嵌入式系统仿真实验第七讲

前面 6 讲，我们仿真了 ARM 的指令集以及如何启动计算机进入到高级语言编程环境，直到第 5 讲、第 6 讲我们才勉强给大家用到个简易的串口发送字符串的示范，表示 CPU 可以和外部进行交互了。现在大家理论章节学习到第四章外部接口，我也看了下网课的视频讲义，估计这样学非常之枯燥无谓，听完讲大家是不是会什么也没法记住。所以，实验仿真课从这节课开始，我要给大家正式讲外围接口的编程驱动以及如何仿真调试了。

1 GPIO 接口

实际上，作为一个嵌入式系统工程师，软件调试首先打交道便是 GPIO 口，因为上电后能够最方便知道的除了电源指示灯是否亮之外，我们在所有外部接口能工作之前，可以最方便使用的就是用某几个 GPIO 口去点 led 灯，让它闪烁，比如跑马灯或者你自己定义的**闪灯顺序来判断程序是否正常，程序到底跑到了哪一块**。

GPIO（General Purpose I/O Ports）就是通用输入/输出端口，是最为方便、可以简单使用的引脚，通过它们输出高低电平或者通过它们读入引脚的状态（高电平或低电平）。

大多数 ARM 系统都提供 GPIO 引脚作为系统的 I / O 接口。将某些 GPIO 引脚配置为输入，将其他引脚配置为输出，在启动阶段对 GPIO 引脚进行编程，以使其与某些实际设备（例如开关，传感器，LED 和继电器等）连接显示内部状态或者读取外部设置。早期嵌入式系统，GPIO 接口一般由四个 32 位寄存器组成。

GPIODIR: 设置引脚方向; 0 for input, 1 for output

GPIOSET: 设置引脚输出高电平 (3.3 V)

GPIOCLR: 设置引脚输出低电平 (0 V)

GPIOPIN: 读取该寄存器将返回所有引脚的状态

查 versatile 板子的资料，可以看到重要的外部接口在内存中的映射表，见表 1 所示。

板子的详细资料可以查看这个文档：

https://static.docs.arm.com/dui0225/d/DUI0225D_versatile_application_baseboard_arm926ej_s_ug.pdf

表1 Memory map of ARM versatile/ARM926EJ-S

MPMC Chip Select 0, 128 MB SRAM	0x00000000	128 MB
MPMC Chip Select 1, 128 MB expansion SRAM	0x08000000	128 MB
System registers	0x10000000	4 KB
Secondary Interrupt Controller (SIC)	0x10003000	4 KB
Multimedia Card Interface 0 (MMCID)	0x10005000	4 KB
Keyboard/Mouse Interface 0 (keyboard)	0x10006000	4 KB
Reserved (UART3 Interface)	0x10009000	4 KB
Ethernet Interface	0x10010000	64 KB
USB Interface	0x10020000	64 KB
Color LCD Controller	0x10120000	64 KB
DMA Controller	0x10130000	64 KB
Vectored Interrupt Controller (PIC)	0x10140000	64 KB
System Controller	0x101E0000	4 KB
Watchdog Interface	0x101E1000	4 KB
Timer modules 0 and 1 interface	0x101E2000	4 KB
(Timer 1 at 0x101E2020)	0x101E2FFF	
Timer modules 2 and 3 interface	0X101E3000	4 KB
(Timer 3 at 0x101E3020)	0X101E3FFF	
GPIO Interface (port 0)	0X101E4000	4 KB
GPIO Interface (port 1)	0x101E5000	4 KB
GPIO Interface (port 2)	0X101E6000	4 KB
UART 0 Interface	0x101E1000	4 KB
UART 1 Interface	0X101E2000	4 KB
UART 2 Interface	0x101F3000	4 KB
SSMC static expansion memory	0x20000000	256 MB

在 ARM Versatile-PB 板上，GPIO 接口有三个，port0 至 port2，基地址为 0x101E4000-0x101E6000。 每个端口提供 8 个 GPIO 引脚，这些引脚由一个（8 位）GPIODIR 寄存器和一个（8 位）GPIODATA 寄存器控制。 GPIO 输入也可以使用中断，而不仅仅只能靠查询输入引脚的状态。 由于 qemu 仿真的 ARM 虚拟机没有 GPIO 引脚，所以我们只描述 GPIO 编程的一般原理。 如果是实物实验，在实验室我们一般第一个硬件实验就会使 GPIO 口的实验，比如跑马灯。

