S3C2410 考古笔记

目录

目标

手头上有一个已经淘汰多年的 S3C2410 的开发板，附带只有一张光盘但是光盘里面内容少的可怜：原理图、技术手册、简单的测试工具和预编译的镜像以及一个仅仅三十多页的用户手册。但是这样的板子却具备了一个探索嵌入式系统必备的所有条件：

- 原理图可以了解芯片的连接和外设的配置-虽然原理图和实物并不完全匹配甚至错误之处..
- S3C2410 作为一个 2003 年公布的嵌入式应用处理器，其核心的系统架构和现代的 ARM Cortex-A 系列处理器并无太大差异，反而具有更加简洁的结构和外部接口（Nor/Nand Flash vs eMMC, SDRAM vs DDR4/5），类似于上学时学计算机接口，实验室还是用 Intel 8086 单板机作为实验对象一样，麻雀虽小五脏俱全，很多基本的思想几乎一致.
- 尽可能的抛弃掉原厂提供的开发环境，比如 ADS，调试和烧写工具等，毕竟这些工具也有将近二十年的历史，未必能在现在更新的系统中得到有效的支持，绕开了原有的工具也能更好的了解和掌握芯片的内部结构和工作

但是缺点是作为一个将近二十年的老嵌入式应用处理器，他的开发资料并不算丰富，幸运的是和他极为相似的 S3C2440 却要流行的多，有着众多的开发板和网络开发资源可以参考。

整体目标：

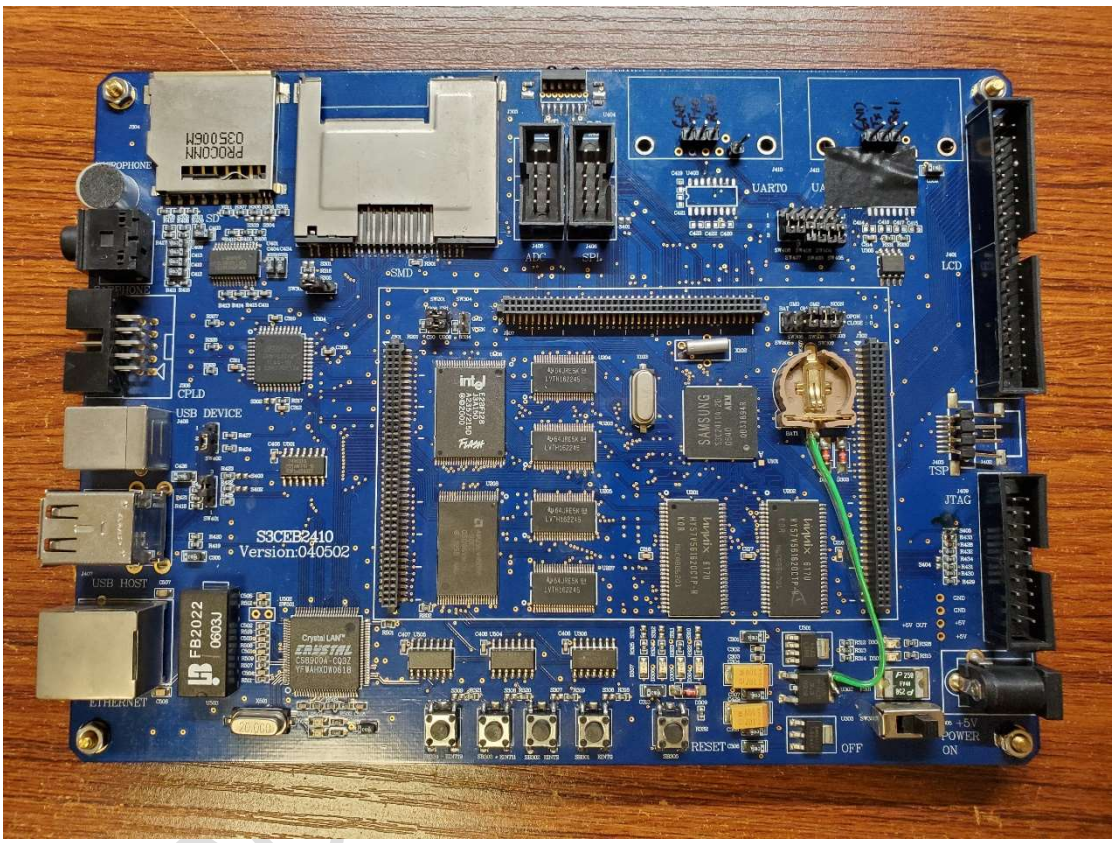
1. 按照个人习使用惯修改板子，使用更为方便的供电方式（可用 USB 或者串口供电），抛弃 232 串口改用 UART TTL 接口
2. 验证芯片能够正常工作，使用 JLink 调试取代二十年前的 ARM-ICE
3. 添加一些 Baremetal 的代码用来熟悉芯片的编译和调试，并了解芯片的外设、时钟与存储器，为后面的 u-boot/kernel 打基础
4. JLink 烧写板载 Nor Flash 取代各种客制化的 Windows 版本 flash 烧写工具
5. 编译运行 U-Boot
6. 编译运行 Kernel
7. 制作根文件系统
8. 研究 U-Boot 编译、启动和加载

9. 研究 Linux kernel 的编译、启动

Barematel

硬件

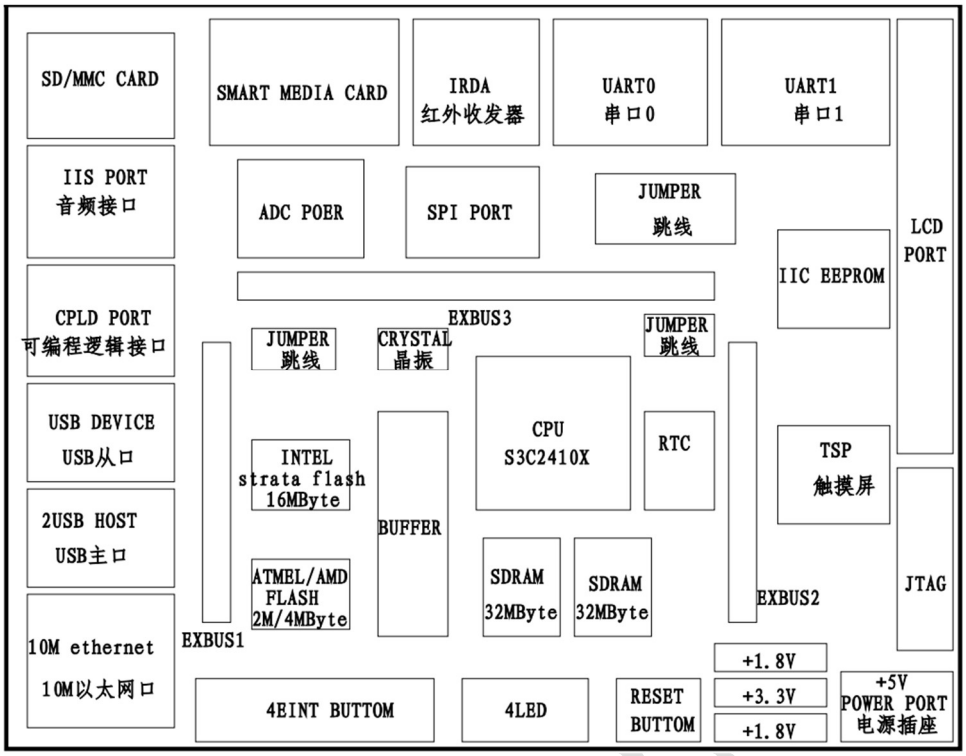
外观：



需要修改的地方：

1. 因为缺少 RTC 电池，故将 3.3V 电源短接至 VBAT
2. 串口接口为 RS232-DB9 接口，改成 TTL 电平的接口，去掉 MAX3232 和没用的跳线帽
3. 原配的 5V 电源已经损坏，改成利用板子上方口 USB 供电或者 TTL 串口模块供电
4. JTAG 调试时可以通过调试器对系统进行硬件复位

板子结构：



开发板的配置：

1. S3C2410A 处理器，外部一个 12MHz 晶振和一个 32.768KHz 表晶振
2. 两片 HY57V561620CTP-H SDRAM 组成 64MB 内存
3. Nor Flash 为 AM29LV160DB，大小 2MB，可以选择 nGCS0/1 启动
4. 另配一片 Intel Strata Flash E28F128J3A，大小 16MB，MLC 结构，可以选择 nGCS1/0 启动（不打算用..）
5. 没有板载 NAND FLASH，可以通过 Smart Media Card 外接

工具链

开发环境

```
→  linux-3.4.2 uname -a
Linux thelaptop 5.14.0-1052-oem #59-Ubuntu SMP Fri Sep 9 09:37:59 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
→  linux-3.4.2 lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.5 LTS
Release:        20.04
Codename:       focal
```

交叉编译器安装

网上多数资料都用的是 arm-linux-gcc v4.4.3, 可以在 FrendlyARM 官网下载:
http://112.124.9.243/arm9net/mini2440/linux/arm-linux-gcc-4.4.3-20100728.tar.gz

解压后拷贝 4.4.3/目录到/usr 下并重命名为 arm-linux-toolchains4.4.3, 并在添加到 PATH 中:

```
export PATH=$PATH:/usr/arm-linux-toolchains4.4.3/bin
```

>> source ~/.zshrc

>> arm-linux-gcc –version

```
➜  linux-3.4.2 arm-linux-gcc --version
.arm-none-linux-gnueabi-gcc (ctng-1.6.1) 4.4.3
Copyright (C) 2010 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

其他基本开发环境

1. sudo apt install build-essential
2. sudo apt install python3 python3-pip python-is-python3
3. sudo apt install libc6-i386 libstdc++6 lib32stdc++6 lib32z1 libncurses-dev

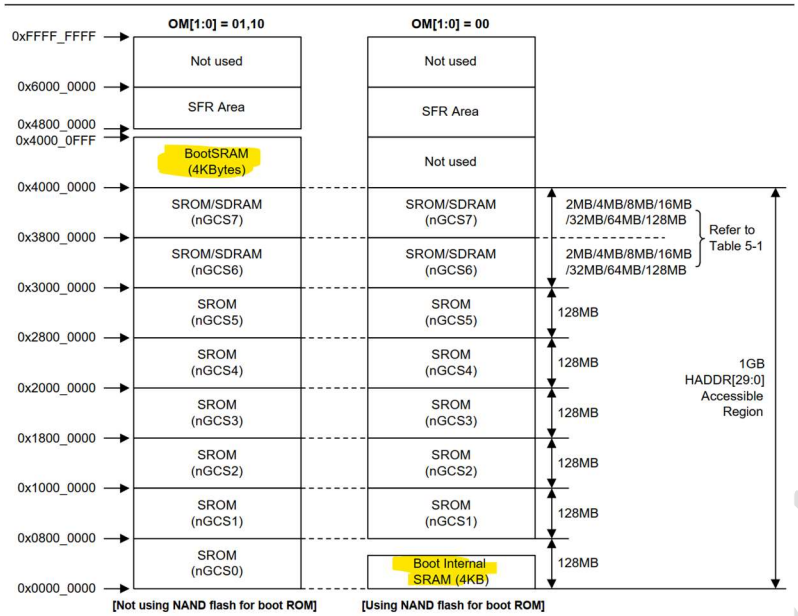
调试

SoC 中典型的存放代码的介质是 ROM，OTP，FLASH 和 RAM（包括 SRAM，SDRAM 等）， ROM 不可更改，一般为 SoC 片内第一启动介质负责加载二级引导代码，OTP 类似但只能烧写一次（按 bit 翻转计算），FLASH 可以多次烧写且为非易失存储器，RAM 则需要引导代码从非易失存储器中加载。

针对 S3C2410，由于没有片内 FLASH，所以可行的调试方案:

1. 编译好测试代码后转为为.bin 文件,通过烧录器或者 JTAG 烧写到 NorFlash 上
2. 通过 JTAG 先配置时钟和 SDRAM, 配置完成后通过 load 方法加载.elf/.bin 到 SDRAM 中

   S3C2410 中有一个 4KB 大小的启动 SRAM 用来帮助引导程序在初始化 SDRAM 之前使用，其地址根据 OM[1:0]配置为 0x4000_0000 或者 0x0



这部分空间可以直接被 CPU 使用不需要像 NorFlash 依赖烧录工具或者 SDRAM 需要 初始化。所以可以通过 JTAG 直接 load 代码到这部分空间进行简单的测试.对于测试 GPIO, PLL, SDRAM 等简单的功能,这部分空间已经足够使用了.

安装 Segger JLink 工具和 JFlash

Win: https://www.segger.com/downloads/jlink/JLink_Windows_V640.exe

Linux: https://www.segger.com/downloads/jlink/JLink_Linux_V640_i386.deb

Linux 下添加 PATH:

```
export PATH=$PATH:/opt/SEGGER/JLink_V640/
```

查看 GDB

```
→  linux-3.4.2 gdb -v
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

JTAG 调试

启动 GDBServer:

JLinkGDBServerCLExe -select USB -device ARM9 -endian little -if JTAG -speed 4000 -noir -LocalhostOnly

```
→  linux-3.4.2 JLinkGDBServerCLExe -select USB -device ARM9 -endian little -if JTAG -speed 4000 -noir -Loca
lhostOnly

SEGGER J-Link GDB Server V6.40 Command Line Version

JLinkARM.dll V6.40 (DLL compiled Oct 26 2018 15:08:28)

Command line: -select USB -device ARM9 -endian little -if JTAG -speed 4000 -noir -LocalhostOnly
-----GDB Server start settings-----
GDBInit file:                  none
GDB Server Listening port:     2331
SWO raw output listening port: 2332
Terminal I/O port:             2333
Accept remote connection:      localhost only
Generate logfile:              off
Verify download:               off
Init regs on start:            off
Silent mode:                   off
Single run mode:               off
Target connection timeout:     0 ms
------J-Link related settings------
J-Link Host interface:         USB
J-Link script:                 none
J-Link settings file:          none
------Target related settings------
Target device:                 ARM9
Target interface:              JTAG
Target interface speed:        4000kHz
Target endian:                 little

Connecting to J-Link...
J-Link is connected.
Firmware: J-Link V9 compiled Jun  2 2222 22:22:22
Hardware: V9.40
S/N: 59401308
Feature(s): GDB, RDI, FlashBP, FlashDL, JFlash, RDDI
Checking target voltage...
Target voltage: 3.29 V
Listening on TCP/IP port 2331
Connecting to target...
J-Link found 1 JTAG device, Total IRLen = 4
JTAG ID: 0x0032409D (ARM9)
Connected to target
Waiting for GDB connection...
```

