

## 嵌入式系统仿真实验第六讲

有了前面的一步步铺垫，我们要真正开始系统级的编程仿真调试了，嵌入式系统的软件开发，必不可少的需要掌握 bootloader+操作系统+驱动编程，最后到图形、多媒体、网络以及 IoT(物联网 Internet of Thing)，随着 AI 技术的兴起，发展到智联网（AIoT），说明嵌入式系统确实一直是未来发展的重要方向之一。

新冠肆虐，希望大家都能平安无事。上网课也不太了解大家真实的学习情况，上个星期在课程群简单弄了个调查问卷，结果如图 1。

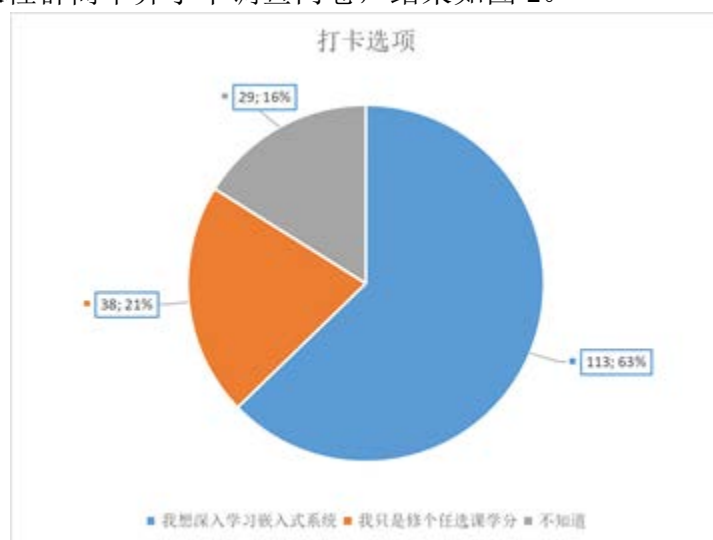



图 1 课程调查问卷统计

看到有这么多名同学是想深入学习嵌入式系统，我确实很欣慰😊。但不知道大家能不能坚持下去，嵌入式系统要学好，确实是较其他程序设计要难度大一些的。

所以我也坚持在给大家编示范代码的基础上，还是简洁的写这个教程文档，这样，即使关在家里没有电脑的同学，也可以用手机看 pdf 文档。

### 1 ARM 启动过程解析

**武汉科技大学**  
WUHAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

ARM Bootloader与代码分析

## 启动代码

功能

- 硬件初始化
- 引导C代码

特征

- 通常用汇编语言编写
- 程序复位运行入口点
- 代码量非常小

程序流程

- 设置中断、异常入口
- 关中断
- 硬件寄存器初始化（GPIO配置、总线配置、PLL时钟）
- 初始化栈指针
- 数据区初始化
- C入口函数调用 BL Main

厚德博学 求是笃行

图 2 ARM 启动代码功能、特征以及大致流程

我的讲义在第 6 章讲启动代码和 bootloader,大家可以对照我的讲义去学习,这部分内容可能比大家上的 mooc 网课要详细的多,课程实际教学可能由于课时限制,没法讲清,特别是 bootloader,一两个学时根本没法讲清楚,所以大家还是要课下多查资料,多学习,多实践才能掌握。

启动代码是用来初始化电路以及用来为高级语言写的软件做好运行前准备的一小段汇编语言程序,是任何处理器上电复位时的程序运行入口点。

引导加载程序是系统加电后运行的第一段软件代码。在 PC 的体系结构中,PC 机中的引导加载程序由 BIOS(其本质就是一段固件程序)和位于硬盘 MBR (Master Boot Record,主引导记录)中的 OS Boot Loader (比如: LILO 和 GRUB 以及我们这门课后面要讲的 uboot 等)一起组成。BIOS 在完成硬件检测和资源分配后,将硬盘 MBR 中的 BootLoader 读到系统的 RAM 中,然后将控制权交给 OS Boot Loader。BootLoader 的主要运行任务就是将内核映像从硬盘上读到 RAM 中,然后跳转到内核的入口点去运行,也即开始启动操作系统。

而在嵌入式系统中,通常并没有像 BIOS 那样的固件程序(注,有的嵌入式 CPU 也会内嵌一段短小的启动程序),因此整个系统的加载启动任务就完全由 BootLoader 来完成。比如在一个基于 ARM9TDMI core 的嵌入式系统中,系统在上电或复位时通常都从地址 0x00000000 处开始执行,而在这个地址处安排的通常就是系统的 Boot Loader 程序。

Boot Loader 是严重地依赖于硬件而实现的,特别是在嵌入式世界。因此,在嵌入式世界里建立一个通用的 Boot Loader 几乎是不可能的。尽管如此,还是有很多开源的 BootLoader 支持绝大多数的市面上的 CPU 架构的 CPU 核和处理器核的计算机。由于需要支持多种 CPU,所以它们都比较庞大,对于初学者,一上手就学这个,可能会适得其反。