例如：假设我们在 GPIO0 的[20:23]这四根管脚接了 4 个 LED 灯，下面的例子就是点亮这四个灯的程序。

```
.set BASE, 0x101E4000 // #define BASE 0x101E4000
.set IODIR, 0x000 // #define IODIR 0x000
.set IOSET, 0x004 // #define IOSET 0x004
.set IOCLR, 0x008 // #define IOCLR 0x008
.set IOPIN, 0x00C // #define IOPIN 0x00C
```

```
ldr r0, =GPIO_BASE    @R0 存放 GPIO0 的 IODIR 寄存器地址
ldr r1, =0x00FFFF00 @装载 32 位立即数，即设置值
str r1, [r0]        @IODIR=0x00FFFF00,IODIR 地址为 0x101E4000
mov r1, #0x00F00000
str r1, [r0,#IOSET] @IOSET=0x00F00000,IOSET 地址为 0x101E4004
```

2 UART 接口驱动

除了 GPIO 我们在启动的时候使用它来指示我们的硬件工作状态外，接下来就是串口通信开始接管我们的软硬件交互了。

实际上前面的例子中我已经用过 UART 了，但因为大家还没学到接口这一章，所以我没有去介绍 UART 的实验工作原理。

在下面的示例程序中，我们将编写一个简单的 UART 驱动程序，用于仿真串行终端上的 I / O。ARM versatile 板支持四个 PL011 UART 设备用于串行 I / O（ARM PL011 2016）。 每个 UART 器件在系统存储器映射中都有一个基地址。 4 个 UART 的基址为：

UART0：0x101F1000

UART1：0x101F2000

UART2：0x101F3000

UART3：0x10090000

每个 UART 都有许多寄存器，这些寄存器距基址有字节偏移。 以下列出了最重要的 UART 寄存器相对基地址的偏移地址以及功能介绍：

0x00 UARTDR 数据寄存器：用于读取/写入字符；

0x18 UARTFR 标志寄存器：TxEmpty，RxFull 等；

0x24 UARIBRD 波特率寄存器：设置波特率；

0x2C UARTLCR 线路控制寄存器：字符位数，奇偶校验等；

0x38 UARTIMIS 中断屏蔽寄存器，用于 TX 和 RX 中断。

我们可以按照以下步骤初始化 UART：

（1）将一个除数因子写入所需波特率的波特率寄存器。 在 ARM PL011 技术参考手册列表中，常用波特率的整数除数因子有（相对于 7.38 MHz UART 时钟）：

0x4 = 1152000，0xC = 38400，0x18 = 192000，0x20 = 14400，0x30 = 9600；

（2）写入线路控制寄存器，指定多少个字符位以及奇偶校验，例如每个字符 8 位，没有奇偶校验；

（3）写入中断屏蔽寄存器以启用/禁用 RX 和 TX 中断。

初始化完成后，我们可以对 UART 的数据寄存器进行读/写操作，对状态寄存器进行读操作：

数据寄存器（offset: 0x00）：数据输入（读）/数据输出（写）。

标志寄存器（offset: 0x18）：表示 UART 端口的状态：

```
  7   6   5   4   3   2   1   0
| TXFE RXFF TXFF RXFE BUSY  -   -   -  |
```

状态位表示的含义如下：

TXFE：Tx buffer empty；

RXFF：Rx buffer full；

TXFF：Tx buffer full；

RXFE：Rx buffer empty；

BUSY：device busy。

访问状态寄存器各个位的标准方法是将它们定义为符号常量，例如：

```
#define TXFE 0x80
#define RXFF 0x40
#define TXFF 0x20
#define RXFE 0x10
#define BUSY 0x08
```

然后将它们用作位掩码来测试标志寄存器的状态位。 接下来我们开始 UART 驱动程序编码（uart.h, uart.c）：

uart.h:

```
#ifndef _UART_H_
#define _UART_H_


#define TXFE 0x80
#define TXFF 0x20
#define RXFE 0x10
#define RXFF 0x40
#define BUSY 0x08

#define UDR 0x00
#define UFR 0x18

#define ARM_VERSATILE_PL011_UART0 0x101F1000
#define ARM_VERSATILE_PL011_UART3 0x10009000

typedef volatile struct uart {
    char *base;
    int n;
}UART;

int uart_init();
char ugetc(UART *up);
void uputc(UART *up, char c);
void upgets(UART *up, char *s);
void uprints(UART *up, char *s);

#endif
```