此时 JTAG 已经扫描到 S3C2410

GDB 连接:

启动 gdb 并连接 server: target remote localhost:2331

```
→  ~ gdb
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) target remote localhost:2331
Remote debugging using localhost:2331
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0xc056ade8 in ?? ()
(gdb)
```

Server Log:

```
Reading all registers
Read 1 bytes @ address 0xC056ADE8 (Data = 0x00)
Read 1 bytes @ address 0xC056ADE8 (Data = 0x00)
Reading 64 bytes @ address 0xC056ADC0
Reading 8 bytes @ address 0xC056ADE8
Reading 7 bytes @ address 0xC056ADE8
```

Reset CPU:

```
(gdb) monitor reset 0
Resetting target (halt after reset)
```

Read CPU Register:

```
(gdb) monitor regs
PC = 00000000, CPSR = 000000D3 (SVC mode, ARM FIQ dis. IRQ dis.)
R0 = 00000000, R1 = 00000000, R2 = 00000000, R3 = 00000000
R4 = 00000000, R5 = 00000000, R6 = 00000000, R7 = 00000000
USR: R8 =00000000, R9 =00000000, R10=00000000, R11 =00000000, R12 =00000000
     R13=00000000, R14=00000000
FIQ: R8 =00000000, R9 =00000000, R10=00000000, R11 =00000000, R12 =00000000
     R13=00000000, R14=00000000, SPSR=00000010
SVC: R13=00000000, R14=00000000, SPSR=00000010
ABT: R13=00000000, R14=00000000, SPSR=00000010
IRQ: R13=00000000, R14=00000000, SPSR=00000010
UND: R13=00000000, R14=00000000, SPSR=00000010
(gdb)
```

Load firmware:

找到编译后的.elf, load xxx.elf 或者 load xxx.elf 0x40000000 加载到指定地址

设置 pc 寄存器:

```
(gdb) monitor reg pc=0x40000000
Writing register (PC = 0x40000000)
```

Makefile & Linker

创建一个简单的 Makefile 用来自动编译 baremetal 的代码,并使用 Linker scripts 指示链接器链接目标和对应地址

编译规则:

```
#Default target
.DEFAULT_GOAL := all

#Define App name
CC = arm-linux-gcc
LD = arm-linux-ld

#Compiler flags
CFLAGS := -Wall

#Global defines
DEFINES:=

#Compile Optimization
#OPTIMIZATIONS := -g -O2
OPTIMIZATIONS := -g

#Source folder and target folder
srctree:=$(shell pwd)
build_folder:=$(srctree)/build

HEADERS:=

#----------- Build Rules -----------
$(build_folder)/%.o::$(srctree)/%.c
    @echo '-------> GCC -c $<  >>> $@'
    $(CC) $(CFLAGS) $(OPTIMIZATIONS) $(DEFINES)
$(HEADERS) -MMD -MP -MF"$(@:%.o=%.dep)" -MT"$(@)" -c -o
$@ $<

$(build_folder)/%.o::$(srctree)/%.s
```

```
    @echo '-------> GCC -c $< >>> $@'
    $(CC) -c $< $(CFLAGS) -o $@
```

核心的规则就两个:

对于任何的在 srctree 下的*.c 转换为 build_folder 下的*.o

```
$(build_folder)/%.o::$(srctree)/%.c
    @echo '-------> GCC -c $<  >>> $@'
    $(CC) $(CFLAGS) $(OPTIMIZATIONS) $(DEFINES)
$(HEADERS) -MMD -MP -MF"$(@:%.o=%.dep)" -MT"$(@)" -c -o
$@ $<
```

对于任何在 srctree 下的*.s 转换为 build_folder 下的*.o:

```
$(build_folder)/%.o::$(srctree)/%.s
    @echo '-------> GCC -c $< >>> $@'
    $(CC) -c $< $(CFLAGS) -o $@
```

链接脚本:

```
MEMORY
{
    ROM  (rx)  : ORIGIN = 0x0, LENGTH = 0x30000000
    RAM  (rwx) : ORIGIN = 0x30000000, LENGTH = 0x8000000
}

ENTRY(_start)


SECTIONS
{
    .text : ALIGN(8)
    {
        KEEP(*(.vectors))
        __rst_hdl_start = .;
        KEEP(*(rst_hdl))
        *(.text)
        *(.text*)
        KEEP(*(.init))
```

```
        KEEP(*(.fini))

        *crtbegin.o(.ctors)
        *crtbegin?.o(.ctors)
        *(EXCLUDE_FILE(*crtend?.o *crtend.o) .ctors)
        *(SORT(.ctors.*))
        *(.ctors)

        *.crtbegin.o(.dtors)
        *.crtbegin?.o(.dtors)
        *(EXCLUDE_FILE(*crtend?.o *crtend.o) .dtors)
        *(SORT(.dtors.*))
        *(.dtors)

        *(.rodata*)

        KEEP(*(.eh_frame*))

    } > ROM

    .ARM.extab :
    {
        *(.ARM.extab* .gnu.linkonce.armextab.*)
    } > ROM

    .ARM.exidx :
    {
        *(.ARM.exidx* .gnu.linkonce.armexidx.*)
    } > ROM

    . = ALIGN(4);
    __end_text = .;

    .data : AT (__end_text)
    {
        . = ALIGN(4);
        __data_start__ = .;
        *(vtable)
        *(.data*)

        . = ALIGN(4);
```

```
        PROVIDE_HIDDEN(__fini_array_start = .);
        KEEP(*(SORT(.fini_array.*)))
        KEEP(*(.fini_array))
        PROVIDE_HIDDEN(__fini_array_end = .);

        KEEP(*(.jcr*))
        . = ALIGN(4);
        __data_end__ = .;
    } > RAM

    .bss :
    {
        . = ALIGN(4);
        __bss_start__ = .;
        *(.bss*)
        *(COMMON)
        . = ALIGN(4);
        __bss_end__ = .;
    } > RAM

    .stack_dummy (COPY):
    {
        . = ALIGN(8);
        __StackLimit = .;
        KEEP(*(.stack*))
        . = +0x1000;
        . = ALIGN(8);
        __StackTop = .;

    } > RAM

    PROVIDE(__stack = __StackTop);
}
```

GPIO 的 makefile: 添加目标文件列表到 srctree,并生成目录树和依赖文件,编译后并用 arm-linux-ld 和链接脚本链接,并生成.map 文件和反汇编文件

```
include ../general/makefile.inc

#----------- OUTPUT TARGET -----------
TARGET = led_blink
```

```
#----------- Sources List -----------
OBJ_FILES_USR := $(srctree)/start.o
OBJ_FILES_USR += $(srctree)/led_test.o


HEADERS += -I../general/include/
#----------- Dependency -----------
DEPS_ns:=$(OBJ_FILES_USR:$(srctree)/%.o=$(build_folder)/%
.dep)
-include $(DEPS_ns)

OBJ_FILES_USR_BLD:=$(patsubst $(srctree)/%.o,
$(build_folder)/%.o, $(OBJ_FILES_USR))

OBJ_FILES += $(OBJ_FILES_USR_BLD)

# Create empty folder
MAKE_CREATE_FOLDER:= $(sort $(dir $(OBJ_FILES_USR_BLD)))

# Make each object file dependent on the system
configuration application parameters.
$(OBJ_FILES): $(OBJ_FILES_USR_BLD)

$(OBJ_FILES_USR_BLD): $(MAKE_CREATE_FOLDER)

$(MAKE_CREATE_FOLDER):
    @mkdir -p $@

#------------makefile targets -----------------
build_src: $(OBJ_FILES)

link_obj: build_src
    @echo "------- Linking -----"
    $(LD) -T ../general/etc/s3c2410_link.ld $(OBJ_FILES)
-o $(build_folder)/$(TARGET).elf -Map
$(build_folder)/$(TARGET).map

post_build:
    @echo "------- Post Build -----"
```

```
    arm-linux-objcopy -O binary -S
$(build_folder)/$(TARGET).elf
$(build_folder)/$(TARGET).bin
    arm-linux-objdump -xD $(build_folder)/$(TARGET).elf >
$(build_folder)/$(TARGET).dis

all: build_src link_obj post_build


clean:
    rm -rf ./build/

.PHONY: all clean
```

GPIO

准备 Makefile

准备 start.s 文件, 关闭看门狗， 初始化堆栈指针为 4KB BOOTRAM 末尾-8B

```
.text
.global _start

_start:
    @Stop Watchdog
    LDR R0,=0x53000000
    MOV R1,#0
    STR R1,[R0]

    @Set SP
    LDR SP,=0x40000ff8

    @Jump to main
    BL main

halt:
    B halt
```

根据手册, 配置 GPIO 输入输出控制寄存器:

Register	Address	R/W	Description	Reset Value
GPFCON	0x56000050	R/W	Configure the pins of port F	0x0
GPFDAT	0x56000054	R/W	The data register for port F	Undefined
GPFUP	0x56000058	R/W	Pull-up disable register for port F	0x0
Reserved	0x5600005C	–	Reserved	Undefined

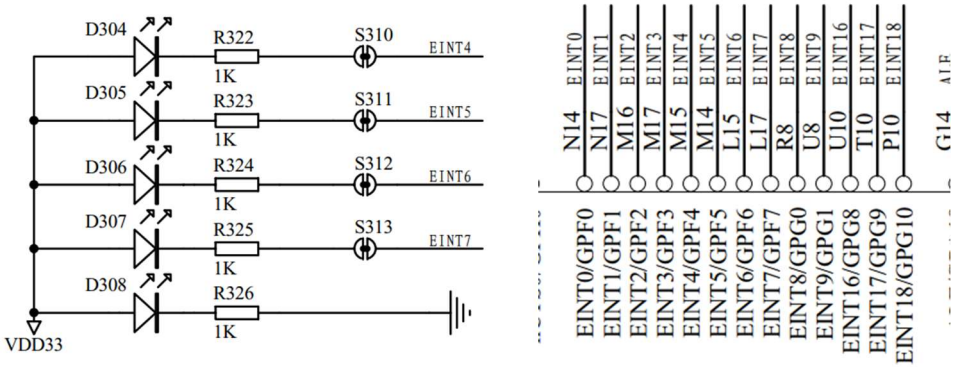
GPFCON	Bit	Description
GPF7	[15:14]	00 = Input 01 = Output 10 = EINT7 11 = Reserved
GPF6	[13:12]	00 = Input 01 = Output 10 = EINT6 11 = Reserved
GPF5	[11:10]	00 = Input 01 = Output 10 = EINT5 11 = Reserved
GPF4	[9:8]	00 = Input 01 = Output 10 = EINT4 11 = Reserved
GPF3	[7:6]	00 = Input 01 = Output 10 = EINT3 11 = Reserved
GPF2	[5:4]	00 = Input 01 = Output 10 = EINT2 11 = Reserved
GPF1	[3:2]	00 = Input 01 = Output 10 = EINT1 11 = Reserved
GPF0	[1:0]	00 = Input 01 = Output 10 = EINT0 11 = Reserved

GPFDAT 配置输出电平:

1.

GPFDAT	Bit	Description
GPF[7:0]	[7:0]	When the port is configured as input port, data from external sources can be read to the corresponding pin. When the port is configured as output port, data written in this register can be sent to the corresponding pin. When the port is configured as functional pin, undefined value will be read.

原理图:



测试代码:

```
#include "s3c2410.h"
#include "stdint.h"
```

```c
typedef struct
{
    uint32_t gpf0 : 2;
    uint32_t gpf1 : 2;
    uint32_t gpf2 : 2;
    uint32_t gpf3 : 2;
    uint32_t gpf4 : 2;
    uint32_t gpf5 : 2;
    uint32_t gpf6 : 2;
    uint32_t gpf7 : 2;
    uint32_t reserved : 16;
}gpfcon_t;

int main(void)
{
    volatile int i = 0;
    volatile gpfcon_t *reg = (gpfcon_t *)&rGPFCON;

    reg->gpf4 = 1;
    reg->gpf5 = 1;
    reg->gpf6 = 1;
    reg->gpf7 = 1;

    while(1)
    {
        rGPFDAT = 0x5 << 4;
        for(i = 0; i < 10000; i++);
        rGPFDAT = 0xA << 4;
        for(i = 0; i < 10000; i++);
    }
}
```

Clock

配置开发板 OM[3:2]跳线为 00 用来选择系统时钟源为外部晶振:

Table 7-1. Clock Source Selection at Boot-Up

Mode OM[3:2]	MPLL State	UPLL State	Main Clock source	USB Clock Source
00	On	On	Crystal	Crystal
01	On	On	Crystal	EXTCLK
10	On	On	EXTCLK	Crystal
11	On	On	EXTCLK	EXTCLK

系统时钟树可大致分为四个线路:

XTAL-----PLL------FCLK （CPU 主时钟)

XTAL-----PLL------HCLK （外部总线时钟，SDRAM，FLASH，DMA，LCD）

XTAL-----PLL------PCLK （片内外设时钟，I2C，UART ..）

XTAL-----USBPLL--USB Host/Device

时钟结构:

