**U-Boot的启动流程**

课程名称：嵌入式硬件设计与实践

学号：1170400423

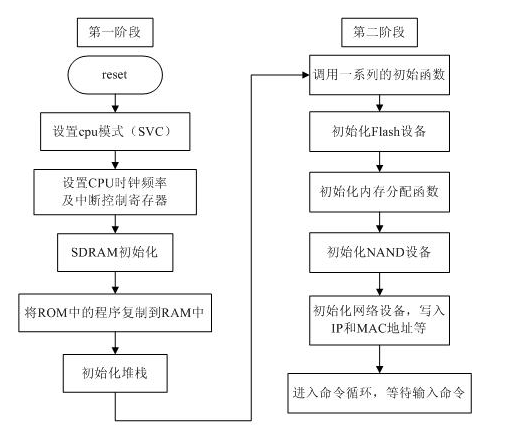
姓名：尉前进

日期：2020.6.22

### U-boot启动流程介绍

### 一、启动流程概述

U-Boot 启动内核的过程可以分为两个阶段，阶段一由汇编语言编写，阶段二由C语言编写，启动流程如下



### 二、实现代码

1. **第一阶段汇编代码**

1）硬件设备初始化将CPU的工作模式设为管理模式(svc）,关闭MMU、CACHE,设置外设寄存器地址。

2）为第二阶段代码准备RAM空间所谓准备RAM空间,初始化内存芯片,使它可用。通过在start.S中调用lowlevel init 函数来设置存储控制器,使得外接的SDRAM可用,并且关闭WATCHDOG,设置FCLK、HCLK、PCLK的比例(即设置CLKDIVN寄存器)。

**1.中断向量表的设置：**

.globl \_start /\*声明一个符号可以被其他文件引用，相当于声明了一个全局变量，.globl与.global相同\*/

\_start:

b start\_code /\* 复位 \*/

ldr pc, \_undefined\_instruction /\* 未定义指令向量 \*/

ldr pc, \_software\_interrupt /\* 软件中断向量 \*/

ldr pc, \_prefetch\_abort /\* 预取指令异常向量 \*/

ldr pc, \_data\_abort /\* 数据操作异常向量 \*/

ldr pc, \_not\_used /\* 未使用 \*/

ldr pc, \_irq /\* irq 中断向量 \*/

ldr pc, \_fiq /\* fiq 中断向量 \*/

/\* 中断向量表入口地址 \*/

注： Start.s文件一开始，就定义了\_start的全局变量。也即，在别的文件，照样能引用这个\_start变量。这段代码验证了我们之前学过的arm体系的理论知识：中断向量表放在从0x0开始的地方。其中，每个异常中断的摆放次序，是事先规定的。比如第一个必须是reset异常，第二个必须是未定义的指令异常等等。

需要注意的是，在这里，我们也可以理解：为何系统一上电，会自动运行代码。因为系统上电后，会从0x0地方取指令，而0x0处放置的是reset标签，直接就跳去reset标签处去启动系统了。

另外，这里使用了ldr指令。而ldr指令中的label，分别用一个.word伪操作来定义。比如:

\_undefined\_instruction: .word undefined\_instruction

我们用source insight跟踪代码后，发现，undefined\_instruction在start.s的后面给出了具体的操作，如下：

undefined\_instruction:

get\_bad\_stack

bad\_save\_user\_regs

bl do\_undefined\_instruction

在跳转到中断服务子程序之前，先有两个宏代码，一个是对stack的操作，一个是用户regs的保存。然后才能跳转如中断服务子程序中执行。

reset:

/\*\* set the cpu to SVC32 mode \*/ /\* CPU 进入 SVC （管理）模式 \*/

mrs r0,cpsr

r0,r0,#0x1f

bic r0,r0,#01f

msr cpsr,r0

cpu\_init\_crit:

/\* flush v4 I/D caches\*/ /\* CPU数据初始化，数据缓冲\*/

mov r0,#(

/\*flush v3/v4 cache \*/

mcr p15,0,r0,c7,c7,0

/\* flush v4 TLB\*/

mcr p15,0,r0,c8,c7,0

注： 关闭数据预取功能；DSB：多核CPU对数据处理指令；ISB：流水线清空指令；关闭MMU，使能I-cache。分支预测：在流水线里，会将后面的代码优先加载到处理器中，由于是循环，会使后面加载的代码无效，故出现了分支预测技术。