uart.c:

```
#include "uart.h"
UART uart[4];

int uart_init()
{
    int i;
    UART *up;
    for(i=0; i<4; i++){
        up = &uart[i];
        up->base = (char *)(ARM_VERSATILE_PL011_UART0 + i*0x1000);
        up->n = i;
    }
    uart[3].base = (char *)(ARM_VERSATILE_PL011_UART3);
```

```
}


char ugetc(UART *up)
{
    while(*(up->base + UFR) & RXFE);
    return *(up->base + UDR);
}

void uputc(UART *up, char c)
{
    while(*(up->base + UFR) & TXFF);
    *(up->base + UDR) = c;
}

void upgets(UART *up, char *s)
{
    while((*s = ugetc(up))!='\r'){
        uputc(up, *s);//echo the input from the UART back to the UART so
user can see what he has just input
        s++;
    }
    *s = 0;
}

void uprints(UART *up, char *s)
{
    while(*s) //output the whole buffer to UART
    {
        uputc(up, *s++);
    }
}
```

接下来，我们把上次的 main.c 文件里面的语句全部用我们写的这个驱动来替换吧：

```
#include "uart.h"

extern UART uart[4];
UART *up;

void __aeabi_unwind_cpp_pr0 (void) {}
void __aeabi_unwind_cpp_pr1 (void) {}
```

```
void softreset(void){
       /*Syscall exit*/
       asm(
    "mov %r0, $0\n"
       "mov %r7, $1\n"
       "swi $0\n"
    );
}

/* qemu-system-arm -M versatilepb -serial stdio -semihosting -kernel
test.bin*/
void shutdown(void){
    register int reg0 asm("r0");
    register int reg1 asm("r1");

    reg0 = 0x18;   // angel_SWIreason_ReportException
    reg1 = 0x20026; // ADP_Stopped_ApplicationExit

    asm("svc 0x00123456");  // make semihosting call
}
void menu(void){
    uprints(up,"*****************MENU*************************\n");
    uprints(up,"***1-press'a' for Checked up and good to go!***\n");
    uprints(up,"***2-press'b' for Orders sir!             ***\n");
    uprints(up,"***3-press'r' for Reset!                 ***\n");
    uprints(up,"***4-press'q' for SHUTDOWN!              ***\n");

uprints(up,"*********************************************\n\n");
}
void main_loop (void)
{
    unsigned char cmd;
    int i;
    char string[64];


    uart_init();
    up = &uart[0];
    uprints(up, "\n\rEnter lines from serial terminal 0\n\r");

    menu();

    for(;;) {
```

```
        cmd = ugetc(up);
    if(cmd<0x80 && cmd>0x60){
        switch(cmd){
            case 0x61:
            uprints(up,"Checked up and good to go!\n");
            break;
            case 0x62:
            uprints(up,"Orders sir!\n");
            break;
            case 0x71:
            shutdown();
            break;
            case 0x72:
            softreset();
            break;
            default:
            uprints(up,"Command unknown!\n");
            uprints(up,"Please press 'a' 'b' 'r' or 'q'!\n");
            break;
            }
        uprints(up,"\n");
        menu();
        uprints(up,"\n");
        }
    }
}
```