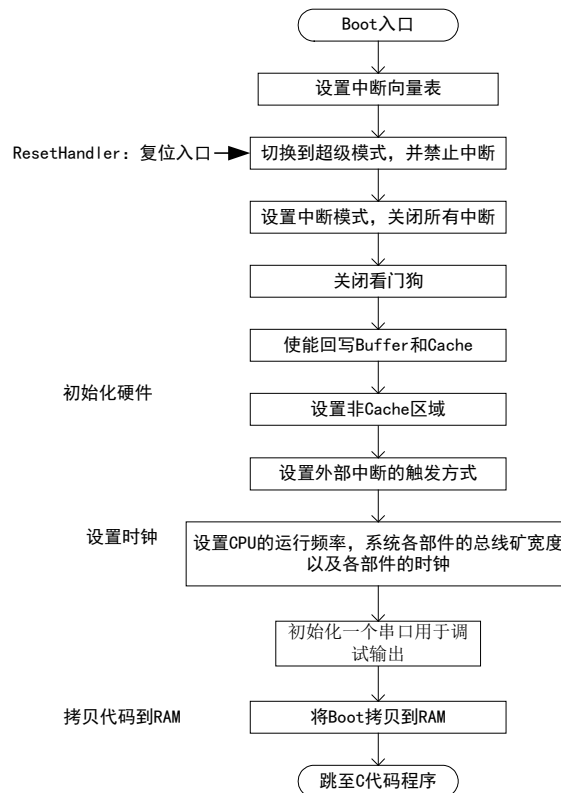


图 3 一般嵌入式系统启动流程图

所以我的教学一直都是先讲 Boot 然后再将操作系统和 Loader。从字面上我们就可以知道 BootLoader=Boot+Loader，Boot 的基本流程见图 3 所示。

网上有很多教程，教初学者如何编写启动代码，这都是很好的学习资料。但我们不是新冠病毒，所以今天不抄作业☺。我们从学习规范的代码开始。

U-Boot 在 CPU 架构方面支持 PowerPC、MIPS、x86、ARM、NIOS、XScale 等诸多常用处理器，市面上的商业软件操作系统几乎全支持。

那么我们就从 U-Boot 抽丝剥茧，开始先学 Boot 流程。（注我们参考的是 U-boot-1.1.6 的版本代码）

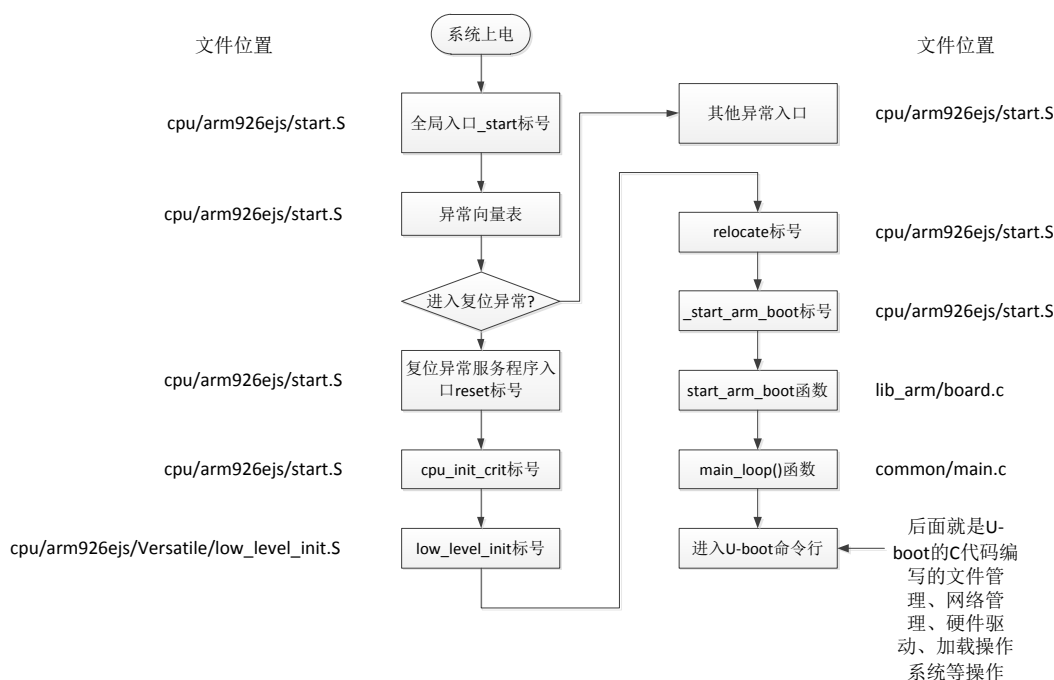


图 4 U-boot 的第一阶段流程代码分析

针对图 3 和图 4，我们可以看到不同的硬件操作顺序大致是一样的，涉及硬件的操作一般放在 cpu 目录下，用汇编编写，在进入 C 语言的主程序之前，针对不同的板子，写了个 low\_level\_init.S 文件，初始化数据区和设置总线带宽、CPU 运行节拍（锁相环参数），然后用汇编语言 relocate 去完成代码拷贝到 RAM。

我们前面一节课仿真用的 arm926ejs 核，qemu 对应的板子是 Versatile，但我看到这个虚拟的板子 low\_level\_init.S 里面什么都没有，所以我就有去看 board/smdk2410 这块板子的 low\_level\_init.S 文件，打算拿它来做实验。Smdk2410 是三星公司以它的 s3c2410 出的公板，是 ARM920T 核的嵌入式微处理器核，比 arm926ejs（v5TEJ 指令体系，增加了 Java 指令扩展）要早一个版本，属于 ARMv4 架构，但这个没有关系，我们只是拿它来做仿真实验，一些寄存器和总线大致是一致的，仿真是没有关系的，反正没有硬件，当然如果是硬件，你们正好需要看看硬件上到底有什么操作是不同的。

## 2 开始程序设计

从上面的分析我们看到仿照写四个源程序文件(start.S、low\_level\_init.S，board.c、main.c)就可以实现规范的 boot 流程了。

