    addi s1,s1,-1   //同时内循环次数-1
    bgtz s1,1b      //如果s1 > 0则跳转到1
    nop
    addi s0,s0,-1   //外循环次数减1
    la v0,array     //将数组头地址重新保存在v0 准备下次排序
    addi a1,a1,-1   //a1-1
    move s1,a1      //a1->s1，控制下一次排序的次数，比之前减1
    bgtz s0,1b      //如果s0 > 0 则跳转到1
    nop

    nop             //循环结束
```

2.2 GPIO 的驱动和控制

2.2.1 设计方案

本实验主要实现“呼吸灯”和“流水灯”的效果。

对于 LED 灯，当 GPIO 输出“1”（逻辑高电平）所对应的灯亮，当 GPIO 输出“0”（逻辑低电平）时，所对应的灯暗。当按下按键时，按键接地，GPIO 端口输入呈现“0”（逻辑低电平）；当没有按下按键时。GPIO 端口输入“1”（逻辑高电平）。

根据这两条原理，设计实验方案如下：寻找 8 个 GPIO 端口（29，30，7，6，22，20，17，15）接 LED 灯，寻找 4 个 GPIO 端口（4，5，23，21）接按键，LED 灯的 VCC 接 3.3V，按键的 GND 接地即 GND。然后将 LED 灯和按键通过代码分别定义为输出和输入。接着用代码使能，这里注意，因 LED 灯接的是 3.3V 即高电平，要想让输出逻辑高电平，根据电势差原理，代码中应使能 1（逻辑高电平时）LED 灯暗，使能 0（逻辑低电平）LED 灯亮。

呼吸灯实现方案：先让第一个灯亮，维持第一个灯亮一小段时间，让第一个灯暗，第二个灯亮，再维持第二个亮一小段时间，以此类推，通过延缓相邻灯之间的亮暗速度，实现呼吸灯效果。

流水灯实现方案：先让第一个灯亮，然后迅速暗下来，同时马上让第二个灯亮，接着第二个灯马上暗，第三个灯同时亮起来，以此类推，通过加快相邻灯之间的亮暗速度，实现流水灯效果。

其中，用按键来控制呼吸灯和流水灯的切换。

注：原本目标是用汇编语言实现，后根据实际情况调整为用 C 语言实现。

2.2.2 设计实现

首先，定义按键连接的 4 个 GPIO 端口为输入，LED 灯连接的 8 个端口为输出

```
1. void init_input(){
```

```
    gpio_enable(4, DIR_IN);
    gpio_enable(5, DIR_IN);
    gpio_enable(23, DIR_IN);
    gpio_enable(21, DIR_IN);
}

void init_output(){
    gpio_enable(29, DIR_OUT);
    gpio_enable(30, DIR_OUT);
    gpio_enable(7, DIR_OUT);
    gpio_enable(6, DIR_OUT);
    gpio_enable(22, DIR_OUT);
    gpio_enable(20, DIR_OUT);
    gpio_enable(17, DIR_OUT);
    gpio_enable(15, DIR_OUT);
}
```

然后，通过使能亮，维持当前状态，使能暗这个过程不断重复实现呼吸灯效果。

```
void breath_led()
{
    gpio_write(29,0);
    delay_ms(1000);
    gpio_write(29,1);
    delay_ms(1000);
    gpio_write(29,0);
    delay_ms(1000);

    gpio_write(30,0);
    delay_ms(1000);
    gpio_write(30,1);
    delay_ms(1000);
    gpio_write(29,0);
    delay_ms(1000);

    gpio_write(7,0);
    delay_ms(1000);
    gpio_write(7,1);
    delay_ms(1000);
    gpio_write(7,0);
    delay_ms(1000);

    gpio_write(6,0);
    delay_ms(1000);
    gpio_write(6,1);
```

```
    delay_ms(1000);
    gpio_write(6,0);
    delay_ms(1000);
    control_led();
}
```

接着先定义一个让灯全暗的函数 flow()作为流水灯的初始状态和中间状态

```
void flow(){
    gpio_write(29,1);
    gpio_write(30,1);
    gpio_write(7,1);
    gpio_write(6,1);
    gpio_write(22,1);
    gpio_write(20,1);
    gpio_write(17,1);
    gpio_write(15,1);
}
```

然后通过全暗，当前灯亮起，当前灯迅速变暗同时下一个灯快速亮起的过程来实现流水灯

```
void flow_rea(){
    flow();
    gpio_write(29,0);
    delay_ms(200);
    flow();
    gpio_write(30,0);
    delay_ms(200);
    flow();
    gpio_write(7,0);
    delay_ms(200);
    flow();
    gpio_write(6,0);
    delay_ms(200);
    flow();
    gpio_write(22,0);
    delay_ms(200);
    flow();
    gpio_write(20,0);
    delay_ms(200);
    flow();
    gpio_write(17,0);
    delay_ms(200);
    flow();
    gpio_write(15,0);
    delay_ms(200);
```

```
    control_led();
}
```

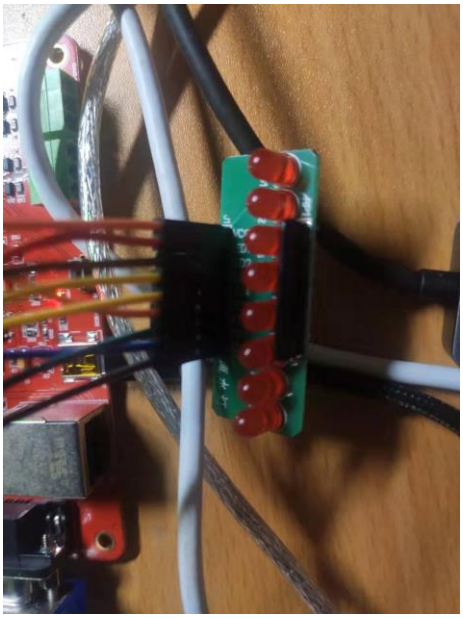
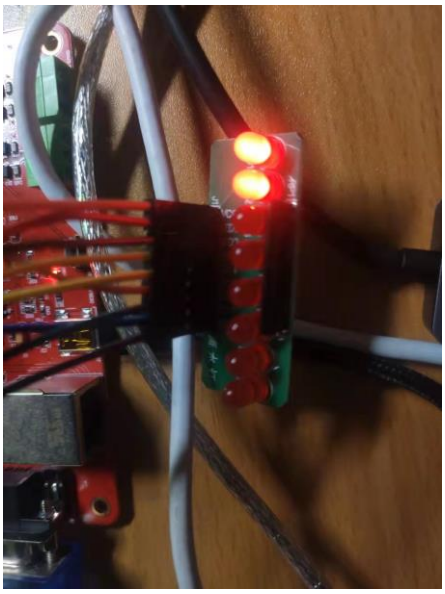
最后，通过 control_led 函数来通过按键切换呼吸灯和流水灯。按下 S1 为流水灯，S2 为呼吸灯。