make run，运行测试，见图 1。

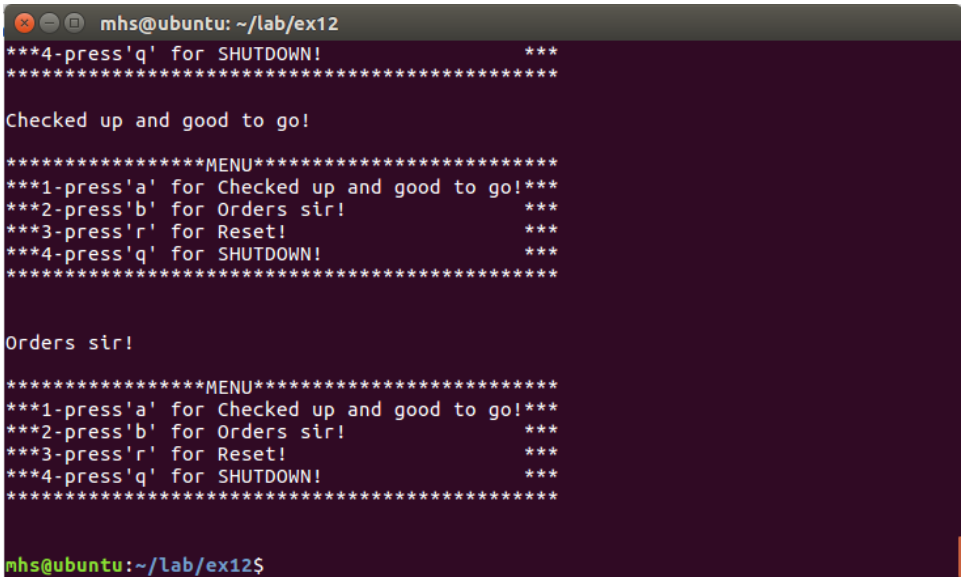

图 1 uart 驱动测试

看看，怎么样，我们用上了开发的第一个设备驱动程序了。

上面这个程序由于没有 C 类库可用，所以我添加了一个网上参照的 mystring 简单字符串处理库，添加到工程中，这样我们把 main.c 里面可以改成用户键入命令后，必须键入回车才会响应命令。

```
for(;;) {
        //cmd = ugetc(up);
        upgets(up, string);
        cmd = string[my_strlen(string)-1];/*读取回车符前最后一个字符*/
```

这样程序的用户交互体验是不是会好很多，不会再像早期的计算机一样，不能按消去符，按错就没法后悔。当然现在我们没有实现消去符的功能，这个留给大家思考吧。

3 framebuffer 图形接口驱动

ARM 板一般都支持彩色 LCD 显示屏，比如我们要仿真的这块 versatile 板其显示屏使用 ARM PL110 彩色 LCD 控制器（ARM PrimeCell 彩色 LCD 控制器 PL110，适用于 ARM926EJ-S 的 ARM versatile 应用基板。 在 versatile 板上，LCD 控制器位于基地址 0x10120000。它具有多个时序和控制寄存器，可以对其进行编程提供不同的显示模式和分辨率。 要使用 LCD 显示屏，控制器的定时和控制寄存器必须正确设置。 ARM 的 Versatile Application Baseboard 手册提供了 VGA 和 SVGA 模式的时序寄存器设置。

Mode	Resolution	OSC1	timeReg0	timeReg1	timeReg2
VGA	640x480	0x02C77	0x3F1F3F9C	0x090B61DF	0x067F1800
SVGA	800x600	0x02CAC	0x1313A4C4	0x0505F6F7	0x071F1800

我们来看一看彩色位图的格式，见表 2 所示。

现在我们要给原来光秃秃的仿真界面加点彩色的图像显示，所以现在我们要操纵系统的 fb 设备去进行绘图了。

Framebuffer 的驱动对于 ARM 来说是非常简单的，首先我们需要初始化 fb 的 LCDSYS 系统时钟：LCDCLK SYS_OSCCLK,然后指定 timer0、timer1、timer2 三个寄

存器的值，然后设定 fb 的内存大小地址映射，最后是设置控制寄存器。

如果要绘图，我们对映射后的 fb 进行操作就可以了，非常之简单。

表 2　BMP file format

Offset	Size	Description
---------- 14-byte file header ----------		
0	2	Signature ('MB')
2	4	Size of BMP file in bytes
6	2	Reserved (0)
8	2	Reserved (0)
10	4	Offset to start of image data in bytes
---------- 40-byte image header ----------		
14	4	Size of image header (40)
18	4	Image width in pixels
22	4	Image height in pixels
26	2	Number of image planes (1)
28	2	Number of bits per pixel (24)
---------- Other fields ----------		
50	4	Number of important colors (0)
---------- Rows of image image ----------		
54 to end of file: rows of images		

我们写的 LCD 的驱动代码如下：

```
/************** vid.c file**********************/
int volatile *fb;
int WIDTH = 640; // default to VGA mode for 640x480

int fbuf_init(void)
{
  int x; int i;
  /**** for SVGA 800X600 these values are in ARM DUI02241 *********
  *(volatile unsigned int *)(0x1000001c) = 0x2CAC; // 800x600
  *(volatile unsigned int *)(0x10120000) = 0x1313A4C4;
  *(volatile unsigned int *)(0x10120004) = 0x0505F6F7;
  *(volatile unsigned int *)(0x10120008) = 0x071F1800;
  *(volatile unsigned int *)(0x10120010) = (6*1024*1024);
  *(volatile unsigned int *)(0x10120018) = 0x82B;
  ************************************************************/

  /********* for VGA 640x480 ************************/
  *(volatile unsigned int *)(0x1000001c) = 0x2C77;       // LCDCLK
SYS_OSCCLK
  *(volatile unsigned int *)(0x10120000) = 0x3F1F3F9C;    // time0
  *(volatile unsigned int *)(0x10120004) = 0x090B61DF;    // time1
  *(volatile unsigned int *)(0x10120008) = 0x067F1800;    // time2
  *(volatile unsigned int *)(0x10120010) = (2*1024*1024); //
panelBaseAddress
```

```
  *(volatile unsigned int *)(0x10120018) = 0x82B;       // control
register
  /*************************************************************/

  /******** yet to figure out HOW TO use these palletes *************/
  fb = (int *)(2*1024*1024);  // at 2MB area; enough for VGA 640x480


  // for (x = 0; x < (800 * 600); ++x) // for one BAND
  /******** these will show 3 bands of BLUE, GREEN, RED *********/
  for (x = 0; x < (212*480); ++x)
    fb[x] = 0x00FF0000;  //00BBGGRR
  for (x = 212*480+1; x < (424*480); ++x)
    fb[x] = 0x0000FF00;  //00BBGGRR
  for (x = 424*480+1; x < (640*480); ++x)
    fb[x] = 0x000000FF;  //00BBGGRR

}
```