先把 start.S 拷贝过来，我们看看代码：  
原来的文件前面大致是这样的：

```
_start:
    b    reset
    ldr pc, _undefined_instruction
    ldr pc, _software_interrupt
    ldr pc, _prefetch_abort
    ldr pc, _data_abort
    ldr pc, _not_used
    ldr pc, _irq
    ldr pc, _fiq

_undefined_instruction:
    .word undefined_instruction
_software_interrupt:
    .word software_interrupt
_prefetch_abort:
    .word prefetch_abort
_data_abort:
    .word data_abort
_not_used:
    .word not_used
_irq:
    .word irq
_fiq:
    .word fiq

    .balignl 16, 0xdeadbeef
```

所有的异常他都写了入口函数，我们目前只处理 reset，所以我把其他的异常都屏蔽掉了，这样文件可以裁剪的更小一些。上代码：

Start.S

TEXT\_BASE = 0x00010000 @0x33F80000

/\*

\*\*\*\*\*

\*\*\*\*

\*

\* Jump vector table as in table 3.1 in [1]

\*

\*\*\*\*\*

\*\*\*\*

\*/

.globl \_start

\_start:

b reset

```

    b . /* Undefined */
    b . /* SWI */
    b . /* Prefetch Abort */
    b . /* Data Abort */
    b . /* reserved */
    b . /* IRQ */
    b . /* FIQ */

    .balignl 16,0xdeadbeef

/*

*****
****
*
* Startup Code (reset vector)
*
* do important init only if we don't start from memory!
* setup Memory and board specific bits prior to relocation.
* relocate armboot to ram
* setup stack
*

*****
****
*/

_TEXT_BASE:
    .word    TEXT_BASE

.globl _armboot_start
_armboot_start:
    .word _start

/*
* These are defined in the board-specific linker script.
*/
.globl _bss_start
_bss_start:
    .word __bss_start

.globl _bss_end
_bss_end:
    .word _end

```

```

#ifdef CONFIG_USE_IRQ
/* IRQ stack memory (calculated at run-time) */
.globl IRQ_STACK_START
IRQ_STACK_START:
    .word 0x0badc0de

/* IRQ stack memory (calculated at run-time) */
.globl FIQ_STACK_START
FIQ_STACK_START:
    .word 0x0badc0de
#endif

/*
 * the actual reset code
 */

reset:
    /*
     * set the cpu to SVC32 mode
     */
    mrs r0,cpsr
    bic r0,r0,#0x1f
    orr r0,r0,#0xd3
    msr cpsr,r0

    /*
     * we do sys-critical inits only at reboot,
     * not when booting from ram!
     */
#ifdef CONFIG_SKIP_LOWLEVEL_INIT
    bl cpu_init_crit
#endif

#ifdef CONFIG_SKIP_RELOCATE_UBOOT
relocate:                /* relocate U-Boot to RAM */
    adr r0, _start        /* r0 <- current position of code */
    ldr r1, _TEXT_BASE     /* test if we run from flash or RAM */
    cmp r0, r1             /* don't reloc during debug */
    beq stack_setup
                            */

    ldr r2, _armboot_start
    ldr r3, _bss_start

```

```

    sub r2, r3, r2      /* r2 <- size of armboot      */
    add r2, r0, r2      /* r2 <- source end address      */

copy_loop:
    ldmia r0!, {r3-r10} /* copy from source address [r0] */
    stmia r1!, {r3-r10} /* copy to target address [r1] */
    cmpr0, r2           /* until source end addreee [r2] */
    ble copy_loop
#endif /* CONFIG_SKIP_RELOCATE_UBOOT */

    /* Set up the stack */
stack_setup:
@   ldr r0, _TEXT_BASE /* upper 128 KiB: relocated uboot */
@   sub r0, r0, #384 << 10 /*#CFG_MALLOC_LEN*/ /* malloc area */
@   sub r0, r0, #64 /*#CFG_GBL_DATA_SIZE*/ /* bdfinfo */

@   sub sp, r0, #12 /* leave 3 words for abort-stack */

    ldr sp, =stack_top /*modified by Huasong Min */

clear_bss:
    ldr r0, _bss_start /* find start of bss segment */
    ldr r1, _bss_end /* stop here */
    mov r2, #0x00000000 /* clear */

clbss_l:str r2, [r0] /* clear loop... */
    add r0, r0, #4
    cmpr0, r1
    ble clbss_l

    ldr pc, _start_armboot
    b . /*added by Huasong Min */

_start_armboot:
    .word start_armboot

/*

*****
****

```

```

*
* CPU_init_critical registers
*
* setup important registers
* setup memory timing
*

*****
****
*/

cpu_init_crit:
/*
 * flush v4 I/D caches
 */
mov    r0, #0
mcr p15, 0, r0, c7, c7, 0 /* flush v3/v4 cache */
mcr p15, 0, r0, c8, c7, 0 /* flush v4 TLB */

/*
 * disable MMU stuff and caches
 */
mrc p15, 0, r0, c1, c0, 0
bic r0, r0, #0x00002300 /* clear bits 13, 9:8 (--V- --RS) */
bic r0, r0, #0x00000087 /* clear bits 7, 2:0 (B--- -CAM) */
orr r0, r0, #0x00000002 /* set bit 2 (A) Align */
orr r0, r0, #0x00001000 /* set bit 12 (I) I-Cache */
mcr p15, 0, r0, c1, c0, 0

/*
 * Go setup Memory and board specific bits prior to relocation.
 */
mov    ip, lr /* perserve link reg across call */
bl    lowlevel_init /* go setup pll,mux,memory */
mov    lr, ip /* restore link */
mov    pc, lr /* back to my caller */

```

看看人家的代码写的多么规范，都有注释，大家对照我的讲义画的那个流程图，很快就可理解掌握。

哦，对了，我除了删掉无关的异常处理之外，修改了堆栈初始化那块，把他原来的那段注释掉了，因为我也不知道目前的硬件配置情况，所以干脆把堆栈指针赋值给在链接文件 **map.lds** 里面设置的栈顶地址，这个我在上一次的实验里有讲：



```
ldr sp, =stack_top
```