```
void control_led()
{
    int i=0;
    for(;;)
    {
        if(gpio_read(4)==0)
        {
            i=1;
        }
        if(gpio_read(5)==0)
        {
            i=2;
        }
        if(gpio_read(23)==0)
        {
            i=3;
        }
        if(gpio_read(21)==0)
        {
            i=4;
        }
        if(i == 1)
        {
            flow_rea();
        }
        if(i == 2)
        {
            breath_led();
        }
    }
}
```

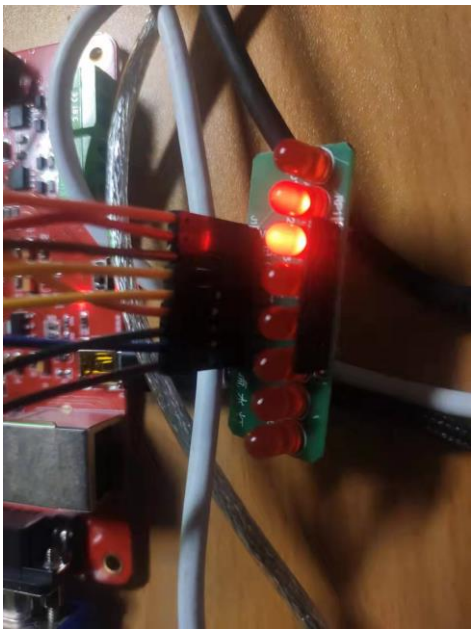
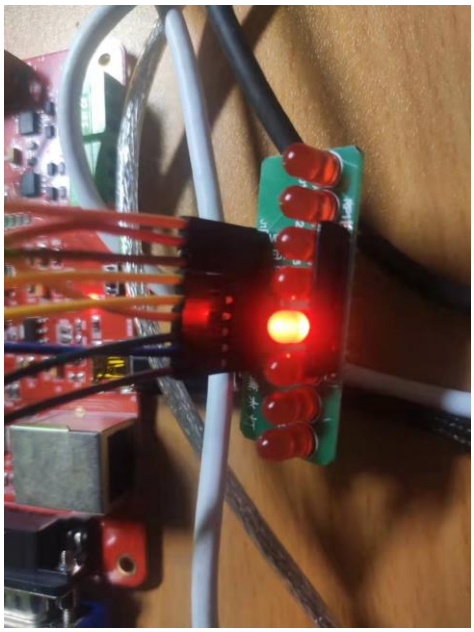
完整代码如下 呼吸灯作为默认状态

```
//-------------------------------------------------------
// 主程序
//-------------------------------------------------------

void init_input(){
    gpio_enable(4, DIR_IN);
    gpio_enable(5, DIR_IN);
    gpio_enable(23, DIR_IN);
    gpio_enable(21, DIR_IN);
}

void init_output(){
    gpio_enable(29, DIR_OUT);
    gpio_enable(30, DIR_OUT);
    gpio_enable(7, DIR_OUT);
    gpio_enable(6, DIR_OUT);
    gpio_enable(22, DIR_OUT);
    gpio_enable(20, DIR_OUT);
    gpio_enable(17, DIR_OUT);
    gpio_enable(15, DIR_OUT);
}


void breath_led()
{
    gpio_write(29,0);
    delay_ms(1000);
    gpio_write(29,1);
    delay_ms(1000);
    gpio_write(29,0);
    delay_ms(1000);

    gpio_write(30,0);
    delay_ms(1000);
    gpio_write(30,1);
    delay_ms(1000);
    gpio_write(29,0);
    delay_ms(1000);

    gpio_write(7,0);
    delay_ms(1000);
    gpio_write(7,1);
    delay_ms(1000);
    gpio_write(7,0);
    delay_ms(1000);

    gpio_write(6,0);
    delay_ms(1000);
    gpio_write(6,1);
    delay_ms(1000);
    gpio_write(6,0);
    delay_ms(1000);
    control_led();
}

void flow(){
    gpio_write(29,1);
    gpio_write(30,1);
    gpio_write(7,1);
    gpio_write(6,1);
    gpio_write(22,1);
    gpio_write(20,1);
    gpio_write(17,1);
    gpio_write(15,1);
}


void control_led()
{
    int i=0;
    for(;;)
    {
        if(gpio_read(4)==0)
        {
            i=1;
            flow_rea();
        }
        if(gpio_read(5)==0)
        {
            i=2;
            breath_led();
        }
        if(gpio_read(23)==0)
        {
            i=3;
        }
        if(gpio_read(21)==0)
        {
            i=4;
        }
    }
}


int main(void)
{
    printk("\r\nmain() function.\r\n");

    ls1x_drv_init();                    /* Initialize de

    install_3th_libraries();            /* Install 3th l

    init_input();
    init_output();

    breath_led();
    control_led();
    delay_ms(100);

    /*
     * Never goto here!
     */
    return 0;
}
```

呼吸灯效果图

流水灯效果图

2.3 串口重定位实现

2.3.1 设计方案

本实验实现 printf()的重定向。原来 printf()是通过串口 5 输出，现创建 iprintf()通过串口 3 输出，实现 printf()的重定向。

首先，创建 uart3.h 和 uart3.c 两个串口三的底层文件，实现初始化，读写等基本功能，再创建 putcharr.c 创建_putchar()函数完成输出单个字符的重定向。然后在 printf.c 和 printf.h 中仿照 printf()函数和_out_char 创建 iprintf()函数和_out_char1 并声明，通过_putchart() -> _out_char1() -> iprintf()的引用逻辑来完成串口 3 的重定位，最后使用 iprintf()输出一个值在串口调试助手里查看输出。

2.3.2 设计实现

首先在/libc 下新建 uart3.h 和 uart3.c 文件，定义 uart3 寄存器，实现 uart3 的数据接受缓冲区，初始化，读，写等基本功能。

Uart3 的定义和数据接受缓冲区

```
/*
 * uart3.c
 *
 * created: 2021/10/23
 *  author:
 */

/*
 * UART3 寄存器定义
 */

#include "uart3.h"

static HW_UART_t *pUART3 = NULL;        // UART3 设备指针

#define UART3_USE_INTERRUPT     0       // 是否使用中断

unsigned char buf_H[0];

#if UART3_USE_INTERRUPT

/*
 * 数据接收缓冲区
 */

//-----------------------------------------------------------------------------
// buffer: cycle mode, drop the most oldest data when add
//-----------------------------------------------------------------------------

/*
 * 保存 UART_BUF_SIZE 个字符.
 *
 * 字节: 0  1  2  3  4  5        ...
 *       __ __ xx xx xx xx __ __ ...
 *             ^           ^
 *             pHead       |
 *                         pTail
 *
 * if full or empty: pHead==pTail;
 */

#define UART_BUF_SIZE   256         /* 缓冲区大小 */

typedef struct
{
    char  Buf[UART_BUF_SIZE];
    int   Count;
    char *pHead;
    char *pTail;
} UART_buf_t;

static UART_buf_t s_RxBuf;          /* 接收缓冲区 */
static UART_buf_t s_TxBuf;          /* 发送缓冲区 */
```

中断句柄

```
/*
 * UART3 初始化
 *
 * 参数:    baudrate: 通信波特率
 *          databits: 数据位数
 *          eccmode:  校验模式, 'O': 奇校验; 'E': 偶校验; 'N': 无
 *          stopbits: 结束位数
 *
 * 返回:    0
 */
int uart3_initialize(int baudrate, int databits, char eccmode, int stopbits)
{
    unsigned int divisor, bus_freq;
    unsigned char lcr;

    if (baudrate < 2400)
        baudrate = 115200;

    pUART3 = (HW_UART_t *)LS1B_UART3_BASE;

    pUART3->lcr = 0;
    pUART3->R1.ier = 0;

    bus_freq = LS1x_BUS_FREQUENCY(CPU_XTAL_FREQUENCY);  // 总线频率
    divisor =  bus_freq / 16 / baudrate;                /* 计算分频系数, 总线频率/baudrate/16 */
    pUART3->lcr = 0x80;                                 // 设置 DLAB
    pUART3->R0.dll = divisor & 0xFF;                    // 分频值低字节
    pUART3->R1.dlh = (divisor >> 8) & 0xFF;             // 分频值高字节
    pUART3->R2.fcr = 0x07;                              /* reset fifo */

    switch (databits)
    {
        case 5:  lcr = 0x00; break;
        case 6:  lcr = 0x01; break;
        case 7:  lcr = 0x02; break;
        case 8:
        default: lcr = 0x03; break;
    }

    switch (eccmode)
    {
            . . .   .       .   .