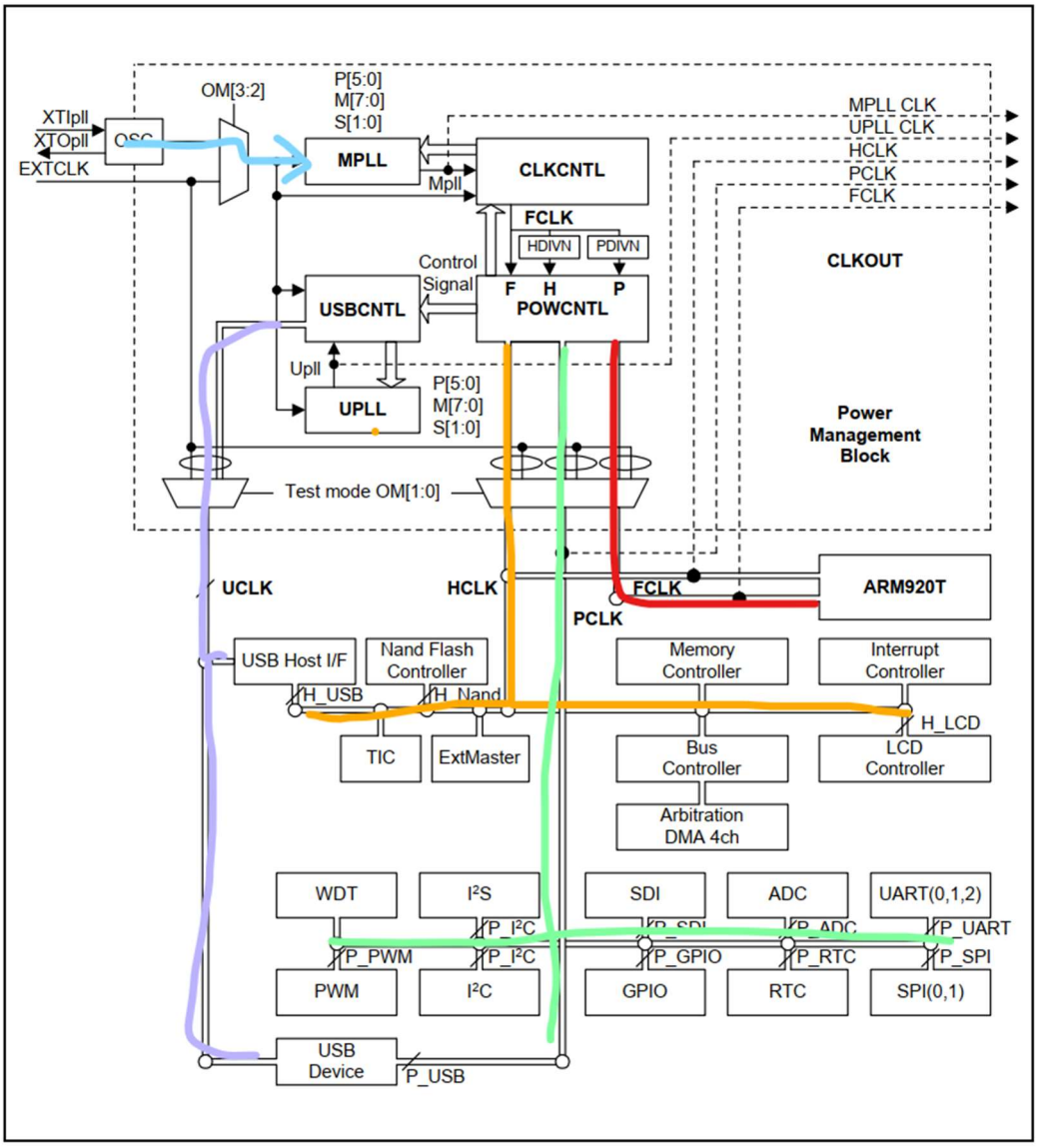


Figure 7-1. Clock Generator Block Diagram

手册中的极限频率:

Table 24-5. S3C2410X Power Supply Voltage and Current

Parameter	Value	Unit	Condition
Typical V_{DDi} / V_{DDIO}	1.8 / 3.3	V	
Max. Operating frequency (FCLK)	203	MHz	
Max. Operating frequency (HCLK)	101.5	MHz	
Max. Operating frequency (PCLK)	50.7	MHz	

HDIVN	PDIVN	FCLK	HCLK	PCLK	Divide Ratio
0	0	FCLK	FCLK	FCLK	1 : 1 : 1 (Default)
0	1	FCLK	FCLK	FCLK / 2	1 : 1 : 2
1	0	FCLK	FCLK / 2	FCLK / 2	1 : 2 : 2
1	1	FCLK	FCLK / 2	FCLK / 4	1 : 2 : 4 (recommended)

PLL 相关寄存器:

Register	Address	R/W	Description	Reset Value
MPLLCON	0x4C000004	R/W	MPLL configuration register	0x0005C080
UPLLCON	0x4C000008	R/W	UPLL configuration register	0x00028080

PLLCON	Bit	Description	Initial State
MDIV	[19:12]	Main divider control	0x5C / 0x28
PDIV	[9:4]	Pre-divider control	0x08 / 0x08
SDIV	[1:0]	Post divider control	0x0 / 0x0

官方建议 PLL 配置:

Input Frequency	Output Frequency	MDIV	PDIV	SDIV
12.00MHz	11.289MHz	N/A	N/A	N/A
12.00MHz	16.934MHz	N/A	N/A	N/A
12.00MHz	22.50MHz	N/A	N/A	N/A
12.00MHz	33.75MHz	82 (0x52)	2	3
12.00MHz	45.00MHz	82 (0x52)	1	3
12.00MHz	50.70MHz	161 (0xa1)	3	3
12.00Mhz	48.00Mhz (note)	120 (0x78)	2	3
12.00MHz	56.25MHz	142 (0x8e)	2	3
12.00MHz	67.50MHz	82 (0x52)	2	2
12.00MHz	79.00MHz	71 (0x47)	1	2
12.00MHz	84.75MHz	105 (0x69)	2	2
12.00MHz	90.00MHz	112 (0x70)	2	2
12.00MHz	101.25MHz	127 (0x7f)	2	2
12.00MHz	113.00MHz	105 (0x69)	1	2
12.00MHz	118.50MHz	150 (0x96)	2	2
12.00MHz	124.00MHz	116 (0x74)	1	2
12.00MHz	135.00MHz	82 (0x52)	2	1
12.00MHz	147.00MHz	90 (0x5a)	2	1
12.00MHz	152.00MHz	68 (0x44)	1	1
12.00MHz	158.00MHz	71 (0x47)	1	1
12.00MHz	170.00MHz	77 (0x4d)	1	1
12.00MHz	180.00MHz	82 (0x52)	1	1
12.00MHz	186.00MHz	85 (0x55)	1	1
12.00MHz	192.00MHz	88 (0x58)	1	1
12.00MHz	202.80MHz	161 (0xa1)	3	1

FCLK=202.8MHz, HCLK=101.4MHz, PCLK=50.7MHz

由于 FCLK ≠ HCLK， 手册要求 CPU 从 fast bus mode 切换为 async bus mode：

2. If HDIVN=1, the CPU bus mode has to be changed from the fast bus mode to the asynchronous bus mode using following instructions.

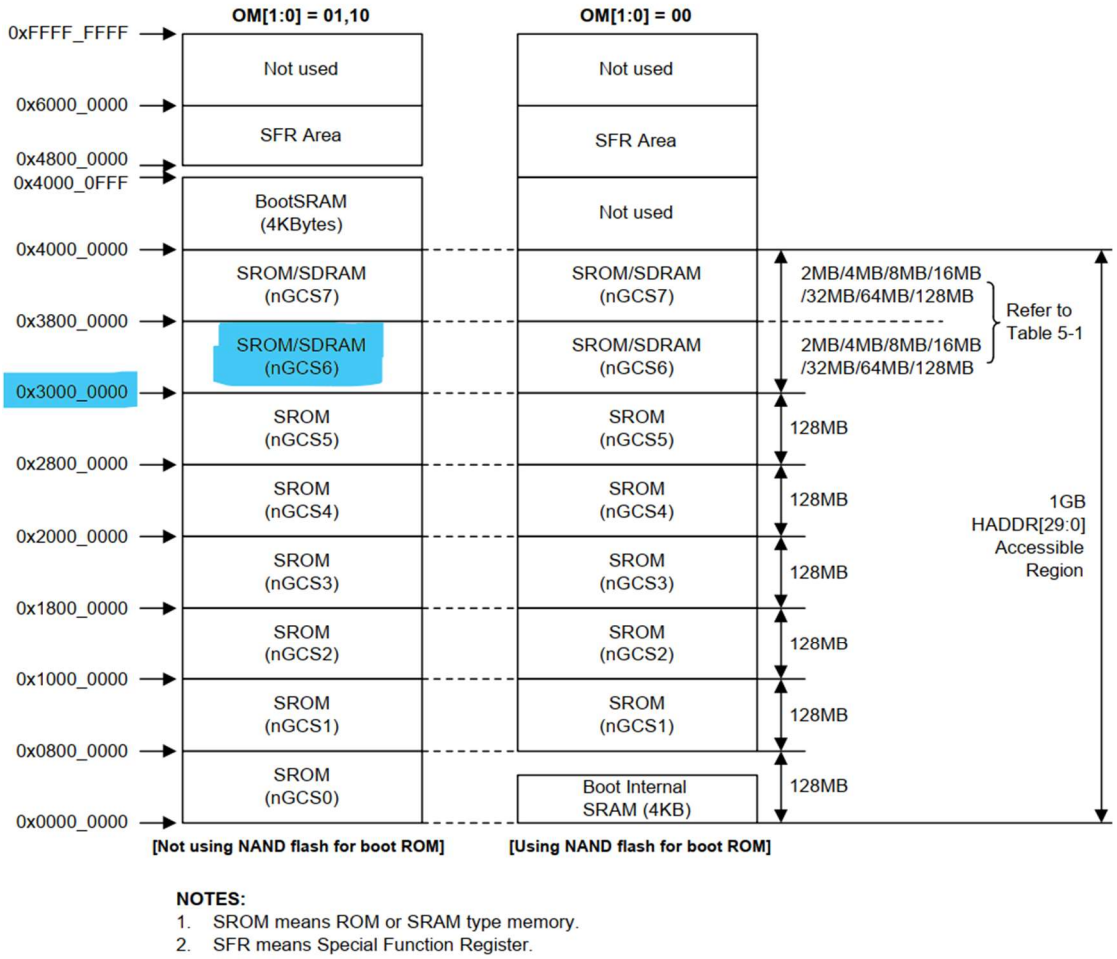
```
MMU_SetAsyncBusMode
    mrc  p15,0,r0,c1,c0,0
    orr  r0,r0,#R1_nF:OR:R1_iA
    mcr  p15,0,r0,c1,c0,0
```

```
_start:
    @Stop Watchdog
    LDR R0,=0x53000000
    MOV R1,#0
    STR R1,[R0]

    @Set SP
    LDR SP,=0x40000ff8

    @Set HCLK and PCLK dividor, HCLK=FCLK/2=101.4MHz,
PCLK=FCLK/4=50.7MHz
    LDR R0,=0x4C000014
    LDR R1,=0x3
    STR R1,[R0]

    @Changing CPU to async mode instead of fast-bus mode
    mrc p15,0,r0,c1,c0,0
    orr r0,r0,#0xc0000000 @#R1_nF:OR:R1_iA
    mcr p15,0,r0,c1,c0,0

    @Setup MPLL, FCLK=202.8MHz when using 12MHz crystal
    LDR R0,=0x4c000004
    LDR R1,=0xA1031
```

```
    STR R1,[R0]

    @Setup UPLL 48MHz
    LDR R0,=0x4c000008
    LDR R1,=0x78023
    STR R1,[R0]
```

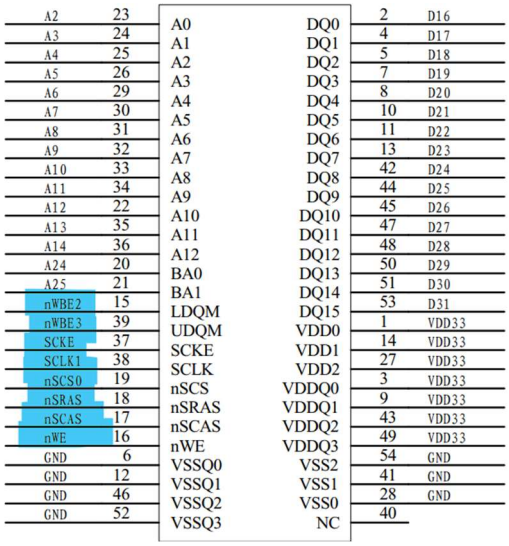
SDRAM

地址映射：



NOTES:
1. SROM means ROM or SRAM type memory.
2. SFR means Special Function Register.

板子上使用了两片 32MB 的 HY57V5616CTP-H SDRAM，总共 64MB，并拼成 32bits 位宽数据接口。控制器信号：



nSCS0 连接到 S3C2410 的 nGCS6, 即 BANK6 片选上

配置 BANK6 位宽，nWBE, no-WAIT 等属性：

BUS WIDTH & WAIT CONTROL REGISTER (BWSCON)

Register	Address	R/W	Description	Reset Value
BWSCON	0x48000000	R/W	Bus width & wait status control register	0x000000

		00 = 8-bit 01 = 16-bit, 10 = 32-bit 11 = reserved	
ST6	[27]	Determine SRAM for using UB/LB for bank 6. 0 = Not using UB/LB (The pins are dedicated nWBE[3:0) 1 = Using UB/LB (The pins are dedicated nBE[3:0])	0
WS6	[26]	Determine WAIT status for bank 6. 0 = WAIT disable, 1 = WAIT enable	0
DW6	[25:24]	Determine data bus width for bank 6. 00 = 8-bit 01 = 16-bit, 10 = 32-bit 11 = reserved	0

配置 SDRAM 时序:

Register	Address	R/W	Description	Reset Value
BANKCON6	0x4800001C	R/W	Bank 6 control register	0x18008

控制器类型:

BANKCONn	Bit	Description	Initial State
MT	[16:15]	Determine the memory type for bank6 and bank7. 00 = ROM or SRAM 01 = Reserved (Do not use) 10 = Reserved (Do not use) 11 = Sync. DRAM	11

MT=11， 只关注[3:0] bits, 根据手册:

Parameter		Symbol	-6		-K		-H		-8		-P		-S		Unit	Note
			Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max		
$\overline{RAS}$ Cycle Time	Operation	tRC	60	-	60	-	65	-	68	-	70	-	70	-	ns	
	Auto Refresh	tRRC	60	-	60	-	65	-	68	-	70	-	70	-	ns	
$\overline{RAS}$ to $\overline{CAS}$ Delay		tRCD	18	-	15	-	20	-	20	-	20	-	20	-	ns	

Trcd=20ns, HCLK=101.4MHz, 约 2cycle

列地址宽度（9bits）：

A0 ~ A12	Address	Row Address : RA0 ~ RA12, Column Address : CA0 ~ CA8 Auto-precharge flag : A10

最终配置：

Memory Type = SDRAM [MT=11] (4-bit)			
Trcd	[3:2]	RAS to CAS delay 00 = 2 clocks 01 = 3 clocks 10 = 4 clocks	10
SCAN	[1:0]	Column address number 00 = 8-bit 01 = 9-bit 10= 10-bit	00

配置刷新寄存器：

REFRESH CONTROL REGISTER

Register	Address	R/W	Description	Reset Value
REFRESH	0x48000024	R/W	SDRAM refresh control register	0xac0000