接下来是 low\_level\_init.S 文件:

```
/* some parameters for the board */
```

```
/*
```

```
*
```

```
* Taken from linux/arch/arm/boot/compressed/head-s3c2410.S
```

```
*
```

```
*      Copyright      (C)      2002      Samsung      Electronics      SW.LEE
```

```
<hitchcar@sec.samsung.com>
```

```
*
```

```
*/
```

```
TEXT_BASE = 0x00010000 @0x33F80000
```

```
BWSCON = 0x48000000
```

```
/* BWSCON */
```

```
DW8      =      (0x0)
```

```
DW16     =      (0x1)
```

```
DW32     =      (0x2)
```

```
WAIT     =      (0x1<<2)
```

```
UBLB     =      (0x1<<3)
```

```
B1_BWSCON = (DW32)
```

```
B2_BWSCON = (DW16)
```

```
B3_BWSCON = (DW16 + WAIT + UBLB)
```

```
B4_BWSCON = (DW16)
```

```
B5_BWSCON = (DW16)
```

```
B6_BWSCON = (DW32)
```

```
B7_BWSCON = (DW32)
```

```
/* BANK0CON */
```

```
B0_Tacs  =  0x0 /* 0clk */
```

```
B0_Tcos  =  0x0 /* 0clk */
```

```
B0_Tacc  =  0x7 /* 14clk */
```

```
B0_Tcoh  =  0x0 /* 0clk */
```

```
B0_Tah   =  0x0 /* 0clk */
```

```
B0_Tacp  =  0x0
```

```
B0_PMC   =  0x0 /* normal */
```

```
/* BANK1CON */
```

```
B1_Tacs  =  0x0 /* 0clk */
```

```
B1_Tcos  =  0x0 /* 0clk */
```

B1\_Tacc = 0x7 /\* 14clk \*/  
 B1\_Tcoh = 0x0 /\* 0clk \*/  
 B1\_Tah = 0x0 /\* 0clk \*/  
 B1\_Tacp = 0x0  
 B1\_PMC = 0x0

B2\_Tacs = 0x0  
 B2\_Tcos = 0x0  
 B2\_Tacc = 0x7  
 B2\_Tcoh = 0x0  
 B2\_Tah = 0x0  
 B2\_Tacp = 0x0  
 B2\_PMC = 0x0

B3\_Tacs = 0x0 /\* 0clk \*/  
 B3\_Tcos = 0x3 /\* 4clk \*/  
 B3\_Tacc = 0x7 /\* 14clk \*/  
 B3\_Tcoh = 0x1 /\* 1clk \*/  
 B3\_Tah = 0x0 /\* 0clk \*/  
 B3\_Tacp = 0x3 /\* 6clk \*/  
 B3\_PMC = 0x0 /\* normal \*/

B4\_Tacs = 0x0 /\* 0clk \*/  
 B4\_Tcos = 0x0 /\* 0clk \*/  
 B4\_Tacc = 0x7 /\* 14clk \*/  
 B4\_Tcoh = 0x0 /\* 0clk \*/  
 B4\_Tah = 0x0 /\* 0clk \*/  
 B4\_Tacp = 0x0  
 B4\_PMC = 0x0 /\* normal \*/

B5\_Tacs = 0x0 /\* 0clk \*/  
 B5\_Tcos = 0x0 /\* 0clk \*/  
 B5\_Tacc = 0x7 /\* 14clk \*/  
 B5\_Tcoh = 0x0 /\* 0clk \*/  
 B5\_Tah = 0x0 /\* 0clk \*/  
 B5\_Tacp = 0x0  
 B5\_PMC = 0x0 /\* normal \*/