    switch (eccmode)
    {
        case 'O': lcr |= 0x08; break;
        case 'E': lcr |= 0x18; break;
    }

    if (stopbits == 2)
        lcr |= 0x04;

    pUART3->lcr = lcr;

#if UART3_USE_INTERRUPT

    /* 初始化数据缓冲区 */
    s_RxBuf.Count = 0;
    s_RxBuf.pHead = s_RxBuf.pTail = s_RxBuf.Buf;
    s_TxBuf.Count = 0;
    s_TxBuf.pHead = s_TxBuf.pTail = s_TxBuf.Buf;

    /* 安装中断 */
    ls1x_install_irq_handler(LS1B_UART3_IRQ, uart3_isr, NULL);

    /* 开中断 */
    LS1x_INTC_EDGE(LS1x_INTC0_BASE) &= ~INTC0_UART3_BIT;
    LS1x_INTC_POL( LS1x_INTC0_BASE) |=  INTC0_UART3_BIT;
    LS1x_INTC_CLR( LS1x_INTC0_BASE)  =  INTC0_UART3_BIT;
    LS1x_INTC_IEN( LS1x_INTC0_BASE) |=  INTC0_UART3_BIT;

    pUART3->R1.ier = 0x01;          /* interrupt on rx */

#endif

    return 0;
}
```

读数据

```
/*
 * UART3 读数据
 *
 * 参数:    buf:    数据缓冲区
 *          size:   读字节数
 *
 * 返回:    本次读的字节数
 */
int uart3_read(unsigned char *buf, int size)
{
    int count = 0;

    if ((pUART3 == NULL) || (buf == NULL))
        return -1;

    if (size < 0)
        return 0;

#if UART3_USE_INTERRUPT

    mips_interrupt_disable();
    count = dequeue_from_buffer(&s_RxBuf, buf, size);
    mips_interrupt_enable();

    return count;
#else

 //   unsigned char *p = buf;
    while (count<size)
    {
        if (pUART3->lsr & 0x01)
        {
//            *p++ = pUART3->R0.dat;
            *buf = pUART3->R0.dat;
            buf_H[0] = *buf;
            uart3_write(buf_H,1); //字符逐字回显
            if(*buf==10)          //如果读到了换行，代表结束读取（发送回车实际上是一个回车一个换行）
                break;
            buf++;
            count++;
        }
        else
            delay_us(100);
    }
    return size;
#endif
```

写数据

```
/*
 * UART3 写数据
 */
int uart3_write(unsigned char *buf, int size)
{
    if ((pUART3 == NULL) || (buf == NULL))
        return -1;

    if (size < 0)
        return 0;
```

```
#if UART3_USE_INTERRUPT

    int i, sent = 0;

    /* if idle, send immediately
    int i, sent = 0;

    /* if idle, send immediately
     */
    if (pUART3->lsr & 0x20)             /* transmitter ready */
    {
        sent = size <= UART_FIFO_SIZE ? size : UART_FIFO_SIZE;

        for (i=0; i<sent; ++i)          /* write data to transmit buffer */
            pUART3->R0.dat = buf[i];

        if (sent > 0)
            pUART3->R1.ier = 0x03;      /* interrupt on rx & tx */
        else
            pUART3->R1.ier = 0x01;      /* interrupt on rx */
    }

    /* add remain data to transmit cached buffer
     */
    if (sent < size)
    {
        mips_interrupt_disable();
        sent += enqueue_to_buffer(&s_TxBuf, buf + sent, size - sent);
        mips_interrupt_enable();
    }

    return sent;
```

```
#else

    int count = 0;
    unsigned char *p = buf;

    while (count < size)
    {
        if (pUART3->lsr & 0x20)         /* 传输准备就绪 */
        {
            pUART3->R0.dat = *p++;      /* 传输单个字符 */
            count++;
        }
        else
            delay_us(100);
    }

    return size;

#endif
}
```

创建 Putcharr.c 并编写_putchar1()函数

```
8
9  #include "uart3.h"
0
1
2  int _putchar1(char ch)
3  {
4      uart3_write(( unsigned char *)&ch, 1);    //无符号强制类型转换，单字符
5      return 0;
6  }
7
```

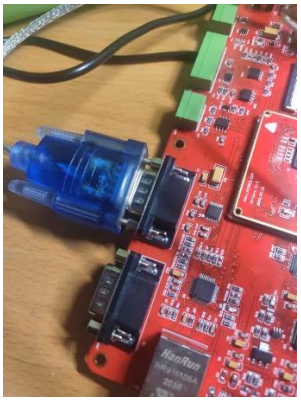
在 Printf.c 中创建 iprintf()和_out_char1()函数

```
static inline void _out_char1(char character, void* buffer, size_t idx, size_t maxlen)
{
    (void)buffer; (void)idx; (void)maxlen;
    if (character)
    {
        _putchar1(character);
    }
}

int iprintf(const char* format, ...)
{
    uart3_initialize(115200, 8, 'N', 1);
    va_list va;
    va_start(va, format);
    char buffer[1];
    const int ret = _vsnprintf(_out_char1, buffer, (size_t)-1, format, va);
    va_end(va);
    return ret;
}
```

在 Printf.h 中声明创建的函数

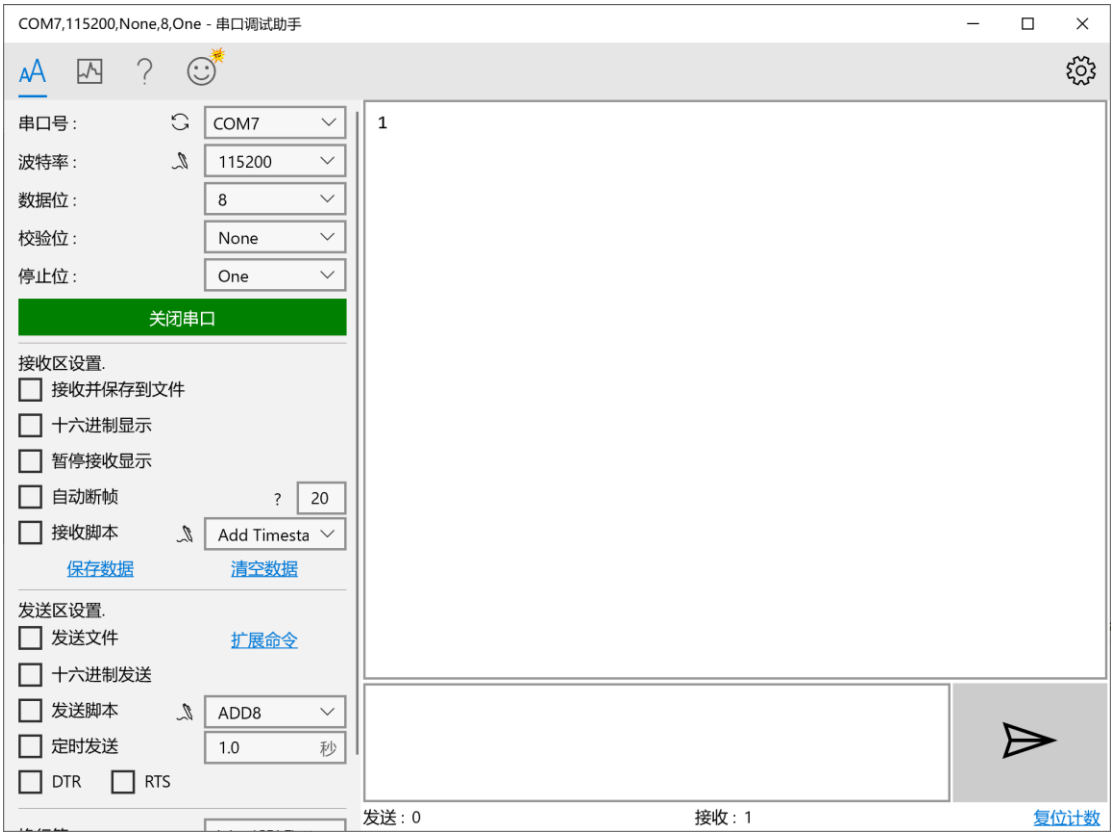
```
extern void _putchar1(char ch);

extern void uart3_initialize(int baudrate, int databits, char eccmode, int stopbits);

void _out_char1(char character, void* buffer, size_t idx, size_t maxlen)

int iprintf(const char* format, ...);
```

板子连接串口 3



测试 串口连接助手结果

```c
int main(void)
{
    int a=1;