/\*\* disable MMU stuff and caches \*/ /\*通过p15关闭mmu\*/

mrC p15,0,r0,c1,c0,0

bic r0,r0,#0x00002300

bic r0,r0,#0x00000087

orr r0,r0,#0x00000002

orr r0,r0,#0x00001000

mcr p15,0,r0,c1,c0,0

/\* Peri port setup\*/ /\* 外设内存空间\*/

ldr r0,=0x70000000

orrr0,r0,#0x13

mcrp15,0,r0,c15,c2,4

@ 256M(0x7000 0000-0x7fff ffff) /\*指明地址\*/

/\* go setup pll,mux,memory \*/

bl lowlevel init

1. 上电后CPU为SVC模式

reset: /\*set the CPU to SVC32 mode \*/

mrs r0,cpsr

bic r0,r0,#0x1f

orr r0,r0,#0xd3

msr cpsr,r0

注：CPU复位后，系统会立即被设置成SVC模式。记得之前有网友发帖咨询这个问题，问系统复位后，cpu处于哪个处理器模式。这个代码，就回答了这个问题。从这个代码中，我们也可以得到一个对寄存器操作的经验：读—修改--写。这里先把cpsr的值读到r0中，清除掉我们想修改的bit位，然后用orr指令来保证其他bit位不被改动，并达到修改寄存器低5位值的目的。最后用msr指令把r0的值给cpsr寄存器达到我们的修改目的。

1. 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

bic r0, r0, #0x00000087

orr r0, r0, #0x00000002

orr r0, r0, #0x00001000

mcr p15, 0, r0, c1, c0, 0

/\* before relocating, we have to setup RAM timing because memory timing is board-dependend, you will find a lowlevel\_init.S in your board directory. \*/

mov ip, lr

bl lowlevel\_init

mov lr, ip

mov pc, lr

#endif /\* CONFIG\_SKIP\_LOWLEVEL\_INIT \*/

无效掉了指令cache和数据cache，并禁止MMU与cache。为什么会有这一步呢？这里无效cache和MMU肯定的原因：在初始化阶段，可以认为我们只有一个任务在跑，没有必要，也不允许使用地址变换。因此最好应该无效掉MMU。由于在cpu\_init\_cri子程序中又一次调用子程序lowlevel\_init，因此，需要事先保护好lr寄存器的内容。当返回时候，再恢复它。在进入lowlevel\_init之前，有必要详细说一下mov ip, lr，这个语句的ip。为了使单独编译的C语言程序和汇编程序之间能相互调用，必须为子程序间的调用规定一定的规则。这就是ATPCS规则。它规定了一些子程序间调用的基本规则。在寄存器的使用规则里，寄存器R12作用子程序间的scratch寄存器，记做ip。mov ip, lr语句的ip由此而来。

.globl lowlevel init

lowlevel init:mov r12,lr

/\* LED on only #8\*/ /\* LED上电\*/

ldrr0,=ELFIN GPIO BASE ldr r1,=0x55540000

strr1,[r0,#GPNCON\_OFFSET]

/\* Disable Watchdog\*/

ldrr0, =0x7e000000

@0x7e004000 /\* 关闭看门狗\*/

orrr0,rO,#0x4000

moVr1,#0

str r1,[r0]

/\* 看门狗控制器的最低位为 0 时，看门狗不输出复位信号 \*/

@ External interrupt pending clear

ldr r0,

=(ELFIN GPIO BASE+EINTPEND OFFSET)

/\*EINTPEND\*/

r r1,[r0]

str r1,[r0] /\* 关闭所有中断\*/

@ Disable all interrupts(VIc0 and VIc1)

mVn r3,#0x0

str r3,[r0,#oINTMSK]

str r3,[r1,#oINTMSK] /\* 将所有中断设置为IRQ\*/

@ Set all interrupts as IRQ

movr3,#0x0

strr3,[r0,#oINTMOD]

strr3,[r1,#oINTMOD]