B6\_MT = 0x3 /\* SDRAM \*/  
 B6\_Trtd = 0x1  
 B6\_SCAN = 0x1 /\* 9bit \*/

B7\_MT = 0x3 /\* SDRAM \*/  
 B7\_Trtd = 0x1 /\* 3clk \*/

```
B7_SCAN      = 0x1 /* 9bit */
```

```
/* REFRESH parameter */
```

```
REFEN        = 0x1 /* Refresh enable */
```

```
TREFMD       = 0x0 /* CBR(CAS before RAS)/Auto refresh */
```

```
Trp          = 0x0 /* 2clk */
```

```
Trc          = 0x3 /* 7clk */
```

```
Tchr         = 0x2 /* 3clk */
```

```
REFCNT       = 1113 /* period=15.6us, HCLK=60Mhz, (2048+1-15.6*60) */
```

```
/* **** */
```

```
_TEXT_BASE:
```

```
.word TEXT_BASE
```

```
.globl lowlevel_init
```

```
lowlevel_init:
```

```
/* memory control configuration */
```

```
/* make r0 relative the current location so that it */
```

```
/* reads SMRDATA out of FLASH rather than memory ! */
```

```
ldr r0, =SMRDATA
```

```
ldr r1, _TEXT_BASE
```

```
sub r0, r0, r1
```

```
ldr r1, =BWSCON /* Bus Width Status Controller */
```

```
add r2, r0, #13*4
```

```
0:
```

```
ldr r3, [r0], #4
```

```
str r3, [r1], #4
```

```
cmp r2, r0
```

```
bne 0b
```

```
/* everything is fine now */
```

```
mov pc, lr
```

```
.ltorg
```

```
/* the literal pools origin */
```

```
SMRDATA:
```

```
.word
```

```
(0+(B1_BWSCON<<4)+(B2_BWSCON<<8)+(B3_BWSCON<<12)+(B4_BWSCON<<16)+(B5_BWSCON<<20)+(B6_BWSCON<<24)+(B7_BWSCON<<28))
```

```
.word
```

```
((B0_Tacs<<13)+(B0_Tcos<<11)+(B0_Tacc<<8)+(B0_Tcoh<<6)+(B0_Tah<<4)+(B0_Tacp<<2)+(B0_PMC))
```

```
.word
```

```

((B1_Tacs<<13)+(B1_Tcos<<11)+(B1_Tacc<<8)+(B1_Tcoh<<6)+(B1_Tah<<4)+(B1_Tacp
<<2)+(B1_PMC))
    .word
((B2_Tacs<<13)+(B2_Tcos<<11)+(B2_Tacc<<8)+(B2_Tcoh<<6)+(B2_Tah<<4)+(B2_Tacp
<<2)+(B2_PMC))
    .word
((B3_Tacs<<13)+(B3_Tcos<<11)+(B3_Tacc<<8)+(B3_Tcoh<<6)+(B3_Tah<<4)+(B3_Tacp
<<2)+(B3_PMC))
    .word
((B4_Tacs<<13)+(B4_Tcos<<11)+(B4_Tacc<<8)+(B4_Tcoh<<6)+(B4_Tah<<4)+(B4_Tacp
<<2)+(B4_PMC))
    .word
((B5_Tacs<<13)+(B5_Tcos<<11)+(B5_Tacc<<8)+(B5_Tcoh<<6)+(B5_Tah<<4)+(B5_Tacp
<<2)+(B5_PMC))
    .word ((B6_MT<<15)+(B6_Trcd<<2)+(B6_SCAN))
    .word ((B7_MT<<15)+(B7_Trcd<<2)+(B7_SCAN))
    .word
((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Trc<<18)+(Tchr<<16)+REFCNT)
    .word 0x32
    .word 0x30
    .word 0x30

```

大家可以看到这个文件我用的是 **smdk2410** 这个板子的，我完全没有去修改它，这个文件操作的是系统时钟相关的寄存器、带宽、锁相环参数设定等操作。然后从启动 flash 中把数据读出进行内存映射，读之前需要计算 **SMRDATA** 大小，**SMRDATA** 是存放内存控制参数的起始地址，这些 32 位的参数需要连续放置，通过循环依次将 **SMRDATA** 的值赋给内存控制寄存器进行初始化，完成内存映射。

程序最前面有一句：

```
TEXT_BASE = 0x00010000
```

这句是我加的，前面教程我已经讲过，qemu 起始地址是 0x00010000，所以这里我直接赋值，而不是通过 config 去配置。

lowlevel\_init 完成后接下来返回到 start.S 里面执行 relocate 那段搬运代码到 RAM，然后设置堆栈，清 bss 段，clear loop 完，就转入 start\_armboot，到了 board.c 文件里面，u-boot 的 board.c 程序里面做了很多处理，这里我不要它了，我写个最简单的程序，采用一个串口，然后打印一条信息，然后转去 main.c 里面的 main\_loop() 函数去执行。

board.c:

```

volatile unsigned int * const UART0DR = (unsigned int *)0x101f1000;

void print_uart0(const char *s) {
    while(*s != '\0') { /* Loop until end of string */
        *UART0DR = (unsigned int)(*s); /* Transmit char */
        s++; /* Next char */
    }
}

```

```

void start_armboot (void)
{
    print_uart0("boot!\n");
    main_loop ();
}

```

`print_uart0` 函数我已经在前面给大家都用过了，它是 `qemu` 给 `arm926ejs` 缺省初始化了的串口 0，大家如果学习到网课的外部接口这一章的话，正好可以体会一下串口的操作，请大家完成串口的初始化以及数据发送接收操作。

接下来 `main.c` 就是我们要写的开始让计算机干活的 C 语言程序了，下面是我自己写的一段程序，这个是我阅读 `arm926ejs` 的资料得出来的简洁串口发送接收命令，进行菜单操作的示范程序。

```

/*
 * armboot - Startup Code for ARM926EJS CPU-core
 *
 * Copyright (c) 2020 WUST
 *
 * Copyright (c) 2020 Huasong Min <mhuasong@wust.edu.cn>
 *
 * See file CREDITS for list of people who contributed to this
 * project.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the IRIS of
 * WUST, Wuhan University of Sci&Tech, 947 Heping Road, qinshan, Wuhan,
 * 430081 China
 */

#include <stdint.h>

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

```

```

typedef volatile struct {
    uint32_t DR;
    uint32_t RSR_ECR;
    uint8_t reserved1[0x10];
    const uint32_t FR;
    uint8_t reserved2[0x4];
    uint32_t LPR;
    uint32_t IBRD;
    uint32_t FBRD;
    uint32_t LCR_H;
    uint32_t CR;
    uint32_t IFLS;
    uint32_t IMSC;
    const uint32_t RIS;
    const uint32_t MIS;
    uint32_t ICR;
    uint32_t DMACR;
}pl011_T;

enum {
    RXFE = 0x10,
    TXFF = 0x20,
};

pl011_T * const UART0 = (pl011_T *)0x101f1000;
pl011_T * const UART1 = (pl011_T *)0x101f2000;
pl011_T * const UART2 = (pl011_T *)0x101f3000;

static inline char upperchar(char c) {
    if((c >= 'a') && (c <= 'z')) {
        return c - 'a' + 'A';
    } else {
        return c;
    }
}

static void uart_echo(pl011_T *uart) {
    if ((uart->FR & RXFE) == 0) {
        while(uart->FR & TXFF);
        uart->DR = upperchar(uart->DR);
    }
}

unsigned char getc(pl011_T *uart){