    //printk("\r\nmain() function.\r\n");

    ls1x_drv_init();                    /* Initialize device drivers */

    install_3th_libraries();            /* Install 3th libraies */

    uart3_initialize(115200, 8, 'N', 1);

    iprintf("%d",a);
    //printf("%d",a);



    /*
     * Never goto here!
     */
    return 0;
}
```



2.4 Linux 操作系统的移植

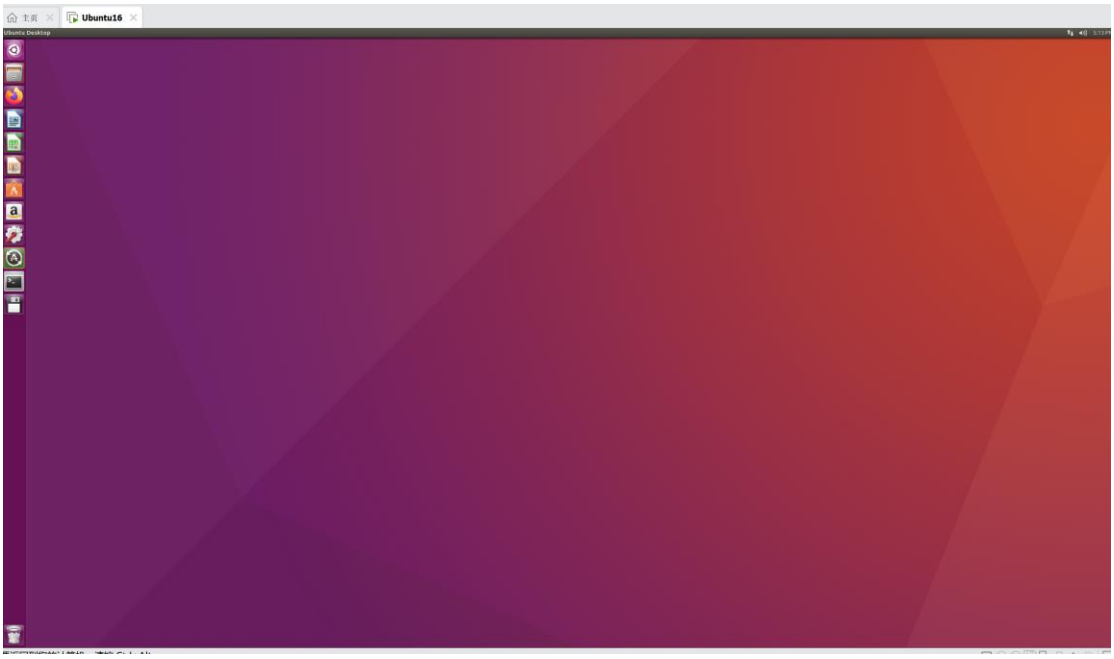
2.4.1 设计方案

本实验实现了 linux 操作系统的移植，在移植后的系统里打印了 BUCT。

首先下载 vmware 装好 ubuntu16 虚拟机，在 ubuntu 下安装 mipsel-linux-gcc 交叉编译器，然后编译 Linux 源码和文件系统。接着通过 Putty 连接板子至 PMON 命令行，与主机相连，通过 tftpd 将主机里的内核和文件系统下载到龙芯 1B 开发板内，启动移植后的编译系统，运行在 ubuntu 下交叉编译输出的 Print 文件，输出 BUCT。