@ Pending Interrupt clear /\*清除所有中断\*/

/\* init system clock \*/ /\*初始化内存\*/

bl system\_clock\_init

/\* for UART\*/ /\* 初始化时钟dll\*/

bl uart\_asm\_init

bl mem\_ctrl\_asm init

if 1

ldr r0

=(ELFIN CLOCK POWER BASE+RST STAT OFFSET）E1,[r0]

ldr

r1,r1,#0xfffffff7

bic

cmp r1,#0x8

beq wakeup\_ reset /\* 唤醒内存\*/

ldr r0,=0xff000ff

/\* r0 <- current base addr of code \*/

bic r1,pc,r0

/\* r1<- original base addr in ram \*/ /\* 判断代码是否位于RAM\*/

ldr r2,\_TEXT \_PHY\_ BASE

/\* r0<- current base addr of code \*/ /\*如果是则跳过拷贝\*/

bicr2,r2,r0

cmpr1,r2 /\*compare r0,r1 \*/ /\*\_start 等于\_TEXT\_BASE 说明是下载到 RAM 中运行 \*/

/\* r0 == r1 then skip flash copy

moveq r3,#0xf

bl load\_ bl2\_irom

movinand:

Cmp r3, #0xf

beq after\_ copy

bl movi\_bl2 \_copy

drer\_copy

#ifdef CONFIG\_ ENABLE\_MMU

enable mmu: /\*开启MMU\*/

/\* enable domain access \*/mcrp15,0,r5,c3,c0,0

@ load domain access register

/\* Set the TTB register \*/mcrp15,0,r1,c2,c0,0

/\* Enable the MMU \*/

mmu\_on:

mrc p15,0,r0,c1,c0,0

orr r0,r0,#1 /\* 如果定义了宏CONFIG ENABLE MMU, 开启MMU\*/

/\* Set CR M to enable MMU\*/

mcr p15,0,r0,c1,c0,0

注：设置存储控制器，使得外接的SDRAM可用，并且关闭WATCHDOG，设置FCLK、HCLK、PCLK的比例（即设置CLKDIVN寄存器）。

/\*设置堆栈 \*/

stack\_setup: /\*开始转入C程序\*/

ldr r0, \_TEXT\_BASE

/\* upper 128 KiB: relocated uboot \*/

sub r0, r0, #CONFIG\_SYS\_MALLOC\_LEN /\* malloc area \*/

sub r0, r0, #CONFIG\_SYS\_GBL\_DATA\_SIZE /\* 跳过全局数据区\*/

#ifdef CONFIG\_USE\_IRQ

sub r0, r0, #(CONFIG\_STACKSIZE\_IRQ+CONFIG\_STACKSIZE\_FIQ)

#endif

sub sp, r0, #12

BSS段的清零

clear\_bss: /\* 清除bss段\*/

ldr r0, \_bss\_start /\*BSS 段开始地址，在 u-boot.lds 中指定\*/

ldr r1, \_bss\_end /\* BSS 段结束地址，在 u-boot.lds 中指定\*/

mov r2, #0x00000000

clbss\_l:str

r2, [r0] /\* 将 bss 段清零\*/

add r0, r0, #4

cmp r0, r1

ble clbss\_l

ldr pc, \_start\_armboot

\_start\_armboot: .word start\_armboot /\* 跳转到第二阶段代码入口 start\_armboot 处\*/

本段代码先设置了BSS段的起始地址与结束地址，然后循环清楚所有的BSS段。至此，所有的cpu初始化工作（stage1阶段）已经全部结束了。后面的代码，将通过ldr pc, \_start\_armboot，进入C代码执行。这个C入口的函数，是在u-boot-1.1.6\lib\_arm\board.c文件中。它标志着后续将全面启动C语言程序，同时它也是整个u-boot的主函数。

1. **第二阶段C语言代码**

uboot的第二阶段就是要初始化剩下的还没被初始化的硬件，主要是SOC外部硬件（譬如inand、网卡芯片）、uboot本身的一些东西（uboot的命令、环境变量等），然后最终初始化完必要的东西后进入uboot的命令行准备接受命令。start\_armboot函数不仅标志着后续将全面启动C语言程序，同时它也是整个u-boot的主函数。

uboot启动后自动运行打印出很多信息，这些信息就是uboot第一和第二阶段不断进行初始化时，打印出来的信息，然后uboot进入了bootdelay然后执行bootcmd对应的启动命令，如果这时候用户不干涉，会执行bootcmd进入自动启动内核的流程了。（uboot的生命周期就结束了）；所以uboot完结于命令行下，读取命令，解析命令，执行命令。命令行死循环是uboot的最终归宿。