```

```

        if ((uart->FR & RXFE) == 0) {
            while(uart->FR & TXFF);
            //uart->DR = upperchar(uart->DR);
            return uart->DR;
        }
    }

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){
    print_uart0("*****MENU*****\n");
;
    print_uart0("***1-press'a' for Checked up and good to go!***\n");
    print_uart0("***2-press'b' for Orders sir!                ***\n");
    print_uart0("***3-press'r' for Reset!                    ***\n");
    print_uart0("***4-press'q' for SHUTDOWN!                  ***\n");
    print_uart0("*****\n\n");
};

}

void main_loop (void)
{
    unsigned char cmd;

    menu();

    for(;;) {

```

```

        cmd = getc(UART0);
    if(cmd<0x80 && cmd>0x60){
        switch(cmd){
            case 0x61:
                print_uart0("Checked up and good to go!\n");
                break;
            case 0x62:
                print_uart0("Orders sir!\n");
                break;
            case 0x71:
                shutdown();
                break;
            case 0x72:
                softreset();
                break;
            default:
                print_uart0("Command unknown!\n");
                print_uart0("Please press 'a' 'b' 'r' or 'q'!\n");
                break;
        }
        print_uart0("\n");
        menu();
        print_uart0("\n");
    }
}

```

程序相对简单,力求大家能掌握原理,由于我们目前为止没有C语言库可用,所以直接用串口来进行交互。

程序代码编完了,编译之前还有一个链接文件没有写, `map.lds` 文件上次课我已经讲过了,这次也没什么特别的,代码如下:

```

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 : { *(.data) }        /*指定读写数据段*/

. = 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 = .;
}

```

和上次没什么变化吧,接下来要开始编译调试了,我习惯还是写个 Makefile:

```

all:
    arm-none-linux-gnueabi-as -mcpu=arm926ej-s -g start.S -o start.o
    arm-none-linux-gnueabi-as -mcpu=arm926ej-s -g low_level_init.S -o
low_level_init.o
    arm-none-linux-gnueabi-gcc -c -mcpu=arm926ej-s -g board.c -o board.o
#-nostdlib
    arm-none-linux-gnueabi-gcc -c -mcpu=arm926ej-s -g main.c -o main.o
-static

    arm-none-linux-gnueabi-ld -T map.lds main.o board.o low_level_init.o
start.o -o test.elf
    arm-none-linux-gnueabi-objcopy -O binary test.elf test.bin

test: all
    qemu-system-arm -M versatilepb -m 128M -gdb tcp::1024 -serial stdio
-kernel test.elf -S &
    sleep 3
    ddd --debugger arm-none-linux-gnueabi-gdb test.elf

run:    all
    #qemu-system-arm -M versatilepb -serial stdio -semihosting -kernel
test.bin

```

clean:

```
rm -rf *.o test.elf test.bin
```

### 3 调试运行

和上次几乎一样的 Makefile，可以用于调试，但这次情况有点特别，因为我们用的 `versatilepb` 这个板子去仿真的，交互输入输出用的 `uart0`，所以仿真的时候添加了 `-serial stdio` 参数，而且在系统中 `qemu` 每次程序如果运行完后没法用 `exit` 退出，所以我写了两个函数，一个是退出 `test` 程序，但 `qemu` 会直接又进入 `reset`，所以我把这个函数命令“软复位” `softreset`，而另一函数 `shutdown` 配合 `-semihosting` 可以完全退出 `qemu`，这样免得大家总是要先在 `qemu` 按 `q` 退出才能退出运行。

直接执行一下：

make run

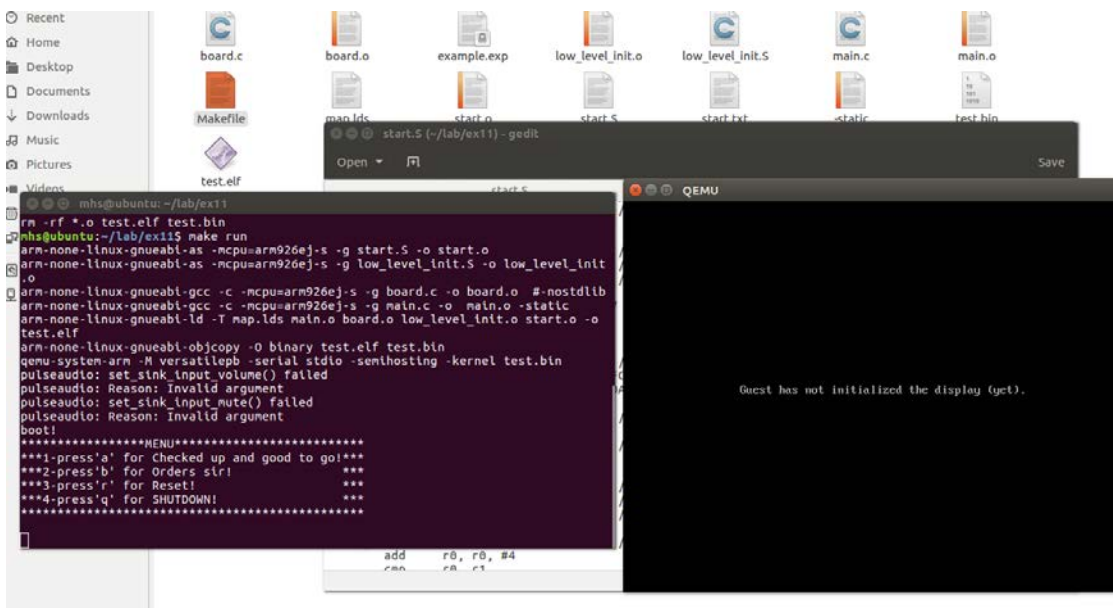


图5 运行界面

看看，直接自动编译、链接，运行，出现了一个 `qemu` 的仿真黑窗，然后启动 `boot` 后打印了菜单。

```
mhs@ubuntu: ~/lab/ex11
***4-press'q' for SHUTDOWN!***
*****
Checked up and good to go!

*****MENU*****
***1-press'a' for Checked up and good to go!***
***2-press'b' for Orders sir!***
***3-press'r' for Reset!***
***4-press'q' for SHUTDOWN!***
*****

Orders sir!

*****MENU*****
***1-press'a' for Checked up and good to go!***
***2-press'b' for Orders sir!***
***3-press'r' for Reset!***
***4-press'q' for SHUTDOWN!***
*****
```