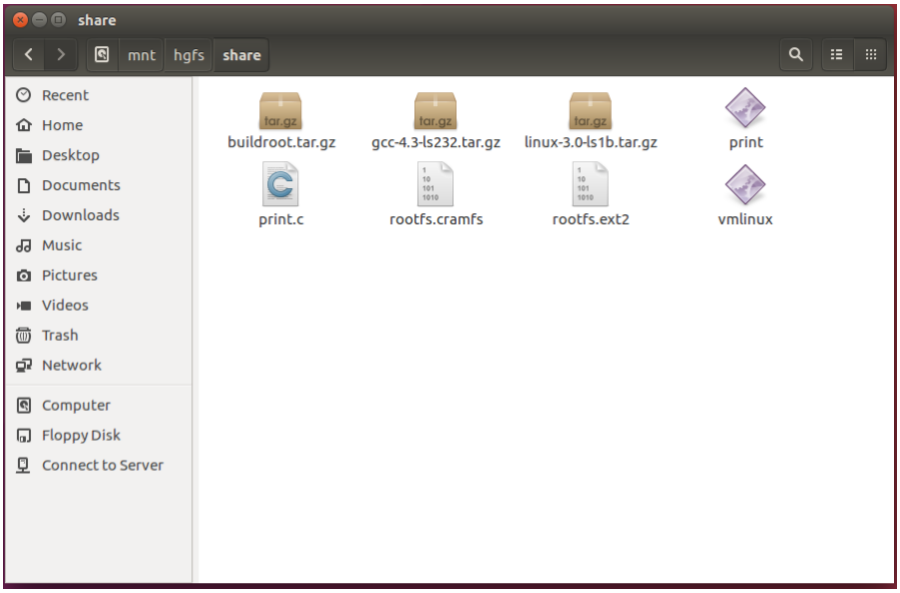
2.4.2 设计实现

首先下载 vmware,装好 ubuntu16



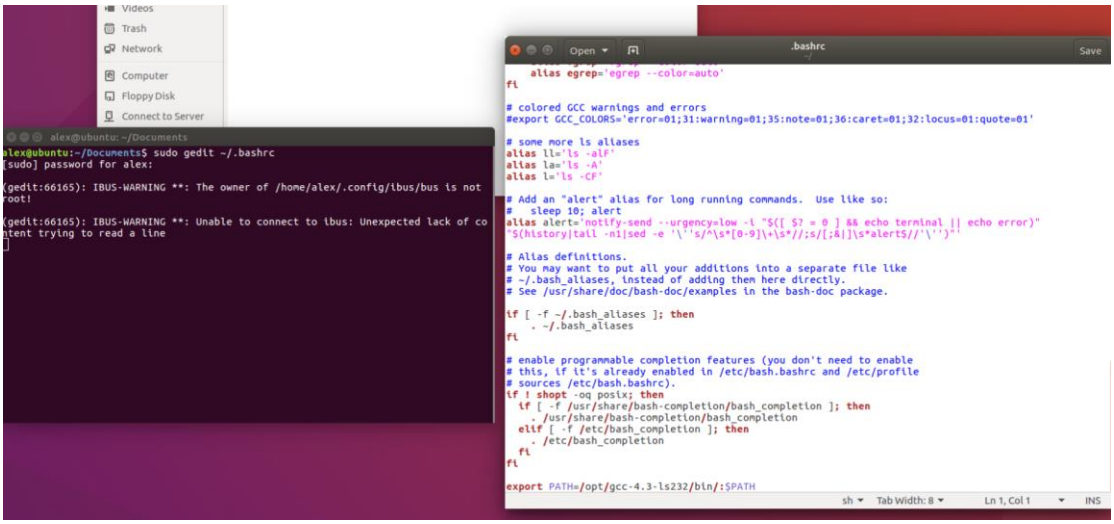
然后开启共享文件夹，将 buildroot.tar.gz，gcc-4.3-ls232.tar.gz，linux-

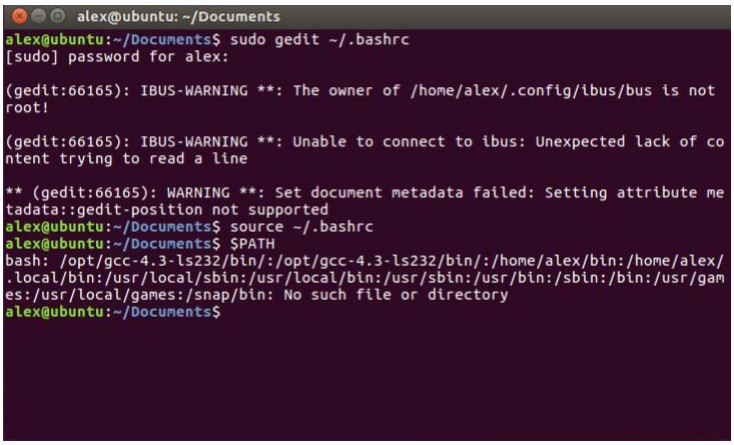
3.0-ls1b.tar.gz 三个文件传输到 ubuntu，在 mnt/hgfs/share 下查看



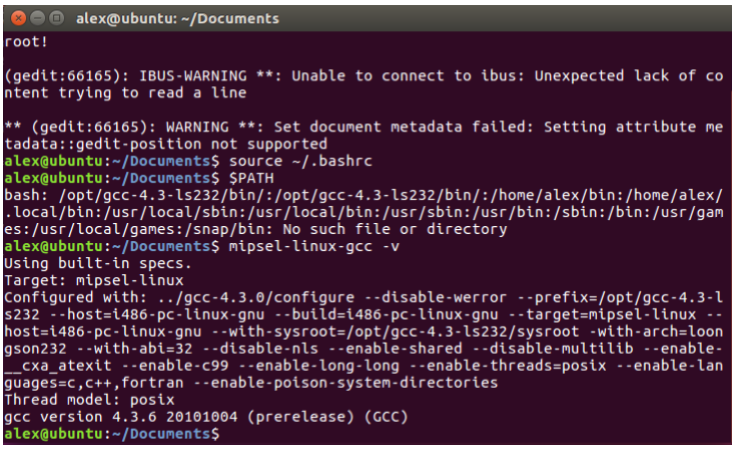
将其复制到 Documents 下

将 gcc-4.3-ls232.tar.gz 解压并安装，将 gcc 路径添加到环境变量





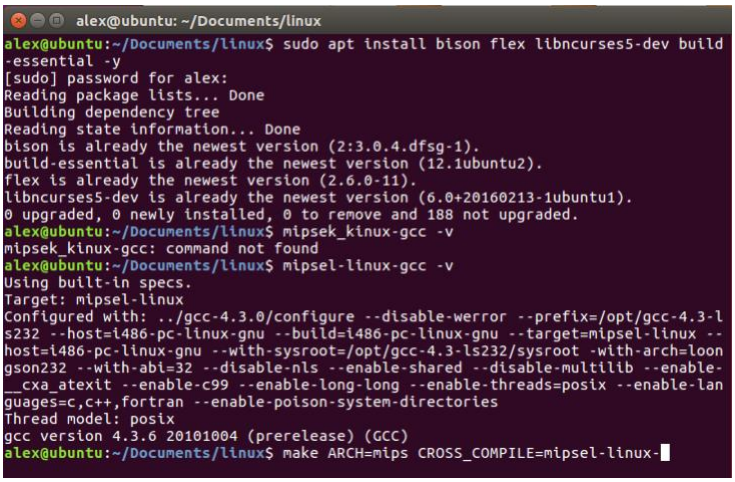
Mipsel-linux-gcc -v 查看交叉编译器版本



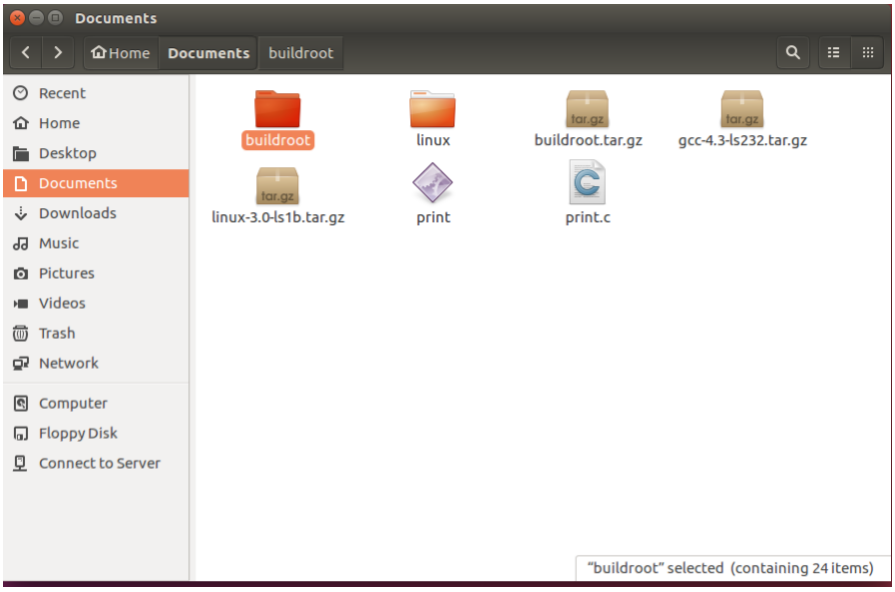
接下来进行 Linux 源码编译

解压 Linux.zip 并进入解压后的文件夹

安装必要的包，检查交叉编译器已成功安装，开始编译