根据手册：

Parameter		Symbol	-6		-K		-H		-8		-P		-S		Unit	Note
			Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max		
$\overline{RAS}$ Cycle Time	Operation	tRC	60	-	60	-	65	-	68	-	70	-	70	-	ns	
	Auto Refresh	tRRC	60	-	60	-	65	-	68	-	70	-	70	-	ns	
$\overline{RAS}$ to $\overline{CAS}$ Delay		tRCD	18	-	15	-	20	-	20	-	20	-	20	-	ns	
$\overline{RAS}$ Active Time		tRAS	42	100K	45	100K	45	100K	48	100K	50	100K	50	100K	ns	
$\overline{RAS}$ Precharge Time		tRP	18	-	15	-	20	-	20	-	20	-	20	-	ns	
$\overline{RAS}$ to $\overline{RAS}$ Bank Active Delay		tRRD	12	-	15	-	15	-	16	-	20	-	20	-	ns	

Trp=20ns, 约 2cycles, Trc=65ns, Tsrc = Trc-Trp = 45ns 约 5cycles

FEATURES

- Single 3.3±0.3V power supply
- All device pins are compatible with LVTTL interface
- JEDEC standard 400mil 54pin TSOP-II with 0.8mm of pin pitch
- All inputs and outputs referenced to positive edge of system clock
- Data mask function by UDQM, LDQM
- Internal four banks operation
- Auto refresh and self refresh
- 8192 refresh cycles / 64ms
- Programmable Burst Length and Burst Type
  - 1, 2, 4, 8 or Full page for Sequential Burst
  - 1, 2, 4 or 8 for Interleave Burst
- Programmable $\overline{\text{CAS}}$ Latency ; 2, 3 Clocks

64ms 内发生 8192 次刷新，刷新周期为 7.8125us

Refresh Count = 2^11 + 1 – 101.4 * 7.8125 = 1257

REFRESH	Bit	Description	Initial State
REFEN	[23]	SDRAM Refresh Enable 0 = Disable 1 = Enable (self/auto refresh)	1
TREFMD	[22]	SDRAM Refresh Mode 0 = Auto Refresh 1 = Self Refresh In self-refresh time, the SDRAM control signals are driven to the appropriate level.	0
Trp	[21:20]	SDRAM RAS pre-charge Time 00 = 2 clocks 01 = 3 clocks 10 = 4 clocks 11 = Not support	10
Tsrc	[19:18]	SDRAM Semi Row Cycle Time 00 = 4 clocks 01 = 5 clocks 10 = 6 clocks 11 = 7 clocks SDRAM's Row_Cycle time (Trc) = Tsrc + Trp If) Trp = 3 clocks & Tsrc = 7 clocks, Trc = 3 + 7 = 10 clocks	11
Reserved	[17:16]	Not used	00
Reserved	[15:11]	Not used	0000
Refresh Counter	[10:0]	SDRAM refresh count value. Refresh period = (2^{11}-refresh_count+1)/HCLK Ex) If refresh period is 15.6 us and HCLK is 60 MHz, the refresh count is as follows: Refresh count = 2^{11} + 1 - 60x15.6 = 1113	0

BANKSIZE 与控制配置：

BANKSIZE REGISTER

Register	Address	R/W	Description	Reset Value
BANKSIZE	0x48000028	R/W	Flexible bank size register	0x0

BANKSIZE	Bit	Description	Initial State
BURST_EN	[7]	ARM core burst operation enable. 0 = Disable burst operation. 1 = Enable burst operation.	0
Reserved	[6]	Not used	0
SCKE_EN	[5]	SDRAM power down mode enable control by SCKE 0 = SDRAM power down mode disable 1 = SDRAM power down mode enable	0
SCLK_EN	[4]	SCLK is enabled only during SDRAM access cycle for reducing power consumption. When SDRAM is not accessed, SCLK becomes 'L' level. 0 = SCLK is always active. 1 = SCLK is active only during the access (recommended).	0
Reserved	[3]	Not used	0
BK76MAP	[2:0]	BANK6/7 memory map 010 = 128MB/128MB 001 = 64MB/64MB 000 = 32M/32M 111 = 16M/16M 110 = 8M/8M 101 = 4M/4M 100 = 2M/2M	010

SDRAM MODE REGISTER SET REGISTER (MRSR)

Register	Address	R/W	Description	Reset Value
MRSRB6	0x4800002C	R/W	Mode register set register bank6	xxx
MRSRB7	0x48000030	R/W	Mode register set register bank7	xxx

MRSR	Bit	Description	Initial State
Reserved	[11:10]	Not used	–
WBL	[9]	Write burst length 0: Burst (Fixed) 1: Reserved	x
TM	[8:7]	Test mode 00: Mode register set (Fixed) 01, 10 and 11: Reserved	xx
CL	[6:4]	CAS latency 000 = 1 clock, 010 = 2 clocks, 011=3 clocks Others: reserved	xxx
BT	[3]	Burst type 0: Sequential (Fixed) 1: Reserved	x
BL	[2:0]	Burst length 000: 1 (Fixed) Others: Reserved	xxx

代码

```
/* Set Bank6 as 32bit, nWBE, no WAIT */
rBWSCON = 0x2000000;
```

```
    /* Bank6 as SDRAM and set up timing, Trcd=20ns,
column address width=a0-a8(9bits)*/
    rBANKCON6 = 0x18001;

    /* Setup refresh,
trp=20ns(3cycle),tsrc=45ns(5cycle),refresh counter=1257*/
    rREFRESH = 0x8404e9;

    /* Setup bank size as 64MB*/
    rBANKSIZE = 0xB1;

    /* SDRAM mode , CAS=2 clocks*/
    rMRSRB6 = 0x20;
```

UART

GPHCON 配置 PINMUX 为 UART

ULCONn	Bit	Description	Initial State
Reserved	[7]		0
Infra-Red Mode	[6]	Determine whether or not to use the Infra-Red mode. 0 = Normal mode operation 1 = Infra-Red Tx/Rx mode	0
Parity Mode	[5:3]	Specify the type of parity generation and checking during UART transmit and receive operation. 0xx = No parity 100 = Odd parity 101 = Even parity 110 = Parity forced/checked as 1 111 = Parity forced/checked as 0	000
Number of Stop Bit	[2]	Specify how many stop bits are to be used for end-of-frame signal. 0 = One stop bit per frame 1 = Two stop bit per frame	0
Word Length	[1:0]	Indicate the number of data bits to be transmitted or received per frame. 00 = 5-bits 01 = 6-bits 10 = 7-bits 11 = 8-bits	00

波特率计算:

UART BAUD RATE DIVISOR REGISTER

There are three UART baud rate divisor registers including UBRDIV0, UBRDIV1 and UBRDIV2 in the UART block.
The value stored in the baud rate divisor register (UBRDIVn), is used to determine the serial Tx/Rx clock rate (baud rate) as follows:

UBRDIVn = (int)(PCLK / (bps x 16)) –1

or

UBRDIVn = (int)(UCLK / (bps x 16)) –1

Where, the divisor should be from 1 to (2^{16}-1) and UCLK should be smaller than PCLK.

For example, if the baud-rate is 115200 bps and PCLK or UCLK is 40 MHz, UBRDIVn is:

UBRDIVn = (int)(40000000 / (115200 x 16)) -1
= (int)(21.7) -1
= 21 -1 = 20

PCLK= 50.7MHz, UBRDV = (int)26.5 = 26

U-Boot 移植

下载与准备

wget https://github.com/u-boot/u-boot/archive/refs/tags/v2016.05.tar.gz

在 vscode 的工作区 settings.json 中屏蔽不相关文件夹：

```
{
    "files.exclude": {
        "arch/[b-z]*": true,
        "arch/avr*": true,
        "arch/arc*": true,
        "arch/arm/imx*": true,
        "arch/arm/dts*": true,
        "arch/arm/mach-*": true,
        "arch/arm/cpu/arm7*": true,
        "arch/arm/cpu/arm92[^0]*": true,
        "arch/arm/cpu/arm9[^2]*": true,
        "arch/arm/cpu/arm[^9]*": true,
        "arch/arm/cpu/[b-zB-Z]*": true,
        "arch/arm/cpu/arm920t/[a-r]*": true,
        "arch/arm/include/asm/arch-[^s]*": true,
        "arch/arm/include/asm/arch-s[^3]*": true,
        "arch/arm/include/asm/arch-s3c[^2]*": true,
        "board/[^s]*": true,
        "board/sa[^m]*": true,
        "board/s[^a]*": true,
        "board/samsung/[^s]*": true,
        "board/samsung/smdk[^2]*": true,
        "include/configs/[^s]*": true,
        "include/configs/s[^m]*": true,
        "include/configs/sm[^d]*": true,
        "include/configs/smdk[^2]*": true,
        "configs/[^s]*": true,
        "configs/s[^m]*": true,
        "configs/sm[^d]*": true,
        "configs/smdk[^2]*": true
    },
    "search.exclude": {
        "arch/[b-z]*": true,
```

```
        "arch/avr*": true,
        "arch/arc*": true,
        "arch/arm/imx*": true,
        "arch/arm/dts*": true,
        "arch/arm/mach-*": true,
        "arch/arm/cpu/arm7*": true,
        "arch/arm/cpu/arm92[^0]*": true,
        "arch/arm/cpu/arm9[^2]*": true,
        "arch/arm/cpu/arm[^9]*": true,
        "arch/arm/cpu/[b-zB-Z]*": true,
        "arch/arm/cpu/arm920t/[a-r]*": true,
        "arch/arm/include/asm/arch-[^s]*": true,
        "arch/arm/include/asm/arch-s[^3]*": true,
        "arch/arm/include/asm/arch-s3c[^2]*": true,
        "board/[^s]*": true,
        "board/sa[^m]*": true,
        "board/s[^a]*": true,
        "board/samsung/[^s]*": true,
        "board/samsung/smdk[^2]*": true,
        "include/configs/[^s]*": true,
        "include/configs/s[^m]*": true,
        "include/configs/sm[^d]*": true,
        "include/configs/smdk[^2]*": true,
        "configs/[^s]*": true,
        "configs/s[^m]*": true,
        "configs/sm[^d]*": true,
        "configs/smdk[^2]*": true
    }
}
```

编译

编译方法:

make distclean

make smdk2410_config

make ARCH=arm CROSS_COMPILE=arm-linux-

底层支持

根据前面在 Baremetal 中的实验结果，在 board/Samsung/smdk2410/lowlevel_init.S 中修改 SDRAM 参数:

```
#define B6_MT             0x3 /* SDRAM */
#define B6_Trcd           0x0 /* Trcd 2clock */
#define B6_SCAN           0x1 /* 9bit */

#define B7_MT             0x3 /* SDRAM */
#define B7_Trcd           0x1 /* 3clk */
#define B7_SCAN           0x1 /* 9bit */

/* REFRESH parameter */
#define REFEN             0x1 /* Refresh enable */
#define TREFMD            0x0 /* CBR(CAS before RAS)/Auto
refresh */
#define Trp          0x0 /* 2clk */
#define Trc          0x1 /* 5clk */
#define Tchr              0x2 /* 3clk */
#define REFCNT            1257    /*  */
```

删除 Tchr<<16:

```
SMRDATA:
    .word (0+(B1_BWSCON<<4)+(B2_BWSCON<<8)+(B3_BWSCON<<12)+(B4_BWSCON<<16)+(B5_BWSC
    .word ((B0_Tacs<<13)+(B0_Tcos<<11)+(B0_Tacc<<8)+(B0_Tcoh<<6)+(B0_Tah<<4)+(B0_Ta
    .word ((B1_Tacs<<13)+(B1_Tcos<<11)+(B1_Tacc<<8)+(B1_Tcoh<<6)+(B1_Tah<<4)+(B1_Ta
    .word ((B2_Tacs<<13)+(B2_Tcos<<11)+(B2_Tacc<<8)+(B2_Tcoh<<6)+(B2_Tah<<4)+(B2_Ta
    .word ((B3_Tacs<<13)+(B3_Tcos<<11)+(B3_Tacc<<8)+(B3_Tcoh<<6)+(B3_Tah<<4)+(B3_Ta
    .word ((B4_Tacs<<13)+(B4_Tcos<<11)+(B4_Tacc<<8)+(B4_Tcoh<<6)+(B4_Tah<<4)+(B4_Ta
    .word ((B5_Tacs<<13)+(B5_Tcos<<11)+(B5_Tacc<<8)+(B5_Tcoh<<6)+(B5_Tah<<4)+(B5_Ta
    .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)+REFCNT)
    .word 0x32
    .word 0x30
    .word 0x30
```

Board/Samsung/smdk2410/smdk2410.c 中已经根据手册配置了时钟:

```
#define FCLK_SPEED 1

#if (FCLK_SPEED == 0)       /* Fout = 203MHz, Fin = 12MHz for Audio */
#define M_MDIV  0xC3
#define M_PDIV  0x4
#define M_SDIV  0x1
#elif (FCLK_SPEED == 1)     /* Fout = 202.8MHz */
#define M_MDIV  0xA1
#define M_PDIV  0x3
#define M_SDIV  0x1
#endif
```

NorFlash

在 include/configs/smdk2410.h 中定义了 NorFlash 的信息：

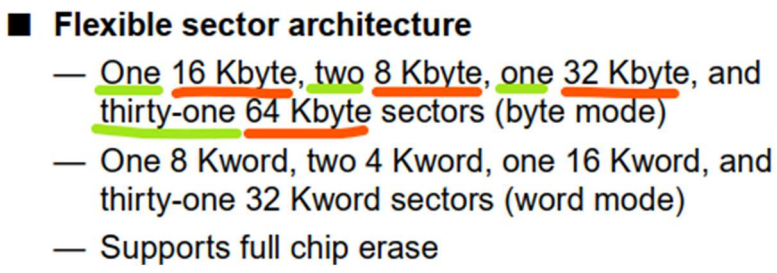
```
#define CONFIG_SYS_FLASH_CFI
#define CONFIG_FLASH_CFI_DRIVER
#define CONFIG_FLASH_CFI_LEGACY
#define CONFIG_SYS_FLASH_LEGACY_512Kx16
#define CONFIG_FLASH_SHOW_PROGRESS  45
```

在 drivers/mtd/jedec_flash.c 中添加 AM29LV160DB 的信息：

```
static const struct amd_flash_info jedec_table[] = {
#ifdef CONFIG_SYS_FLASH_LEGACY_256Kx8
    {
```

```
    {
        .mfr_id     = (u16)AMD_MANUFACT,
        .dev_id     = AM29LV160DB,
        .name       = "AMD AM29LV160DB",
        .uaddr      = {
            [1] = MTD_UADDR_0x0555_0x02AA /* x16 */
        },
        .DevSize    = SIZE_2MiB,
        .CmdSet     = CFI_CMDSET_AMD_LEGACY,
        .NumEraseRegions= 4,
        .regions    = {
            ERASEINFO(0x04000,1),
            ERASEINFO(0x02000,2),
            ERASEINFO(0x08000,1),
            ERASEINFO(0x10000,31),
        }
    },
```