图6 菜单测试

按'a'，会打印'Check up and good to go!'，按'b'会打印'Orders sir!'，按'r'会重启，按'q'直接退出了，haha，这几句是哪里的对话，你们知道么？☺☺☺

看看 ddd 调试如何：

make test

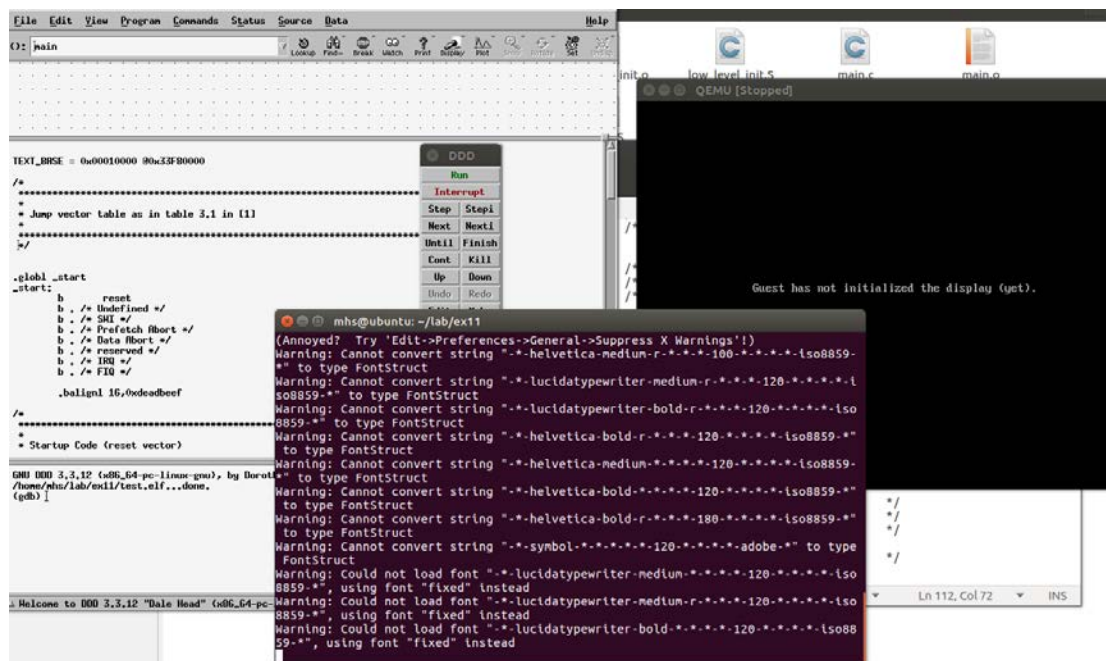


图7 调试界面成功启动

target remote localhost:1024 连连看：

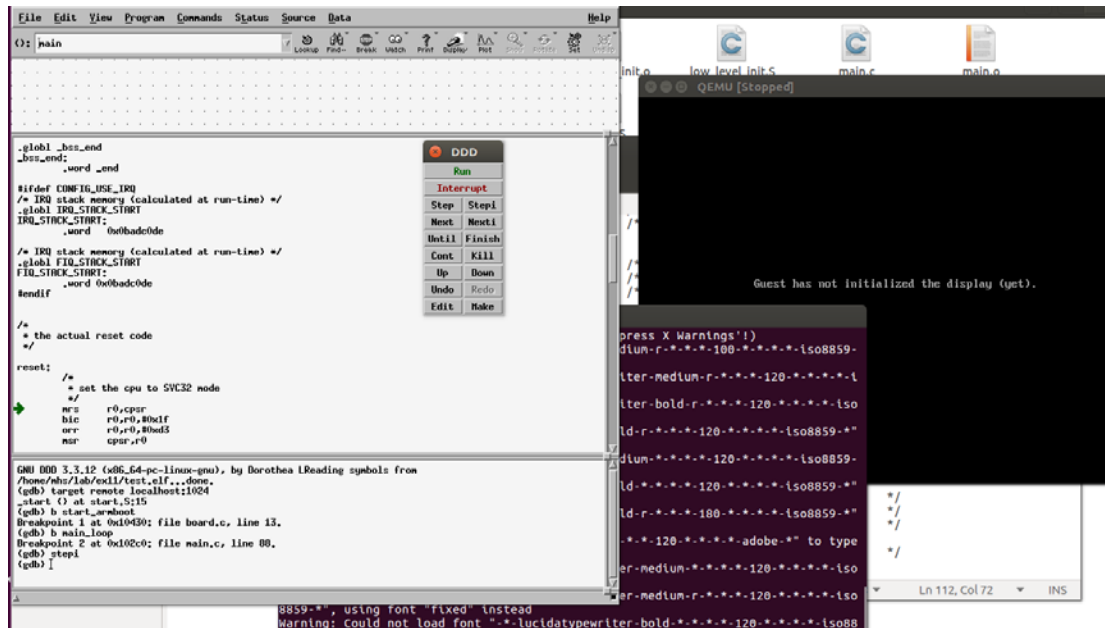


图 8 连接 gdb,设置断点

看到连接成功, 设置断点了成功, pc 停留在 reset 第一句, 接下来你就可以随心调试了。

我在 qemu 的窗口上按 ctrl+alt+2, 然后出来输入命令界面, 按个 c 回车, 可以看到程序运行出菜单了。