**U-Boot 使用一个数组 init\_sequence 来存储对于大多数开发板都要执行的**

**初始化函数的函数指针。**

typedef int (init\_fnc\_t) (void);

init\_fnc\_t \*init\_sequence[] = {

board\_init, /\*开发板相关的配置 \*/

timer\_init, /\* 时钟初始化-- cpu/arm920t/s3c24x0/timer.c \*/

env\_init, /\*初始化环境变量\*/

init\_baudrate, /\*初始化波特率 \*/

serial\_init, /\* 串口初始化 \*/

console\_init\_f, /\* 控制通讯台初始化阶段 \*/

display\_banner, /\*打印 U-Boot 版本、编译的时间- gedit lib\_arm/board.c \*/ dram\_init, /\*配置可用的 RAM \*/

display\_dram\_config, /\* 显示 RAM 大小-- lib\_arm/board.c \*/

NULL,

};

**start\_armboot 函数**

void start\_armboot (void)

{

init\_fnc\_t \*\*init\_fnc\_ptr;

char \*s;

/\* 计算全局数据结构的地址 gd \*/

gd = (gd\_t\*)(\_armboot\_start - CONFIG\_SYS\_MALLOC\_LEN - sizeof(gd\_t));

„ „

memset ((void\*)gd, 0, sizeof (gd\_t));

gd->bd = (bd\_t\*)((char\*)gd - sizeof(bd\_t));

memset (gd->bd, 0, sizeof (bd\_t));

gd->flags |= GD\_FLG\_RELOC;

monitor\_flash\_len = \_bss\_start - \_armboot\_start;

/\* 逐个调用 init\_sequence 数组中的初始化函数 \*/

for (init\_fnc\_ptr = init\_sequence; \*init\_fnc\_ptr; ++init\_fnc\_ptr) {

if ((\*init\_fnc\_ptr)() != 0) {

hang ();

}

}

/\* armboot\_start 在 cpu/arm920t/start.S 中被初始化为 u-boot.lds 连接脚本中的\_start \*/

mem\_malloc\_init (\_armboot\_start - CONFIG\_SYS\_MALLOC\_LEN,

CONFIG\_SYS\_MALLOC\_LEN); /\* NOR Flash 初始化 \*/

#ifndef CONFIG\_SYS\_NO\_FLASH

display\_flash\_config (flash\_init ());

#endif /\* NAND Flash 初始化\*/

#if defined(CONFIG\_CMD\_NAND)

puts ("NAND: ");

nand\_init();

#endif /\*配置环境变量，重新定位 \*/

env\_relocate (); /\* 从环境变量中获取 IP 地址 \*/

gd->bd->bi\_ip\_addr = getenv\_IPaddr ("ipaddr");

stdio\_init ();

/\* get the devices list going. \*/

jumptable\_init ();

„ „

console\_init\_r (); /\* 作为设备完全初始化控制台 \*/

„ „ /\* 启用异常 \*/

enable\_interrupts ();

#ifdef CONFIG\_USB\_DEVICE

usb\_init\_slave();

#endif /\* 环境初始化\*/

if ((s = getenv ("loadaddr")) != NULL) {

load\_addr = simple\_strtoul (s, NULL, 16);}

#if defined(CONFIG\_CMD\_NET)

if ((s = getenv ("bootfile")) != NULL) {

copy\_filename (BootFile, s, sizeof (BootFile));

}

#endif /\* 网卡初始化 \*/

#if defined(CONFIG\_CMD\_NET)

#if defined(CONFIG\_NET\_MULTI)

puts ("Net: ");

#endif

eth\_initialize(gd->bd);

„ „

#endif

/\* main\_loop() 可以返回重试自动启动，如果是,重新运行. \*/

for (;;) {

main\_loop ();

}

**gd\_t 结构体**

U-Boot 使用了一个结构体 gd\_t 来存储全局数据区的数据，这个结构体在

include/asm-arm/global\_data.h 中定义如下：

typedef

struct global\_data {

bd\_t

\*bd;

unsigned long flags;

unsigned long baudrate;

unsigned long have\_console;

unsigned long env\_addr; /\* 结构地址 \*/

unsigned long env\_valid;

unsigned long fb\_base;

void

} gd\_t;

U-Boot 使用了一个存储在寄存器中的指针 gd 来记录全局数据区的地址：

#define DECLARE\_GLOBAL\_DATA\_PTR register volatile gd\_t \*gd asm

("r8")