其中.NumEraseRegions 和.regions 对应手册的扇区种类和数量：



- **Flexible sector architecture**
  - One 16 Kbyte, two 8 Kbyte, one 32 Kbyte, and thirty-one 64 Kbyte sectors (byte mode)
  - One 8 Kword, two 4 Kword, one 16 Kword, and thirty-one 32 Kword sectors (word mode)
  - Supports full chip erase

修改 smdk2410.h 中扇区最大数量定义：1+2+1+31 = 35

```
142  #define CONFIG_SYS_MAX_FLASH_SECT   (35)
```

NandFlash

smdk2410.h 中使能 CONFIG_CMD_NAND:

```
213  /*
214   * NAND configuration
215   */
216  #ifdef CONFIG_CMD_NAND
217  #define CONFIG_NAND_S3C2410
218  #define CONFIG_SYS_S3C2410_NAND_HWECC
219  #define CONFIG_SYS_MAX_NAND_DEVICE  1
220  #define CONFIG_SYS_NAND_BASE        0x4E000000
221  #define CONFIG_CMD_NAND_YAFFS
222  #endif
223
```

文件系统

smdk2410.h 配置：

```
169  /*
170   * File system
171   */
172  #define CONFIG_CMD_UBI
173  #define CONFIG_CMD_UBIFS
174  #define CONFIG_CMD_MTDPARTS
175  #define CONFIG_MTD_DEVICE
176  #define CONFIG_MTD_PARTITIONS
177  #define CONFIG_YAFFS2
178  #define CONFIG_RBTREE
179
```

网卡

由于开发板搭载的网卡为 CS8900 和 u-boot 默认相同，所以不需要移植驱动，修改 smdk2410.h 配置：

```
/*
 * Hardware drivers
 */
#define CONFIG_CS8900       /* we have a CS8900 on-board */
#define CONFIG_CS8900_BASE  0x19000300
#define CONFIG_CS8900_BUS16 /* the Linux driver does accesses as shorts */
```

检查 autoconf.h：

```
#define CONFIG_CMD_PING 1
#define CONFIG_CMD_DHCP 1
```

网络参数：

```
#define CONFIG_NETMASK      255.255.255.0
#define CONFIG_IPADDR       192.168.0.166
#define CONFIG_GATEWAYIP    192.168.0.1
#define CONFIG_SERVERIP     192.168.0.116
#define CONFIG_LIB_RAND
#define CONFIG_NET_RANDOM_ETHADDR
#define CONFIG_OVERWRITE_ETHADDR_ONCE
```

此处使用随机 MAC 地址，首次启动后会有 Warning，可以通过 set ethaddr xx:xx:xx:xx:xx:xx 修改：

```
Err:   serial
Net:   CS8900-0
Warning: CS8900-0 (eth0) using random MAC address - aa:f0:28:5f:57:a0
```

分区表

默认 env param 存储在 norflash 上，修改 env param 默认存储器为 Nand Flash:

```
/* #define CONFIG_ENV_ADDR          (CONFIG_SYS_FLASH_BASE + 0x070000)
#define CONFIG_ENV_IS_IN_FLASH
#define CONFIG_ENV_SIZE          0x10000 */
```

```
#define CONFIG_ENV_IS_IN_NAND
#define CONFIG_ENV_OFFSET       0xE0000
#define CONFIG_ENV_SIZE         0x20000
#define CONFIG_ENV_RANGE        0x20000
```

增加 mtd 分区表定义和启动参数定义：

```
#define MTDIDS_DEFAULT      "nand0=s3c2410_nand.0"
#define MTDPARTS_DEFAULT    "mtdparts=s3c2410_nand.0:"  \
                            "896k(uboot)ro,"            \
                            "128k(params)ro,"           \
                            "4m(kernel),"               \
                            "-(filesystem)"

#define CONFIG_BOOTARGS     "console=ttySAC0,115200 root=/dev/mtdblock3"
#define CONFIG_BOOTCOMMAND  "nand read 30000000 kernel 0x400000;bootm 30000000"
```

Common/board_r.c 增加默认执行 mtdparts default 命令：

```
static int run_main_loop(void)
{
#ifdef CONFIG_SANDBOX
    sandbox_main_loop_init();
#endif
    run_command("mtdparts default", 0);
    /* main_loop() can return to retry autoboot, if so just run it again */
    for (;;)
        main_loop();
    return 0;
}
```