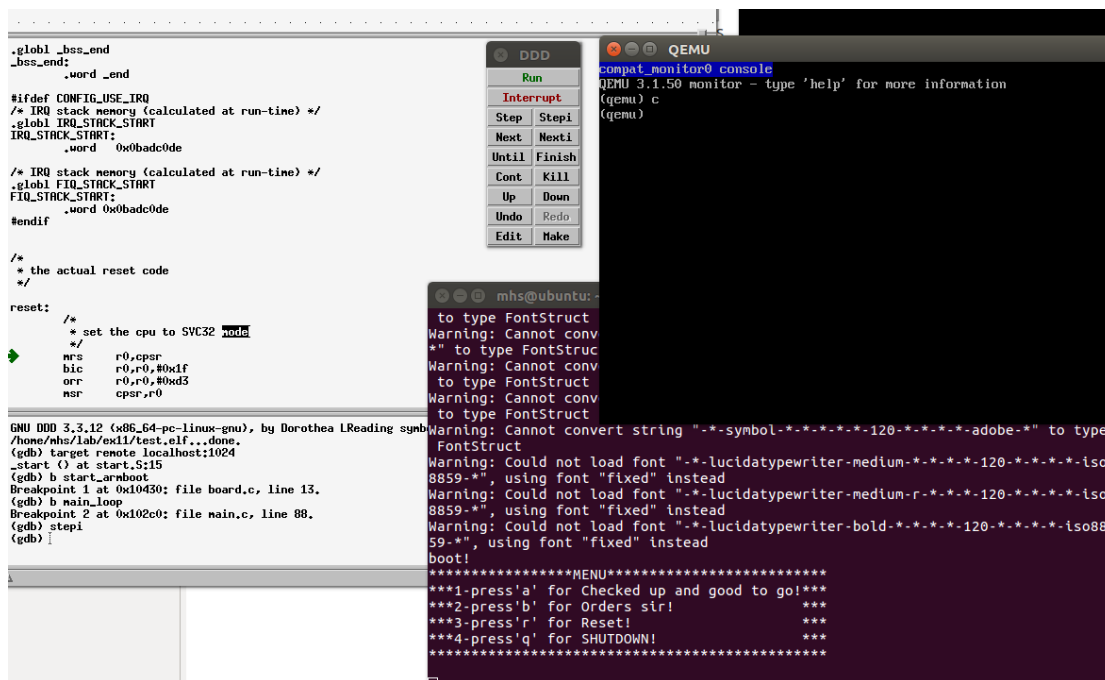


图 9 交互调试

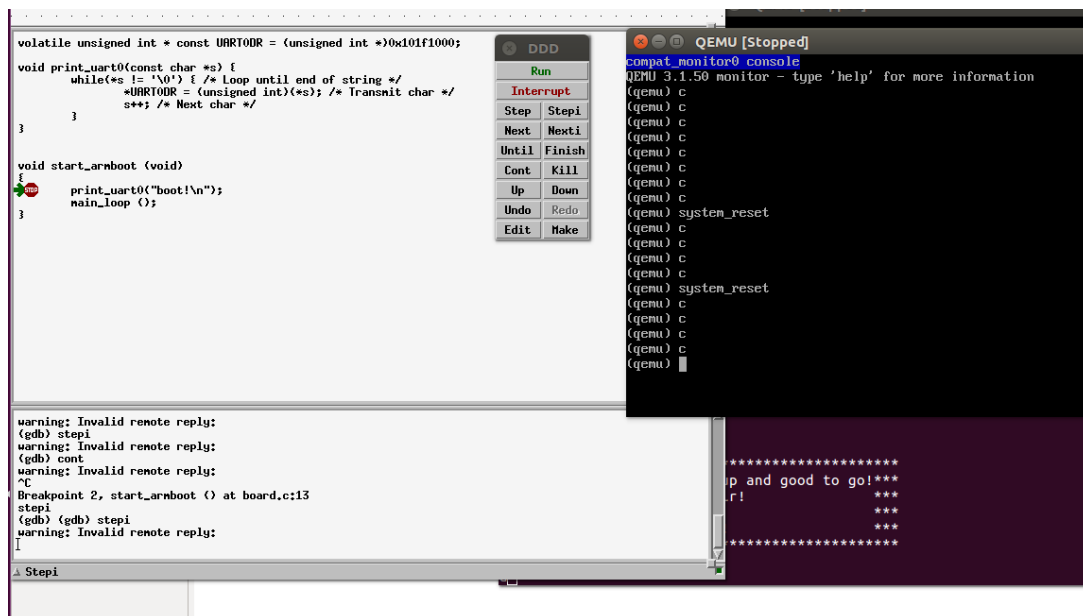


图 10 捕捉断点调试

qemu 的常用调试命令：

#挂起虚拟机：

(qemu)stop

#恢复刚挂起的虚拟机：

(qemu)cont

#马上关闭 qemu：

(qemu)q 或

(qemu)quit

#模拟按物理按键 reset 重启虚拟机：

(qemu)system\_reset

#模拟按物理按键 power 关闭虚拟机：

(qemu)system\_powerdown

从图 10 可以看到，调试小型程序还是比较方便的。

注：本实验教程为武汉科技大学机器人与智能系统研究院闵华松老师的网络课程教学文档，可以复制，不做商业用途。