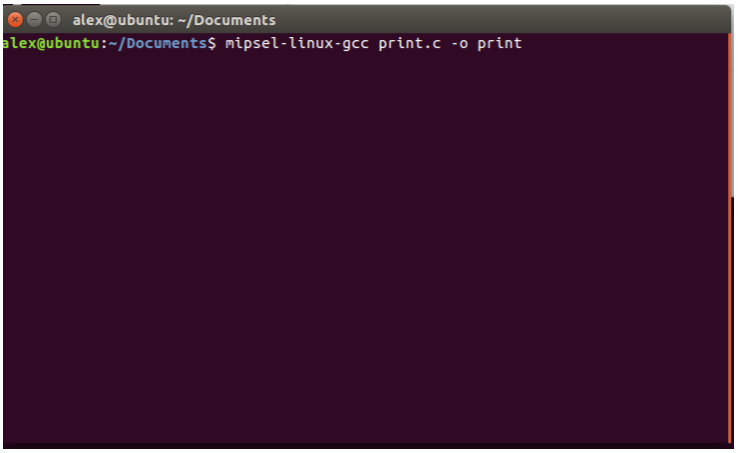


然后编译文件系统，过程同编译 linux 源码一样

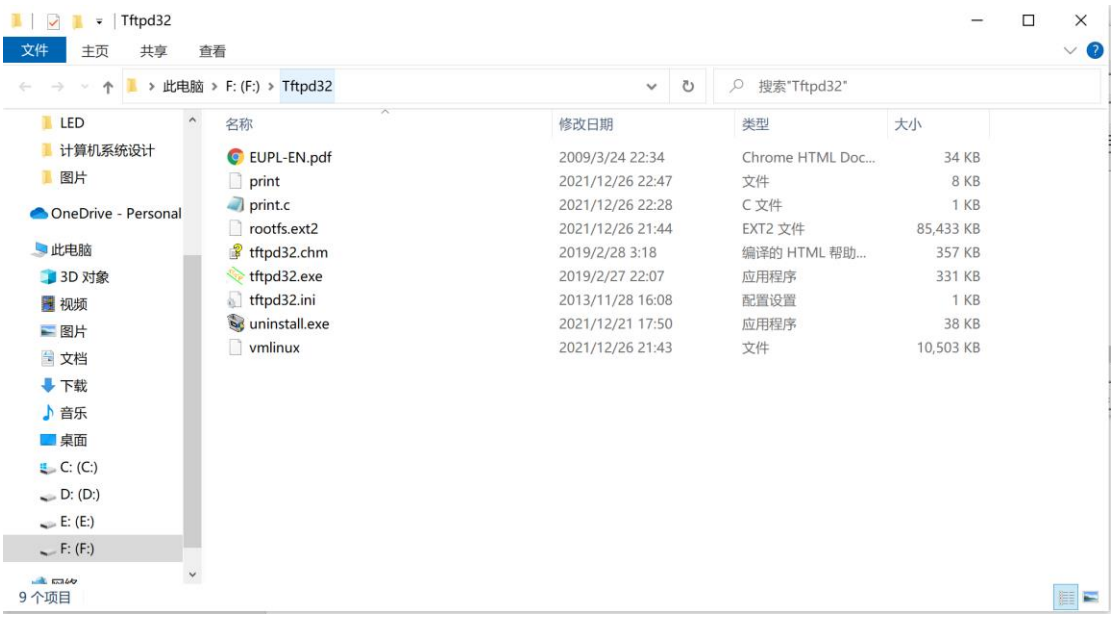


新建 print.c 文件，实现打印 BUCT 的功能

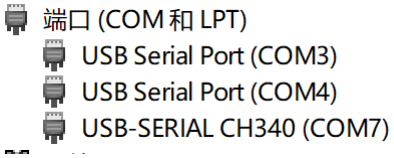
用交叉编译器编译 print.c 文件，得到 print



经过以上编译后，将 buildroot/output/images/rootfs.ext2，linux/vmlinux 和 print 放入共享文件夹，然后打开 tftp 软件，选择 showdir---explore，将这三个文件复制到这。

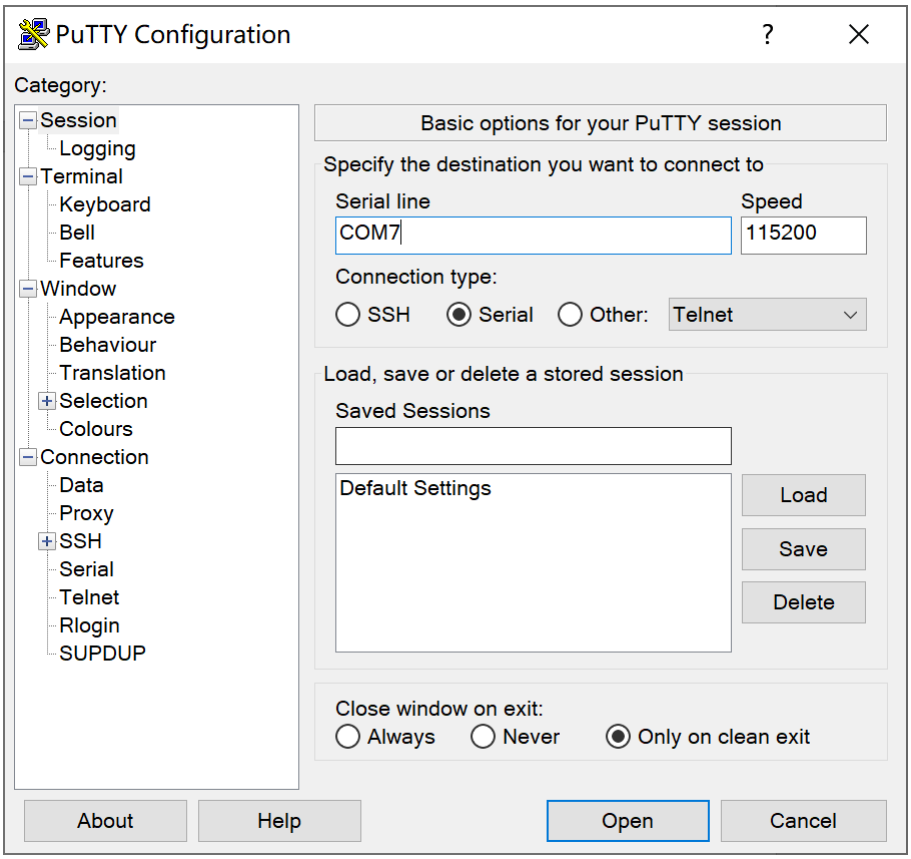


接着打开设备管理器，查看虚拟端口号

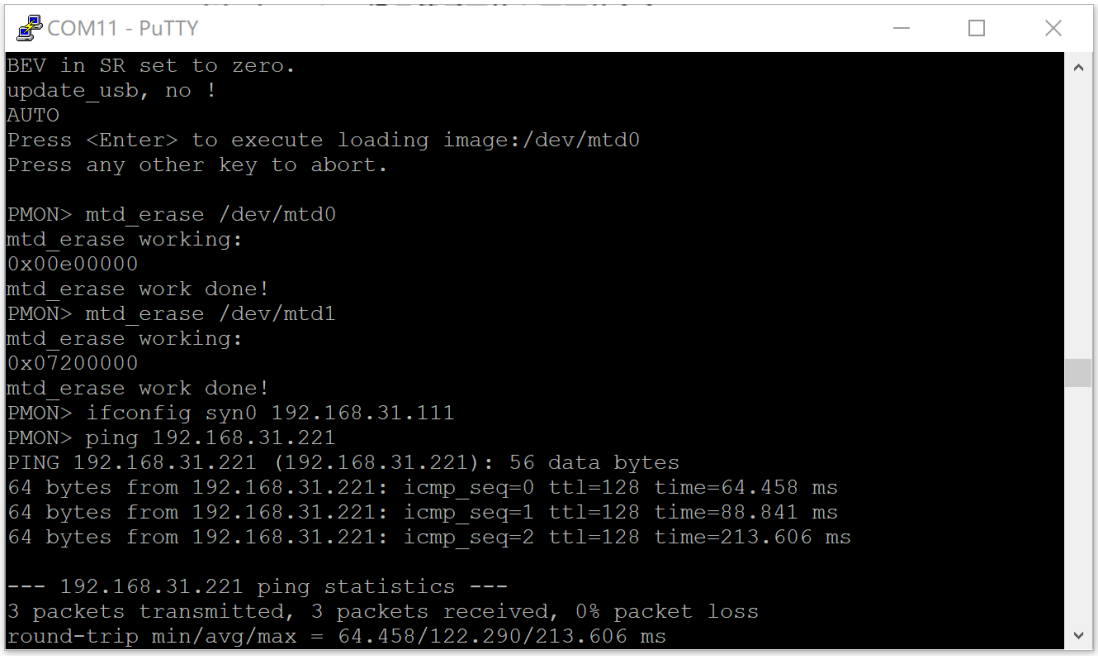


这里为 COM7

接着打开 Putty，配置 type 为 Serial，COM7 端口，速率为 115200



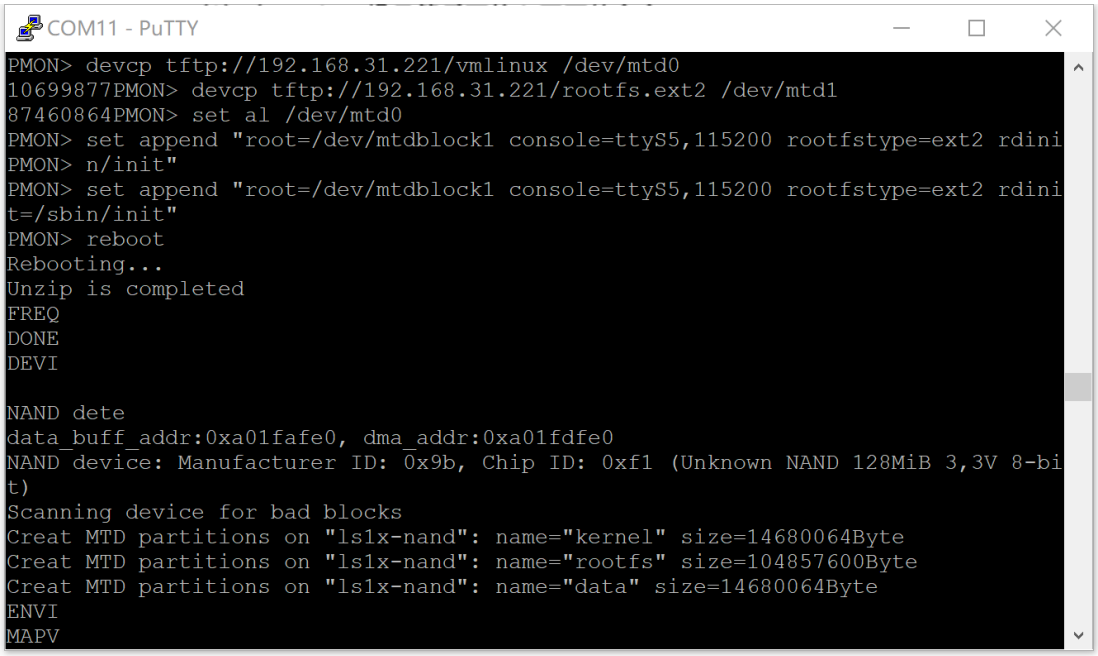