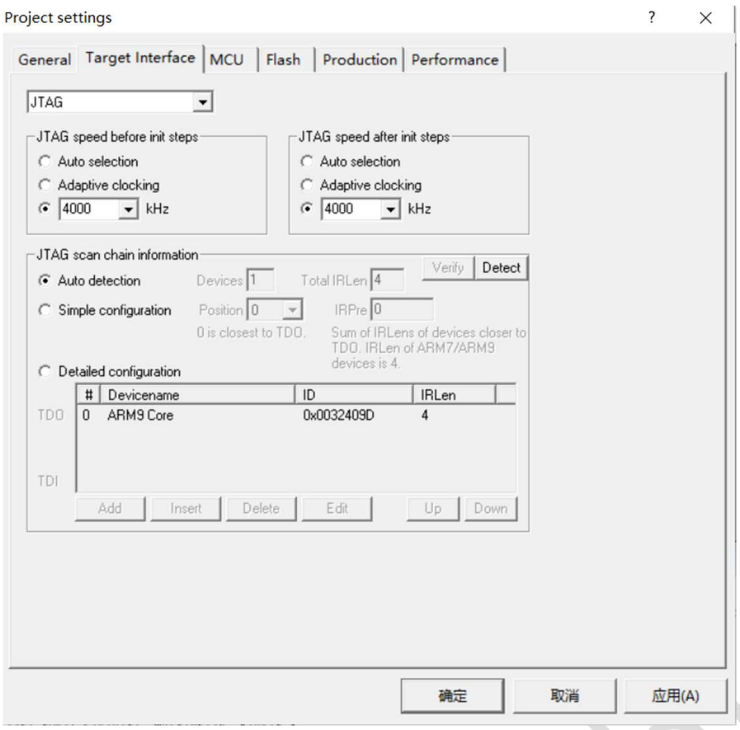
USB 支持

烧写 U-Boot

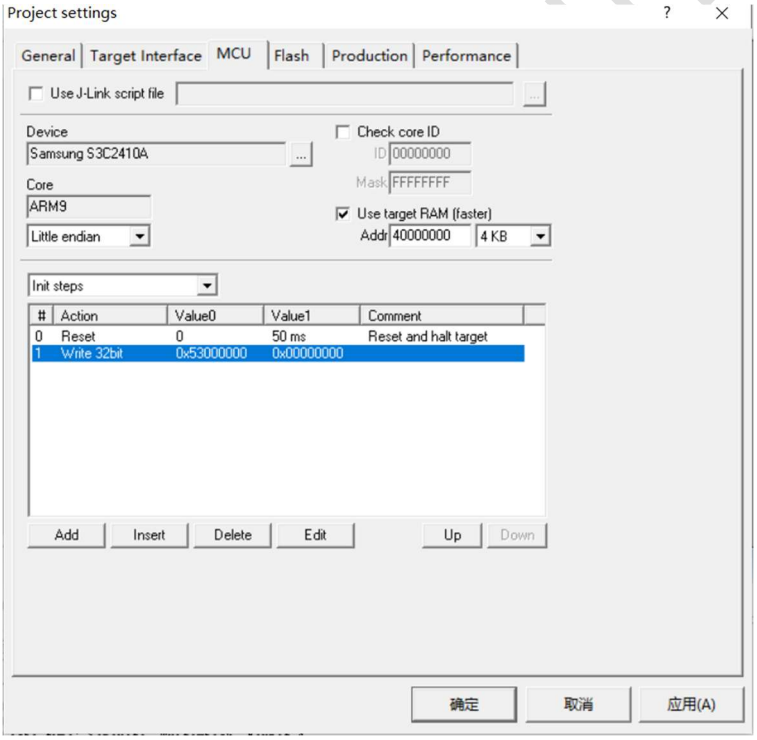
JLink 烧写 U-Boot

打开 jflash.exe

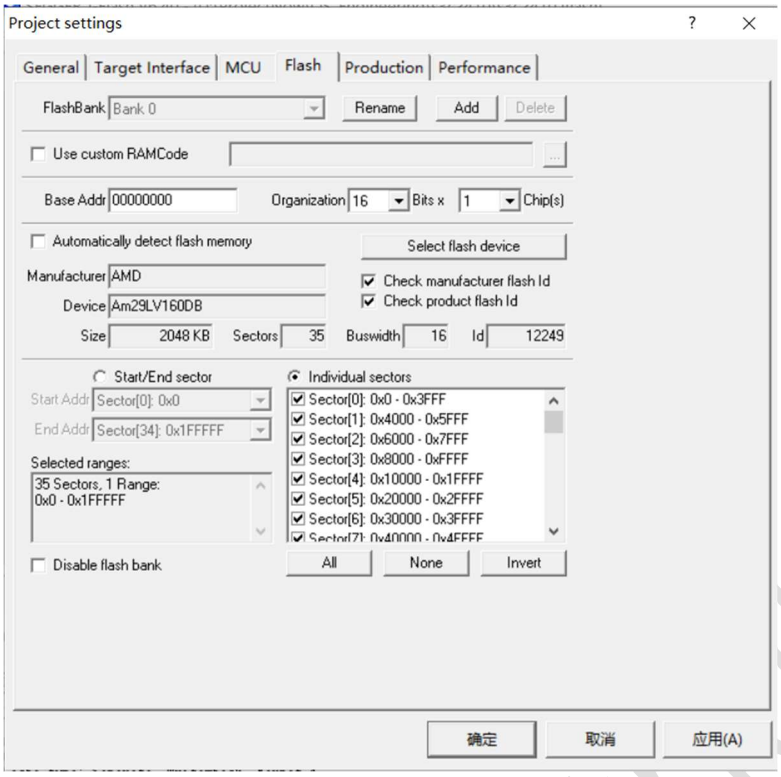
工程设置：



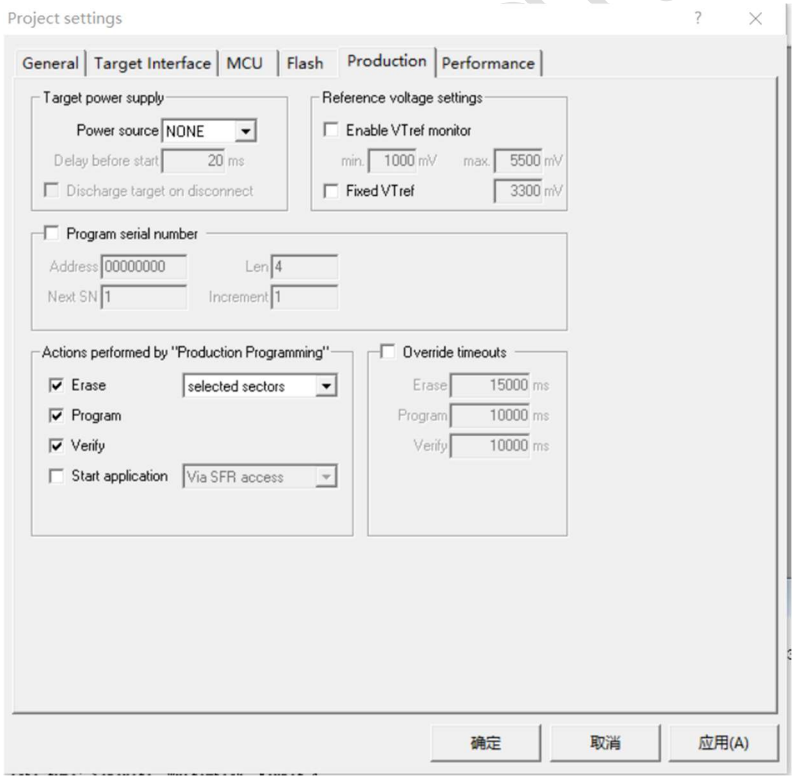
初始化中增加禁用看门狗配置：

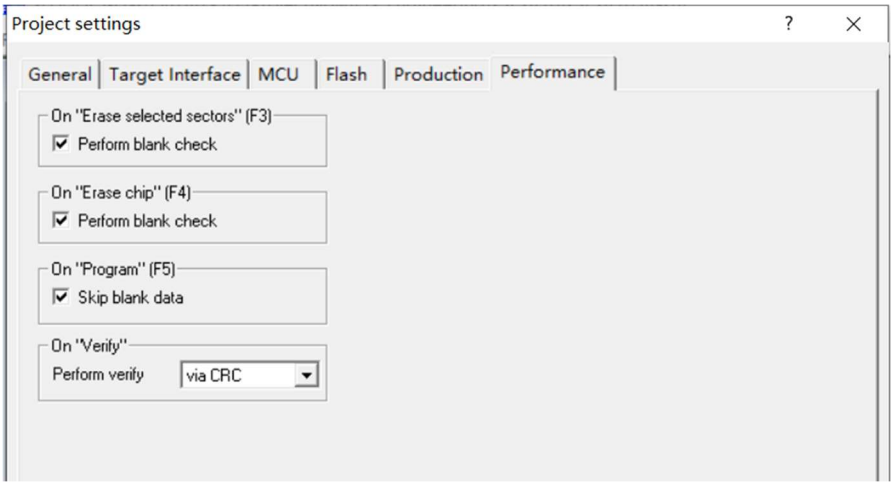


配置 FLASH 型号为 AM29LV160DB：

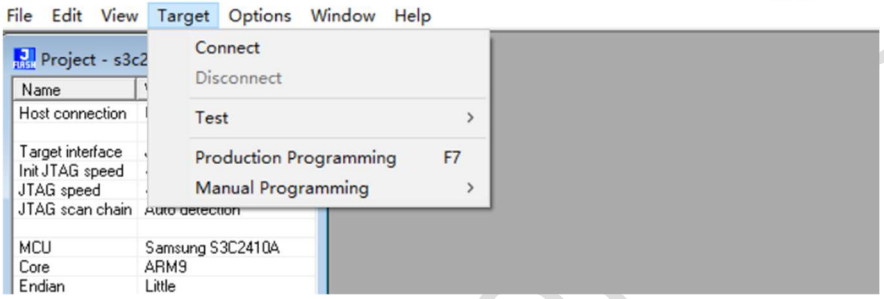


其他配置：

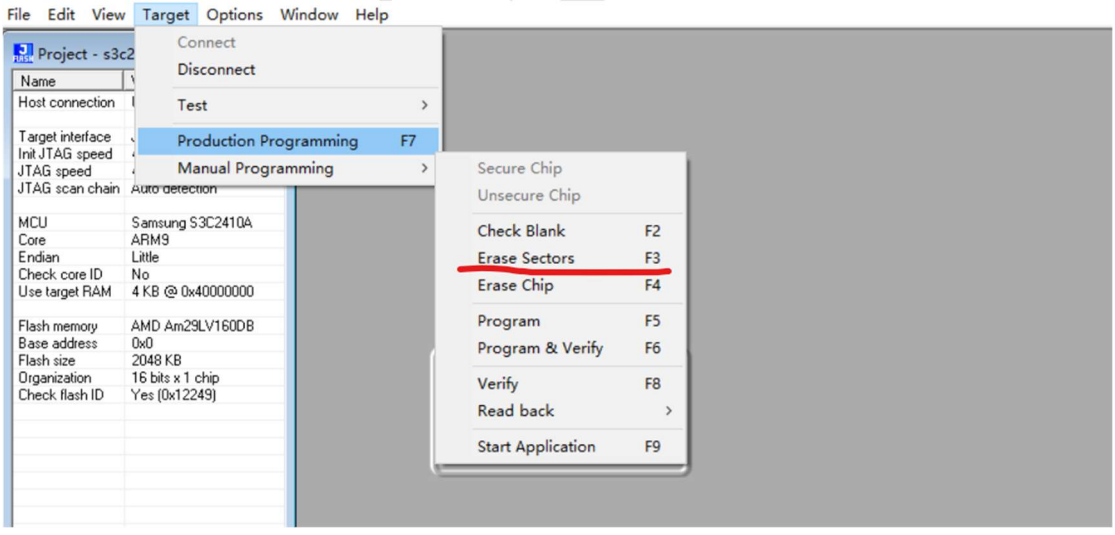


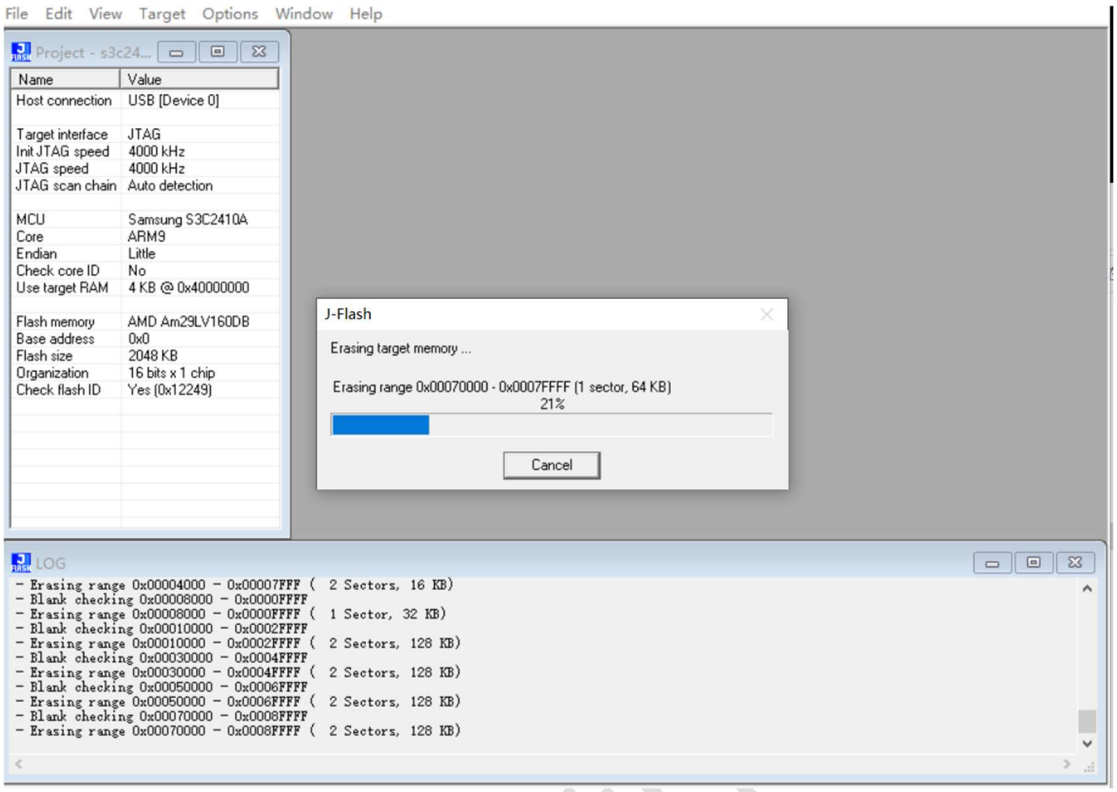


JTAG 连接设备：

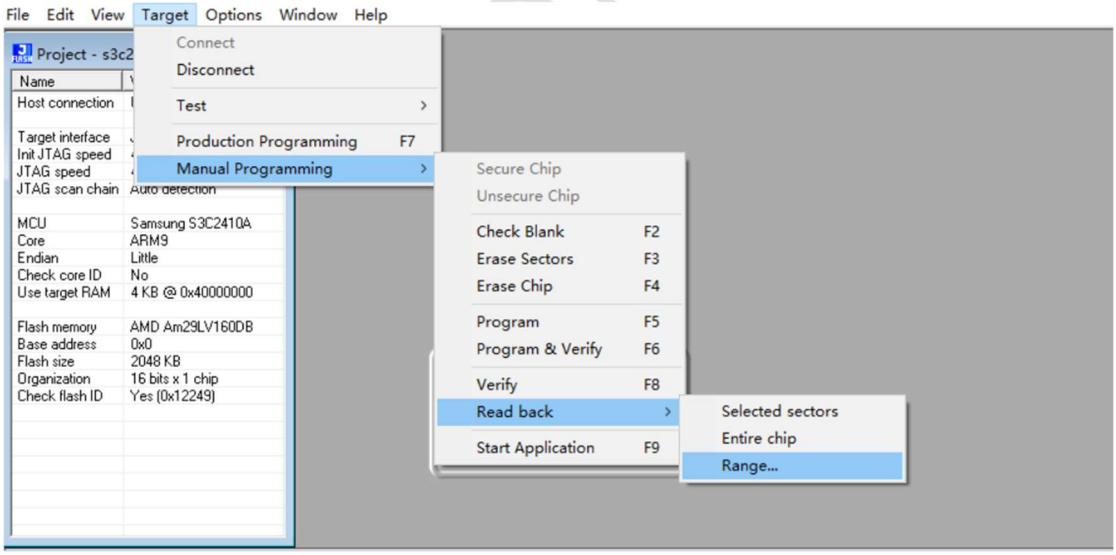


Erase:

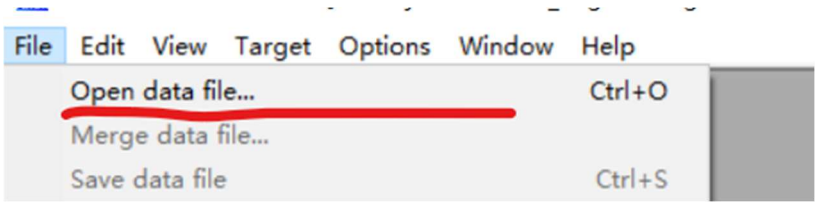




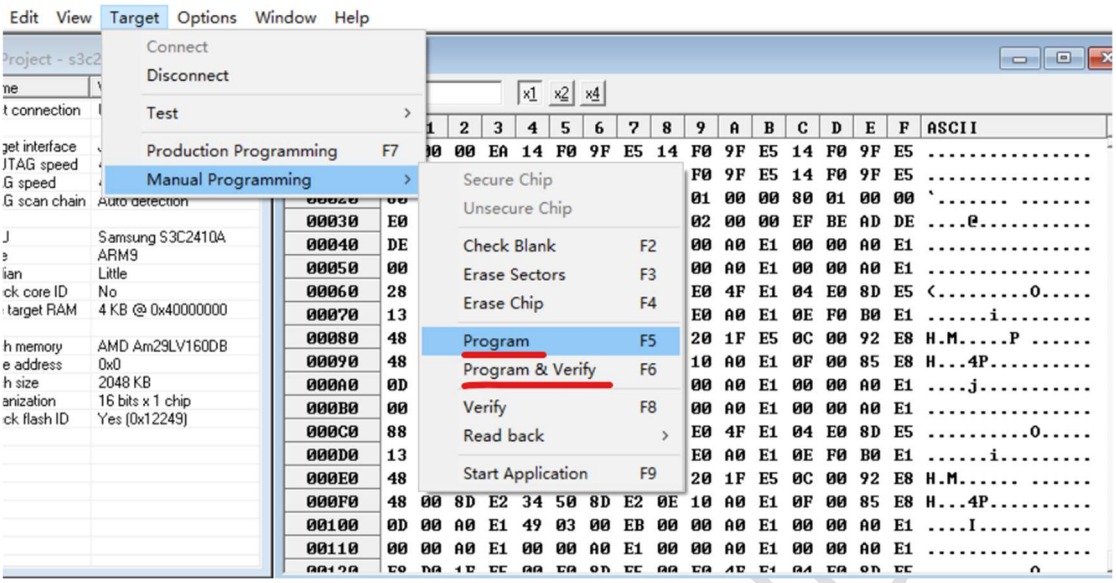
Read Back 检查是否全 FF



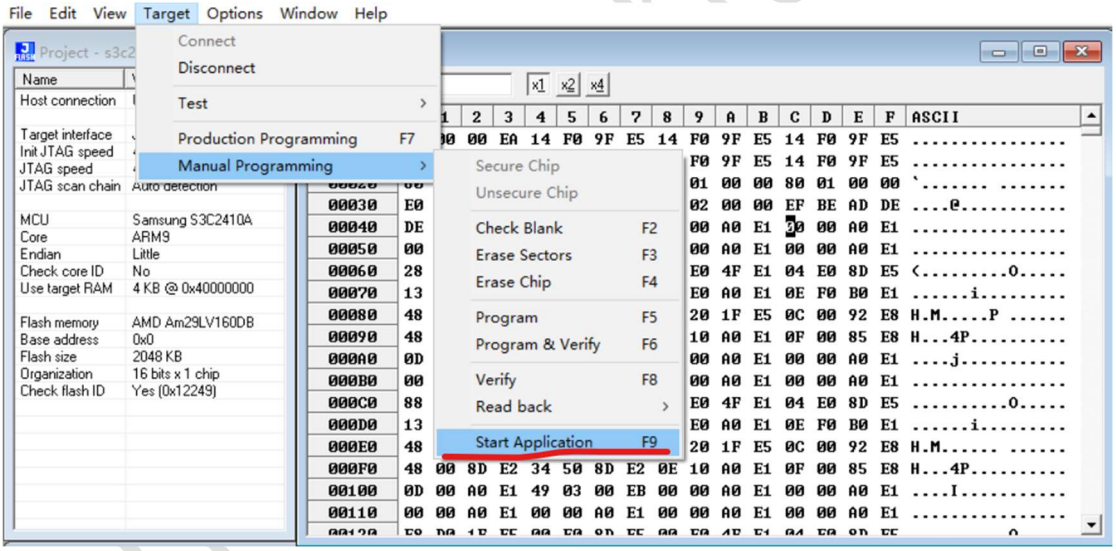
打开 u-boot.bin



烧写或者烧写+验证



启动：



TFTP 烧写 U-Boot

PC 端安装 TFTP Server(Ubuntu):

Sudo apt install tftpd-hpa

修改 tftp 服务器配置：vim /etc/default/tftpd-hpa

重启 tftp 服务：

```
→  boot sudo service tftpd-hpa restart
```

如果已经烧写过 U-Boot 并配置了网卡、IP 和打开了 TFTP 功能（默认），可以通过网络更新 bootload:

tftp 30000000 u-boot.bin

protect off all

erase 0 fffff

cp.b 30000000 0 100000

reset

```
SMDK2410 # tftp 30000000 u-boot2.bin
Using CS8900-0 device
TFTP from server 192.168.0.116; our IP address is 192.168.0.166
Filename 'u-boot2.bin'.
Load address: 0x30000000
Loading: #################################
done
Bytes transferred = 532820 (82154 hex)
```

```
SMDK2410 # protect off all
Un-Protect Flash Bank # 1
SMDK2410 # erase 0 fffff

..................... done
Erased 19 sectors
SMDK2410 # cp.b 30000000 0 100000
Copy to Flash... 9....8....7....6....5....4....3....2....1....done
SMDK2410 #
```

重启:

```
U-Boot 2016.05 (Oct 16 2022 - 22:27:25 +0800)

CPUID: 32410002
FCLK:  202.800 MHz
HCLK:  101.400 MHz
PCLK:   50.700 MHz
DRAM:  64 MiB
WARNING: Caches not enabled
Flash: 2 MiB
NAND:  64 MiB
In:    serial
Out:   serial
Err:   serial
Net:   CS8900-0
Hit any key to stop autoboot:  0
```

JTAG 调试 U-Boot

由于 U-Boot 一般的执行过程是首先在 NorFlash 中执行一部分，随后初始化 SDRAM 并进行 relocate，然后跳转到 SDRAM 后继续执行，由于上电初 SDRAM 不可用和 relocate 过程的存在，使用 JTAG 直接下载并调试 U-Boot 是比较麻烦的。但是用 JTAG Attach 的方式调试 U-Boot 则简单很多，对于部分 U-Boot 代码需要单步调试还是很有帮助。

JTAG Attach 过程并不会复位 CPU，也就是说 CPU 在正常的运行状态，JTAG 此时更像是一个探针，连接上后在对应位置设置断点等待触发就可以了。

这样就引入一个新的问题，U-Boot 的编译和实际 SDRAM 中执行的地址并不一样，前期在 FLASH 中执行是一套地址，而后期则搬移到 SDRAM 中后有是一套地址。u-boot.map 或者 u-boot.elf(即编译产生的 u-boot)的地址和实际运行地址不匹配，那么插入断点的位置就无法确定，位了解决这个问题，首先要直到 u-boot relocate 后的地址。根据后面的分析（Relocate）可知，u-boot relocate 过程中， u-boot 被搬移的目标地址的计算是在 board_f.c 中函数：reserve_uboot()中，修改打印：