很简单吧，我们在主循环里开始用吧。在 main.c 里面我们写了个显示 24 位彩色位图 bmp 文件的函数：

```
extern char _binary_image1_start, _binary_image2_start;

#define WIDTH 640

int show_bmp(char *p, int start_row, int start_col)
{
    int h, w, pixel, rsize, i, j;
    unsigned char r, g, b;
    char *pp;
    int *q = (int *)(p+14); // skip over 14-byte file header
    w = *(q+1); // image width in pixels
    h = *(q+2); // image height in pixels
    p += 54; // p-> pixels in image
    //BMP images are upside down, each row is a multiple of 4 bytes
    rsize = 4*((3*w + 3)/4); // multiple of 4
    p += (h-1)*rsize; // last row of pixels
    for (i=start_row; i<start_row + h; i++){
    pp = p;
        for (j=start_col; j<start_col + w; j++){
        b = *pp; g = *(pp+1); r = *(pp+2); // BRG values
        pixel = (b<<16) | (g<<8) | r; // pixel value
        fb[i*WIDTH + j] = pixel; // write to frame buffer
```

```
            pp += 3; // advance pp to next pixel
            }
        p -= rsize; // to preceding row
        }
}
```

而_binary_image1_start, _binary_image2_start 这两个参数表示我们后面用到的两幅图片在内存中的起始地址，这个我们将在链接文件里去写在 data 段的定义那里。

在 main.c 里面，我们首先初始化 fb 驱动：

```
fbuf_init(); // default to VGA mode
```

然后我们在菜单'c'里面随便写了两句，让你按任意键的时候切换显示图片，如果按 i 键就退出循环。

演示结果见图 2，图 3 所示。

图 2 显示第一张图片

图 3 显示第二张图片

可能有人会问，这两个图片是存放在哪里的呢，那么我们来看看我们新的链接文件：

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
/*指定输出可执行文件 elf 格式，32 位 ARM 指令，小端模式*/
/*OUTPUT_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")*/
OUTPUT_ARCH(arm)    /*指定体系结构为 ARM*/
ENTRY(_start)   /*指定输出可执行文件的起始入口为_start*/
SEARCH_DIR("=/usr/local/lib"); SEARCH_DIR("=/lib");
SEARCH_DIR("=/usr/lib");
SECTIONS
{
    . = 0x10000;    /*定位当前地址为 0x10000 地址*/

    . = ALIGN(4);
    .text      :
    {
      start.o   (.text) /*第一个代码段来自目标文件 start.o*/
      *(.text)  /*其他代码段*/
    }

    . = ALIGN(4);
    .rodata : { *(.rodata) }    /*指定只读数据段*/

    . = ALIGN(4);

    .data   :             /*指定读写数据段*/
    {
    _binary_image1_start = .;
      huanghelou.o  (.data)    /* include huanghelou.o as a data section
*/
    _binary_image2_start = .;
      wust.o    (.data)    /* include wust.o as a data section */
      *(.data)
    }

    . = ALIGN(4);   /*指定 bss 段 静态内存分配*/
    __bss_start = .;
    .bss : { *(.bss) }

    . = ALIGN(8);   /*以八字节对齐*/
    . = . + 0x1000; /* 4kB of stack memory */
    stack_top = .;  /*指定栈顶指针地址*/
    _end = .;
```

```
}
```

可以看到我们数据段加载了 huanghelou.o 以及 wust.o，那这两个点 o 文件，则是在 Makefile 文件里面我们用 objcopy 命令生成的。

```
arm-none-linux-gnueabi-objcopy -I binary -O elf32-littlearm -B arm timg.bmp huanghelou.o

arm-none-linux-gnueabi-objcopy -I binary -O elf32-littlearm -B arm wust.bmp wust.o
```

好了，第四章 ARM 外围接口驱动实验我们就介绍这些，希望对大家理解硬件驱动有所帮助。