DECLARE\_GLOBAL\_DATA\_PTR 定义一个 gd\_t 全局数据结构的指针，这个指针存

放在指定的寄存器 r8 中。这个声明也避免编译器把 r8 分配给其它的变量。任何 想要访问全局数据区的代码，只要代码开头加入“DECLARE\_GLOBAL\_DATA\_PTR”

一行代码，然后就可以使用 gd 指针来访问全局数据区了。

根据 U-Boot 内存使用图中可以计算 gd 的值：

gd = TEXT\_BASE －CONFIG\_SYS\_MALLOC\_LEN － sizeof(gd\_t)

**bd\_t 结构体**

bd\_t 在 include/asm-arm.u/u-boot.h 中定义如下：

typedef struct bd\_info {

int

bi\_baudrate; /\* 串口通讯波特率 \*/

unsigned long bi\_ip\_addr; /\* IP 地址\*/

struct environment\_s \*bi\_env; /\* 环境变量开始地址 \*/

ulong bi\_arch\_number; /\* 开发板的机器码 \*/

ulong bi\_boot\_params; /\* 内核参数的开始地址 \*/

struct /\* RAM 配置信息 \*/

{

ulong start;

ulong size;

}bi\_dram[CONFIG\_NR\_DRAM\_BANKS];

} bd\_t;

U-Boot 启动内核时要给内核传递参数，这时就要使用 gd\_t，bd\_t 结构体中

的信息来设置标记列表

**加载linux内核**

int run\_command (const char \*cmd, int flag)

{

cmd\_tbl\_t \*cmdtp;

... /\* 在命令表中查找命令 \*/

if ((cmdtp = find\_cmd(argv[0])) == NULL) {

... /\* 调用函数执行命令\*/

if ((cmdtp->cmd) (cmdtp, flag, argc, argv) != 0) {

...

}

void main\_loop(void)

{

...

#if defined(CONFIG\_BOOTDELAY) && (CONFIG\_BOOTDELAY >= 0)

s = getenv ("bootdelay");

bootdelay = s ? (int)simple\_strtol(s, NULL, 10) : CONFIG\_BOOTDELAY;

debug ("### main\_loop entered: bootdelay=%d\n\n", bootdelay);

s = getenv ("bootcmd");

# ifndef CFG\_HUSH\_PARSER

run\_command (s, 0);

...

}

U\_BOOT\_CMD(

bootm, CFG\_MAXARGS, 1, do\_bootm,

"bootm - boot application image from memory\n",

"[addr [arg ...]]\n - boot application image stored in memory\n"

"\tpassing arguments 'arg ...'; when booting a Linux kernel,\n"

"\t'arg' can be the address of an initrd image\n"

);

int do\_bootm (cmd\_tbl\_t \*cmdtp, int flag, int argc, char \*argv[])

{

......

image\_header\_t \*hdr = &header; //uimage 是内核加了一个头部

if (argc < 2) {

addr = load\_addr; //如果bootm的参数小于2 则使用默认的链接地址

} else {

addr = simple\_strtoul(argv[1], NULL, 16); }

......

data = addr + sizeof(image\_header\_t);

......

switch (hdr->ih\_comp) {

case IH\_COMP\_NONE:

if(ntohl(hdr->ih\_load) == addr) {

printf (" XIP %s ... ", name);

} else {

memmove ((void \*) ntohl(hdr->ih\_load), (uchar \*)data, len);

}

......

after\_header\_check:

do\_bootm\_linux (cmdtp, flag, argc, argv,addr, len\_ptr, verify);

}

void do\_bootm\_linux (cmd\_tbl\_t \*cmdtp, int flag, int argc, char \*argv[],

ulong addr, ulong \*len\_ptr, int verify)

{ /\* 启动内核的函数指针 \*/

void (\*theKernel)(int zero, int arch, uint params);

image\_header\_t \*hdr = &header; bd\_t \*bd = gd->bd;

/\* 获得命令行参数 \*/

#ifdef CONFIG\_CMDLINE\_TAG

char \*commandline = getenv ("bootargs");

#endif

theKernel = (void (\*)(int, int, uint))ntohl(hdr->ih\_ep)

setup\_start\_tag (bd); // 标记必须以它起始

setup\_commandline\_tag (bd, commandline);