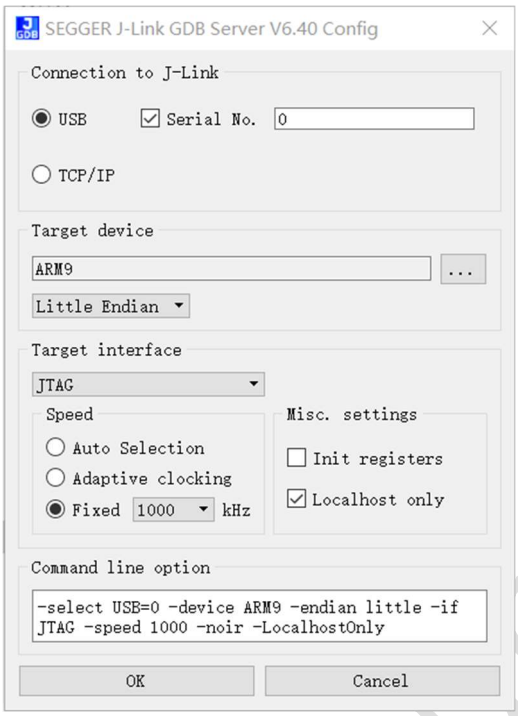
```
508
509     printf("Reserving %ldk for U-Boot at: %08lx\n", gd->mon_len >> 10,
510           gd->relocaddr);
511
```

那么通过打印就可以获取实际的搬移后的 u-boot 地址：

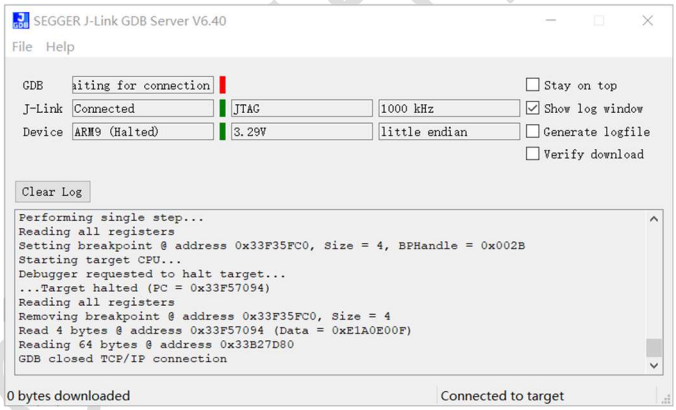
```
FCLK:  202.800 MHz
HCLK:  101.400 MHz
PCLK:   50.700 MHz
DRAM:  Reserving 798k for U-Boot at: 33f28000
64 MiB
WARNING: Caches not enabled
Flash: 2 MiB
```

开始连接 JTAG：

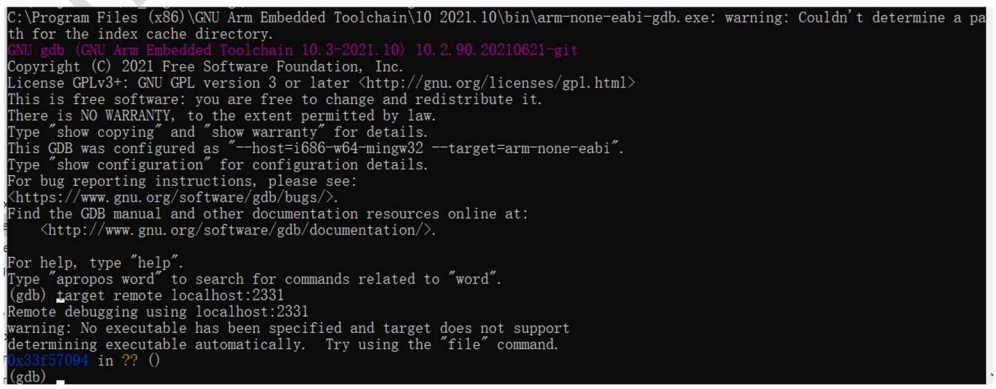
1) 烧写 u-boot 并正常启动，进入 hush shell
2) 确保 JLink 连接正确，打开 JLinkGDBServer 或者 JLinkGDBServerCL, 配置目标位 arm9，取消“Init registers”



3) 启动 GDB Server:



4) 在 u-boot 目录启动 gdb client: 输入 arm-none-eabi-gdb, 启动后输入 target remote localhost:2331 连接 GDB Server



5) monitor halt 暂停 CPU， monitor regs 读取寄存器：当前的 PC 指向 SDRAM

```
(gdb) monitor halt
(gdb) monitor regs
PC = 33F57094, CPSR = 200000D3 (SVC mode, ARM FIQ dis. IRQ dis.)
R0 = 33F9F4D8, R1 = 33F92626, R2 = 00000000, R3 = 33FB785C
R4 = 33FA1328, R5 = 33F9B2AC, R6 = 33FB614C, R7 = 00000001
USR: R8 =00000000, R9 =33B27F08, R10=33FA12F4, R11 =00000001, R12 =00000000
     R13=BEC39CE8, R14=000D7B70
FIQ: R8 =00000000, R9 =00000000, R10=00000000, R11 =00000000, R12 =00000000
     R13=C0665924, R14=C0665924, SPSR=F00000FF
SVC: R13=33B27DB0, R14=33F57094, SPSR=00000013
ABT: R13=C066590C, R14=C0119220, SPSR=80000093
IRQ: R13=C0665900, R14=C0119020, SPSR=60000093
UND: R13=C0665918, R14=C0665918, SPSR=F00000FF
(gdb)
```

6) 加载符号文件到 U-Boot relocate 后的地址：

```
(gdb) add-symbol-file u-boot 0x33f28000
add symbol table from file "u-boot" at
        .text_addr = 0x33f28000
(y or n) y
Reading symbols from u-boot...
(gdb)
```

7) 添加断点，查看断点：

```
(gdb) b cli_hush.c:2967
Breakpoint 1 at 0x33f35fc0: file common/cli_hush.c, line 2967.
(gdb) b run_list_real
Breakpoint 2 at 0x33f35624: file common/cli_hush.c, line 1763.
(gdb) info b
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x33f35fc0 in parse_stream at common/cli_hush.c:2967
2       breakpoint     keep y   0x33f35624 in run_list_real at common/cli_hush.c:1763
(gdb)
```

8) delete [num] 删除断点，c 或者 continue 继续，输入命令等待触发中断

```
(gdb) c
Continuing.

Breakpoint 1, parse_stream (end_trigger=<optimized out>, input=<optimized out>, ctx=<optimized out>,
    dest=<optimized out>) at common/cli_hush.c:2967
2967                    m = map[ch];
(gdb)
```

ARM Linux GDB

前面提到的 GDB 编译可以使用系统自带的 GDB 也可以使用其他 arm 工具链自带的 GDB（如果由，前面的例子用的是 arm-none-eabi-gdb），基本的调试没有差别，但是查看反汇编的时候就不行了，需要针对这个平台做支持，可以下载 GDB 的源码然后编译成支持再 host 平台(x86)下运行，调试 ARM 架构的 GDB 程序。

GDB 源码的下载：https://ftp.gnu.org/gnu/gdb/， 建议使用和当前系统自带 GDB 版本一致的源码

下载好后解压，比如将下载好后的 gdb-9.2.tar.xz 解压到 gdb-9.2 中，执行一下命令：

cd gdb-9.2

mkdir build

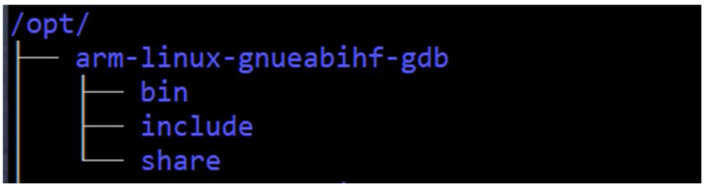
cd build

./configure --target=arm-linux-gnueabihf --prefix=/opt/arm-linux-gnueabihf-gdb

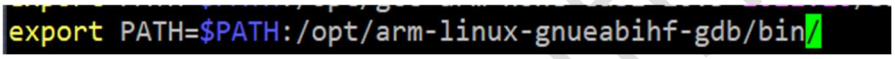
make

sudo make install

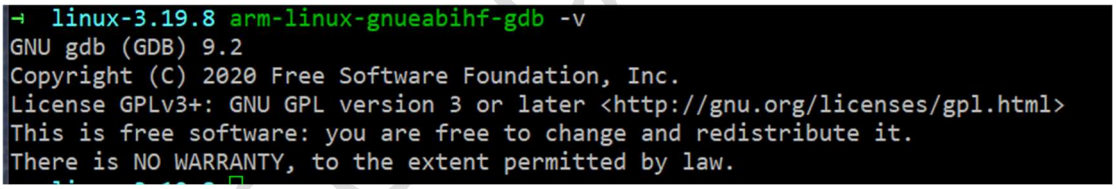
成功后，再/opt/目录下可以看到新生成的 arm-linux-gnueabihf-gdb/



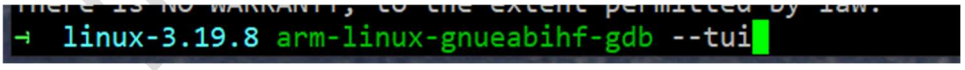
将/opt/arm-linux-gnueabihf-gdb/bin/目录添加到环境变量：



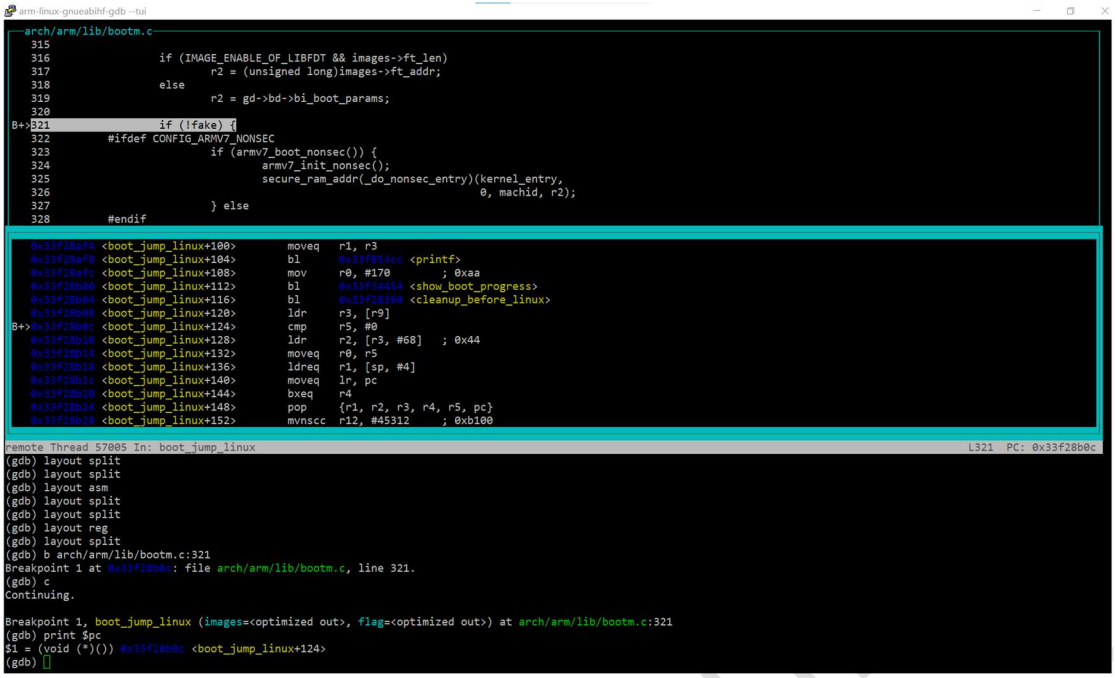
测试执行：



执行成功后，可以开启 TUI 模式：



开启 tui 后，其他的 GDB 操作和前面的完全一致。连上目标后，加载完符号文件，输入 layout split, 或者 layout asm, layout src, layout reg 查看不同窗口：



Linux 内核移植

下载与编译

wget https://mirrors.edge.kernel.org/pub/linux/kernel/v3.x/linux-3.19.8.tar.gz

将 U-Boot 中 mkinage 工具添加至 PATH 中：

```
# Path to your oh-my-zsh installation.
export ZSH="$HOME/.oh-my-zsh"
export PATH=$PATH:/usr/arm-linux-toolchains4.4.3/bin
export PATH=$PATH:/home/lab/s3c2410/uboot/u-boot-2016.05/tools
export PATH=$PATH:/opt/SEGGER/JLink_V640/
export PATH=$PATH:~/.local/bin/
```

```
tools
tools vim ~/.zshrc
tools source ~/.zshrc
tools
```

查看 S3C2410 默认配置文件：

```
linux-3.19.8 ll arch/arm/configs | grep 2410
-rw-rw-r-- 1 lab lab  11K 5月  11  2015 s3c2410_defconfig
```

生成配置文件：

```
linux-3.19.8 make ARCH=arm CROSS_COMPILE=arm-linux- s3c2410_defconfig
#
# configuration written to .config
#
```

可选： 替换顶层 Makefile 中 ARCH 与 CROSS_COMPILE 定义，可以省去每次 make 时输入 ARCH=arm CROSS_COMPILE=arm-linux-

```
250  # Note: Some architectures assign CROSS_COMPILE in their arch
251  ARCH        ?= arm
252  CROSS_COMPILE   ?= arm-linux-
253
```

Kernel 中的 Machine ID: arch/arm/tools/mach-types:

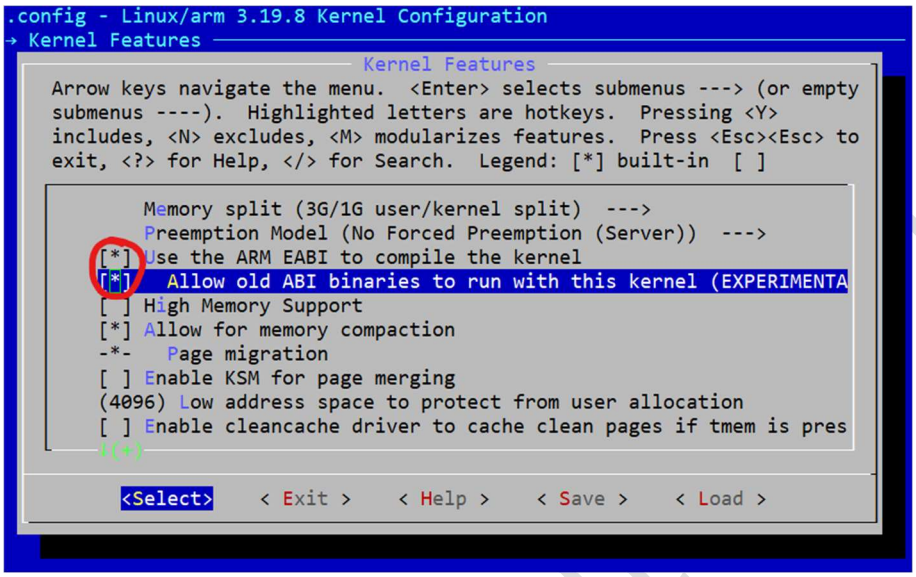
```
66  ks8695          ARCH_KS8695     KS8695          180
67  smdk2410        ARCH_SMDK2410       SMDK2410        193
68  ceiva           ARCH_CEIVA      CEIVA           200
```