然后进入 PMON 命令行，输入 mtd_erase /dev/mtd0 和 mtd_erase /dev/mtd1 擦除数据，再用 ifconfig syn0 192.168.31.111 分配同网段的 IP 地址（主机 IP 为 192.168.31.221），再 ping 主机 IP 192.168..31.221 查看是否连通



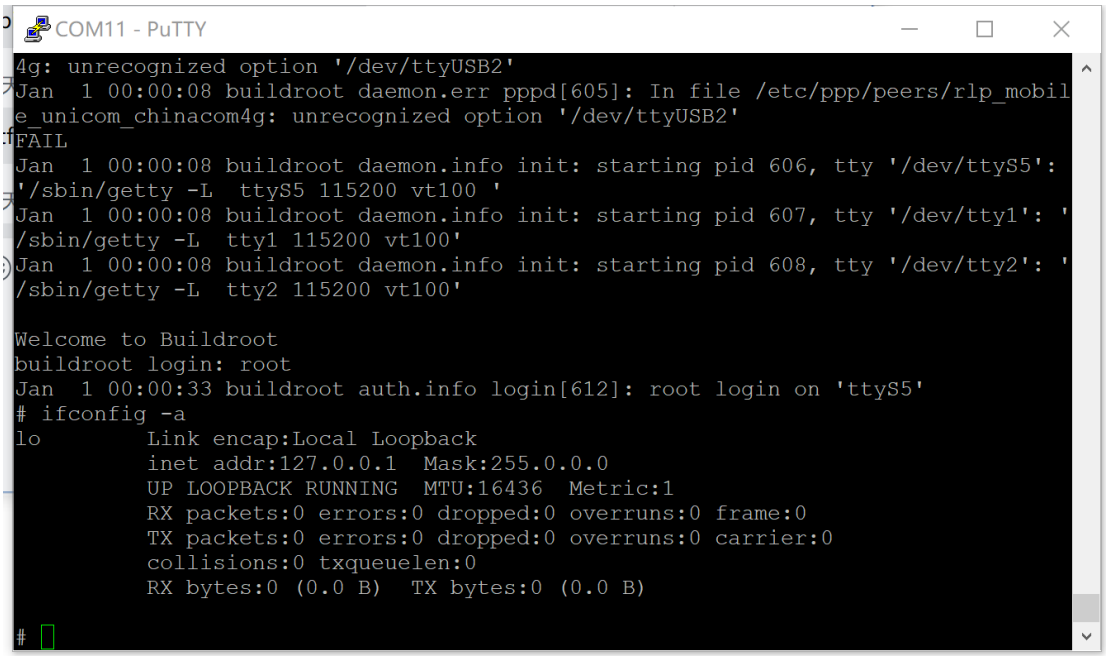
然后传输文件 devcp tftp://192.168.31.221/vmlinux /dev/mtd0 传输 vmlinux

Devcp tftp://192.168.31.221/rootfs.ext2 /dev/mtd1 传输 rootfs.ext2

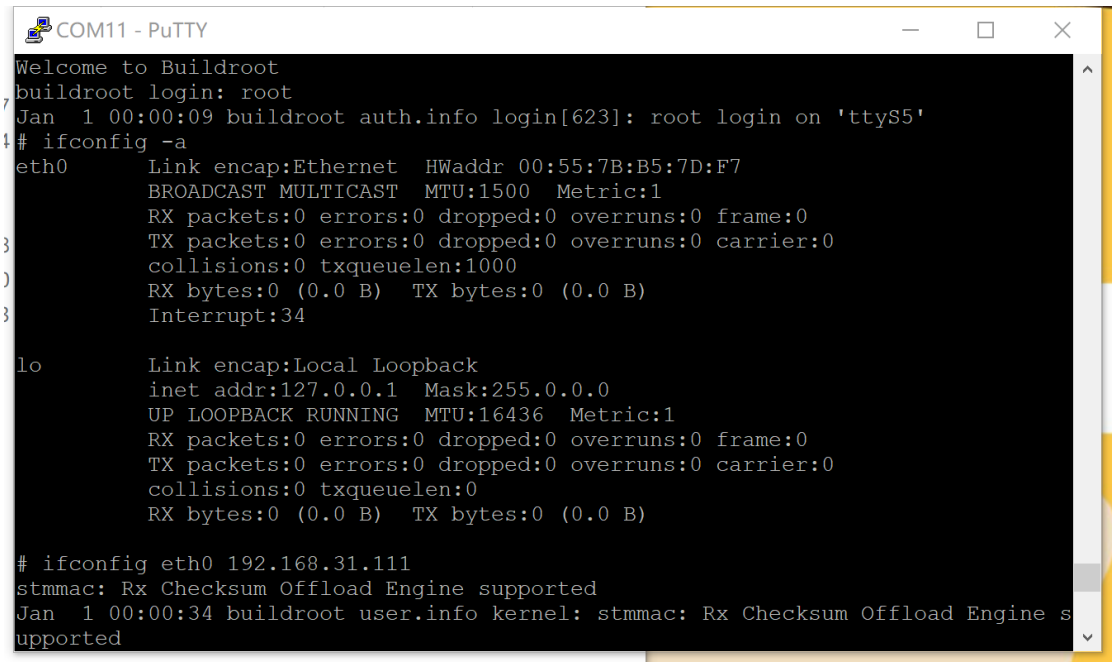
启动文件系统 set append “root=/dev/mtdblock1 console=ttyS5,115200 rootfstype=ext2 rdinit=/sbin/init 并 reboot 重启



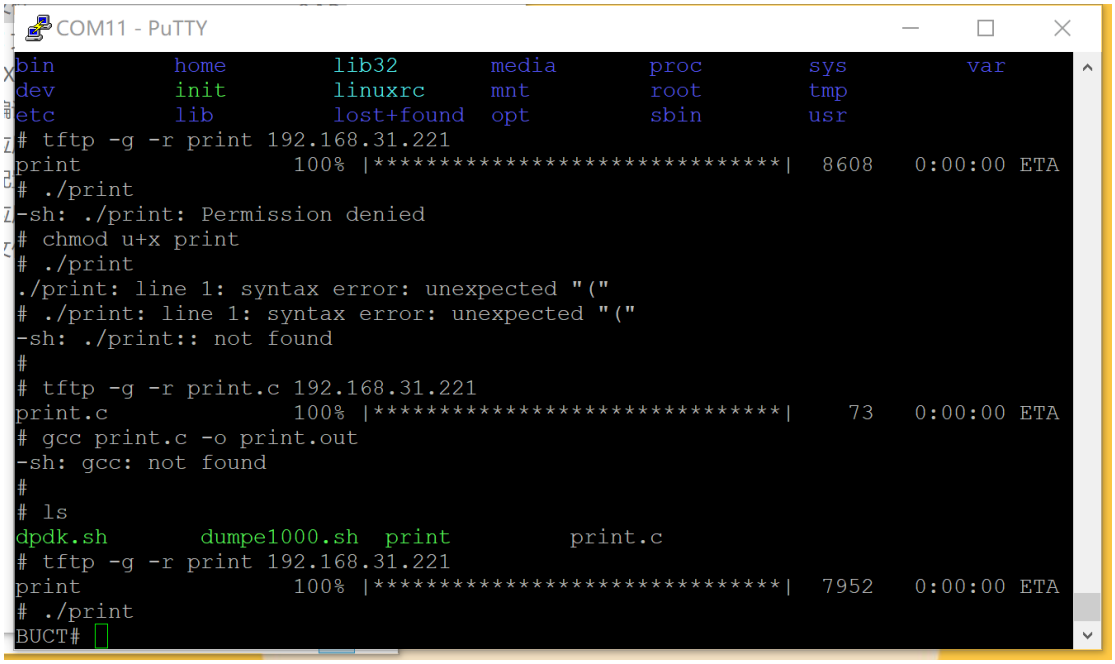
进入系统后用 root 登陆，输入 ifconfig -a 查看网卡



分配 IP ifconfig eth0 192.168.31.111



从主机下载 print 文件，运行 ./print 查看打印结果 BUCT



3 设计调试

3.1 算法的汇编语言实现

1 最后一个 nop 要加上。