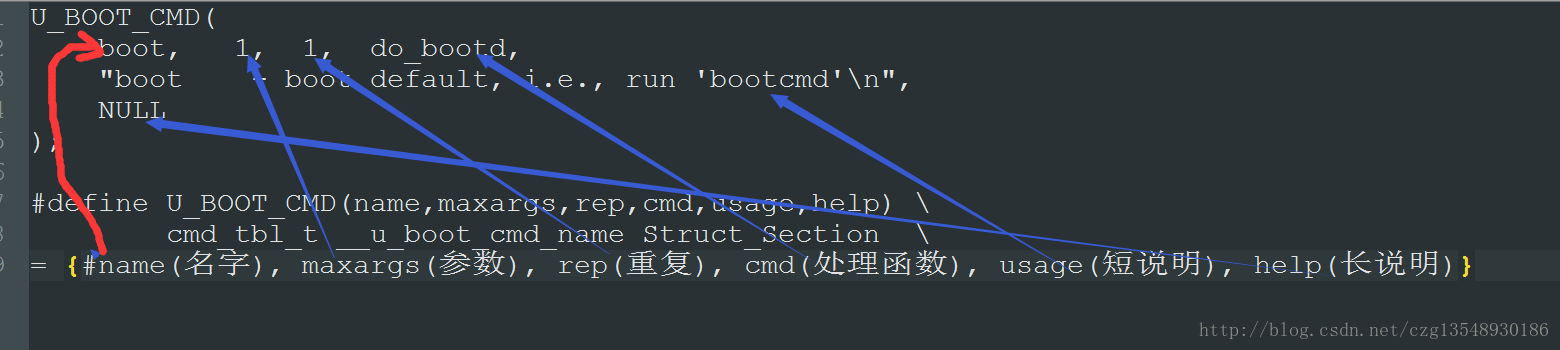
setup\_end\_tag (bd); /\* 标记必须以它结尾 \*/

printf ("\nStarting kernel ...\n\n");

cleanup\_before\_linux ();

/\* 启动内核 第1个参数 0 ，第二个参数 机器ID 第三个参数 tag 地址 \*/

theKernel (0, bd->bi\_arch\_number, bd->bi\_boot\_params);



void main\_loop(void)

{

...

#if defined(CONFIG\_BOOTDELAY) && (CONFIG\_BOOTDELAY >= 0)

s = getenv ("bootdelay");

bootdelay = s ? (int)simple\_strtol(s, NULL, 10) : CONFIG\_BOOTDELAY;

debug ("### main\_loop entered: bootdelay=%d\n\n", bootdelay);

s = getenv ("bootcmd");

# ifndef CFG\_HUSH\_PARSER

run\_command (s, 0);

...

}

run\_main\_loop是board\_r中函数运行列表init\_fnc\_t init\_sequence\_r[]最后一个函数，它又调用了main\_loop，且run\_main\_loop永不返回。

### 三、常用命令

1.nand命令

功能：nand读写，从Nand的 off 偏移地址处读取size 字节的数据到SDRAM的 addr地址

语法：nand cmd param

uboot下NAND操作指令

       指令              Flash内地址         擦除长度

 nand erase     0x100000            0x200000   
       指令              内存中地址         Flash内地址 写入长度

 nand write     0x20000000        0x100000     0x200000

nand read       0x20000000         0x100000     0x200000

1. tftp命令
2. 功能：ftp传输。tftp命令用在本机和tftp服务器之间使用TFTP协议传输文件。TFTP是用来下载远程文件的最简单网络协议，它其于UDP协议而实现。嵌入式linux的tftp开发环境包括两个方面：一是linux服务器端的tftp-server支持，二是嵌入式目标系统的tftp-client支持。

语法：tftp addr file

tftp(选项)(参数)

3.md, mw命令

功能：读写内存

语法：md addr count

使用实例：

修改:mw [内存地址] [值] [长度]  
例如:mw 0x02000000 0 128

表示修改地址为0x02000000~0x02000000+128的内存值为0.

显示:md [内存地址] [长度]  
例如:md 0x02000000 128

表示显示0x02000000的内存数据,长度为128个32bit.

### 总结与感悟

首先感谢老师的线上指导，这是一门实践性较强的课，而且由于本人忙于准备考研相关事宜，因此报告中借鉴了网络上的许多知识，而且内容也有许多不足和漏洞，日后若进行嵌入式方面的研究或者使用Linux系统部署项目，一定在本课程所学的基础上进一步深入研究。