```
.data

array: .byte 23,84,13,92,32,64,57,99,74,19  //数组赋值


.text
    la v0,array     //将数组头地址放进v0
    addi s0,zero,10  //外循环次数为10 储存到s0
    addi s1,zero,9   //内循环次数为9 储存到s1
    move a1,s1      //将s1->a1

1:
    lbu t1,0(v0)    //提取array[0] 加载到t1
    lbu t2,1(v0)    //提取array[1] 加载到t2
    sltu a0,t1,t2   //如何t1 < t3 则为1 反之
    bgtz a0,2f      //如果a0大于1 则跳转2 继续  反之进行交换
    nop
    sb t1,1(v0)    //将array[1]移到t1中
    sb t2,0(v0)    //将array[0]移到t2中

2:
    addi v0,v0,1    //移到下一个  即array[i+1]
    addi s1,s1,-1   //同时内循环次数-1
    bgtz s1,1b      //如果s1 > 0则跳转到1
    nop
    addi s0,s0,-1   //外循环次数减1
    la v0,array     //将数组头地址重新保存在v0 准备下次排序
    addi a1,a1,-1   //a1-1
    move s1,a1      //a1->s1，控制下一次排序的次数，比之前减1
    bgtz s0,1b      //如果s0 > 0 则跳转到1
    nop

    nop             //循环结束


    .set reorder
ENDFRAME(bsp_start)
```

2 进入调试模式但无反应

逐个检查原因，把防火墙关闭后即可运行

3.2 GPIO 的驱动和控制

1 代码部分，LED 灯高低电平与亮暗关系，经过请教同学后，明白了因为 LED 灯接了 3.3V 高电平，因为电势差原理，所以这里输出为低电平才会形成电势差，让 LED 灯得到高电平输入，才输出高电平亮的情况

```
void breath_led()
{
    gpio_write(29,0);
    delay_ms(1000);
    gpio_write(29,1);
    delay_ms(1000);
    gpio_write(29,0);
    delay_ms(1000);

    gpio_write(30,0);
    delay_ms(1000);
    gpio_write(30,1);
    delay_ms(1000);
    gpio_write(29,0);
    delay_ms(1000);

    gpio_write(7,0);
    delay_ms(1000);
    gpio_write(7,1);
    delay_ms(1000);
    gpio_write(7,0);
    delay_ms(1000);

    gpio_write(6,0);
    delay_ms(1000);
    gpio_write(6,1);
    delay_ms(1000);
    gpio_write(6,0);
    delay_ms(1000);
    control_led();
}
```

2 按键控制只能一次性

原先代码将 control_led()放在 main()函数里，只能变换一次，呼吸灯变成流水灯效果后，无法再变成呼吸灯。

经过思考后，后来在呼吸灯和流水灯函数里最后调用一次 control_led()。

```
    gpio_write(6,0);
    delay_ms(1000);
    gpio_write(6,1);
    delay_ms(1000);
    gpio_write(6,0);
    delay_ms(1000);
    control_led();
}
```

```
    gpio_write(17,0);
    delay_ms(200);
    flow();
    gpio_write(15,0);
    delay_ms(200);
    control_led();
}
```

这样在效果完毕后可以转换成另外一个效果。

3.3 串口重定位实现

1 Putcharr.c 不知道怎么写

请教学长后明白了这里应该怎么写，要强制转换成字符类型，size 为 1，

表示输出单个字符

```
#include "uart3.h"


int _putchar1(char ch)
{
    uart3_write(( unsigned char *)&ch, 1);   //无符号强制类型转换，单字符
    return 0;
}
```

2 输出乱码

主函数这里之前没有初始化，造成输出结果前有一两个乱码问号

```
int main(void)
{
    int a=1;

    //printk("\r\nmain() function.\r\n");

    ls1x_drv_init();                    /* Initialize device drivers */

    install_3th_libraries();            /* Install 3th libraies */

    uart3_initialize(115200, 8, 'N', 1);

    iprintf("%d",a);
    //printf("%d",a);
```

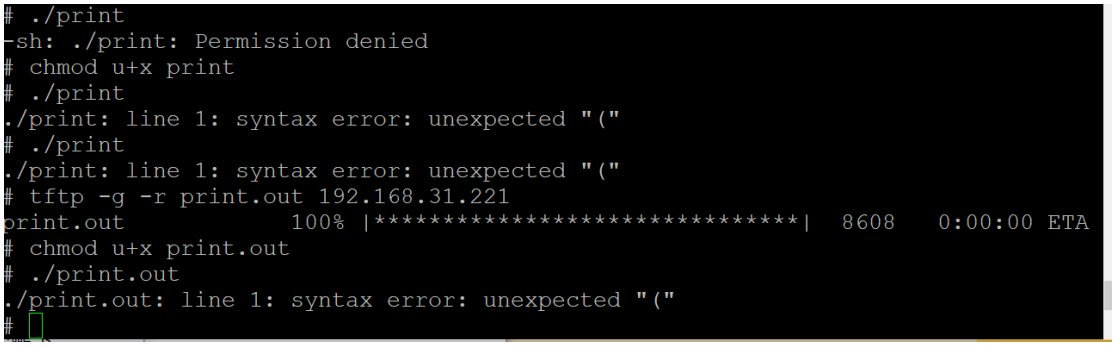
3.4. Linux 操作系统的移植

1 连接不上板子

分配给单片机静态 IP 后，第二天连接不上 PMON 了，报错 putty bind(udptty): Can't assign requested address

只能等其他同学做完后借同学板子

2 运行 ./print 报错



后来回顾过程，发现原因是在 Ubuntu 下是用 gcc 编译的而不是用 mipsel 编译的，所以移植后的系统无法运行，用 mipsel 编译后就成功运行

4 设计结论