U-Boot 中的 Machine ID, arch/arm/include/asm/mach-types.h:

```
58  #define MACH_TYPE_KS8695                180
59  #define MACH_TYPE_SMDK2410              193
60  #define MACH_TYPE_CEIVA                 200
```

修改内核配置：make menuconfig:

Kernel Features 目录下：



为了能直接生成 U-Boot 支持的 uImage 文件，首先先将前面 U-Boot 编译阶段产生的 mkimage 所在路径添加至环境变量，例如 u-boot-xxxx/tools/mkimage:

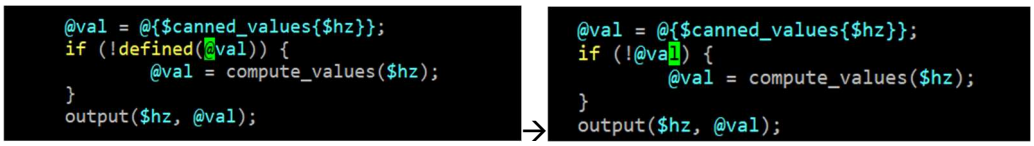
```
# Path to your oh-my-zsh installation.
export ZSH="$HOME/.oh-my-zsh"
export PATH=$PATH:/usr/arm-linux-toolchains4.4.3/bin
export PATH=$PATH:/home/lab/s3c2410/uboot/u-boot-2012.04.01/tools
export PATH=$PATH:/opt/SEGGER/JLink_V640/
export PATH=$PATH:~/.local/bin/
export PATH=$PATH:/opt/gcc-arm-none-eabi-10.3-2021.10/bin/
```

编译：

```
→  linux-3.19.8 make uImage
```

```
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
  UIMAGE  arch/arm/boot/uImage
Image Name:   Linux-3.19.8
Created:      Mon Oct 17 01:38:13 2022
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    2814024 Bytes = 2748.07 kB = 2.68 MB
Load Address: 30108000
Entry Point:  30108000
  Image arch/arm/boot/uImage is ready
```

编译中可能会遇到 timeconst.pl 编译错误，修改 kernel/timeconst.pl 373 行：



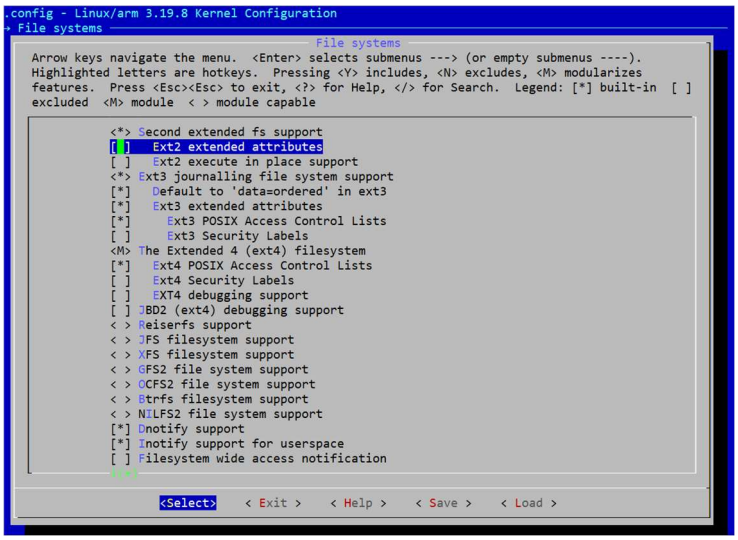
分区表

在 arch/arm/mach-s3c24xx/common-smdk.c 下修改默认分区表：

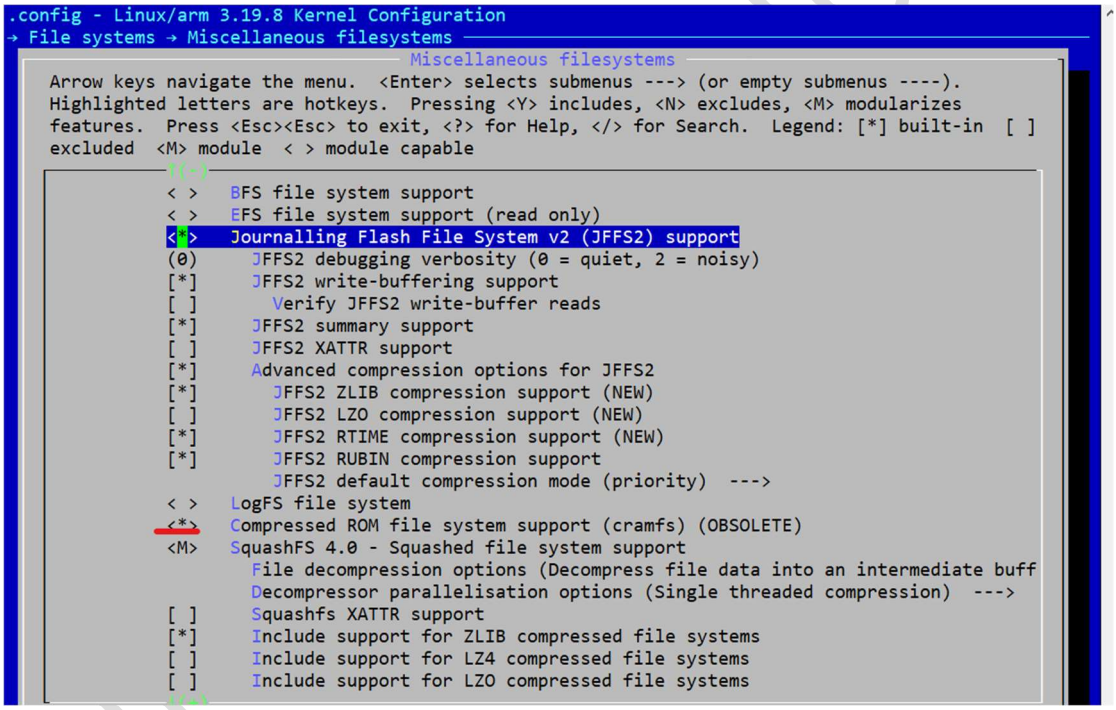
```
112 static struct mtd_partition smdk_default_nand_part[] = {
113     [0] = {
114         .name   = "loader",
115         .size   = SZ_512K+SZ_256K+SZ_128K,
116         .offset = 0,
117     },
118     [1] = {
119         .name   = "params",
120         .offset = SZ_512K+SZ_256K+SZ_128K,
121         .size   = SZ_128K,
122     },
123     [2] = {
124         .name   = "kernel",
125         .offset = SZ_1M,
126         .size   = SZ_4M,
127     },
128     [3] = {
129         .name   = "rootfs",
130         .offset = SZ_4M+SZ_1M,
131         .size   = MTDPART_SIZ_FULL
132     }
133 };
```

配置内核文件系统

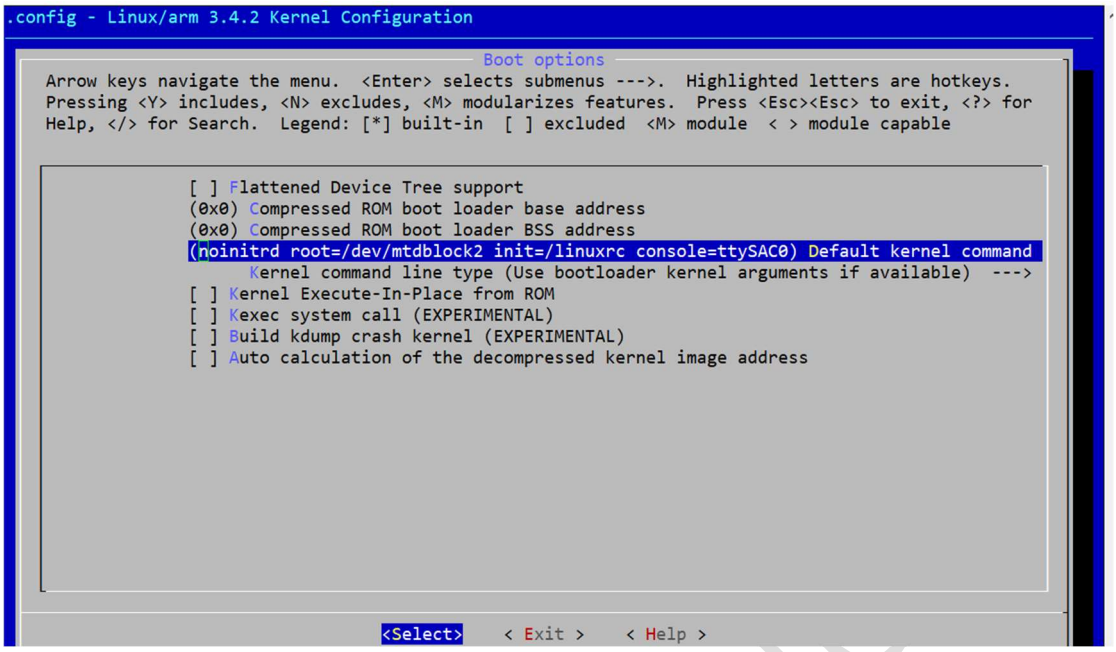
关闭 Ext2 支持：



CRAMFS



Boot Option:

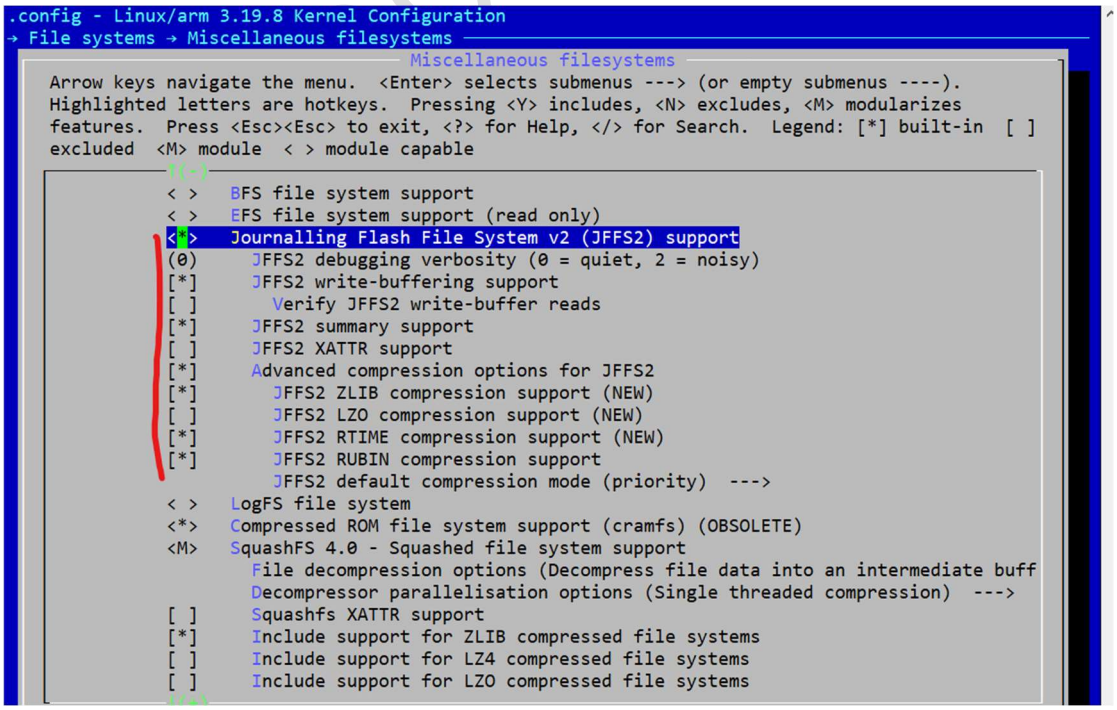


如果在 U-Boot 中传入了启动参数，则在 U-Boot 中设置：

```
SMDK2410 # setenv bootargs noinitrd root=/dev/mtdblock3 init=/linuxrc console=tt
ySAC0,115200 rootfstype=jffs2
SMDK2410 # saveenv
```

JFFS2

配置 JFFS2:



YAFFS2

烧写内核

SDRAM 中调试

将 uImage 通过 TFTP 拷贝至一个不干扰解压内核解压至 0x30108000 地址的内存中：

```
SMDK2410 #
SMDK2410 # tftp 31000000 uImage
Using CS8900-0 device
TFTP from server 192.168.0.116; our IP address is 192.168.0.166
Filename 'uImage'.
Load address: 0x31000000
Loading: #################################################################
         #################################################################
         ##
done
Bytes transferred = 1933520 (1d80d0 hex)
```

启动内核：

```
SMDK2410 # tftp 31000000 uImage3
Using CS8900-0 device
TFTP from server 192.168.0.116; our IP address is 192.168.0.166
Filename 'uImage3'.
Load address: 0x31000000
Loading: #################################################################
         #################################################################
         #############################################################
         203.1 KiB/s
done
Bytes transferred = 2814088 (2af088 hex)
SMDK2410 # bootm 31000000
## Booting kernel from Legacy Image at 31000000 ...
   Image Name:   Linux-3.19.8
   Created:      2022-10-16  17:38:13 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2814024 Bytes = 2.7 MiB
   Load Address: 30108000
   Entry Point:  30108000
   Verifying Checksum ... OK
   Loading Kernel Image ... OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
Booting Linux on physical CPU 0x0
Linux version 3.19.8 (lab@thelaptop) (gcc version 4.4.3 (ctng-1.6.1) ) #2 Mon Oct 17 01:38:08 CST 2022
CPU: ARM920T [41129200] revision 0 (ARMv4T), cr=0000717f
CPU: VIVT data cache, VIVT instruction cache
Machine: SMDK2410
Memory policy: Data cache writeback
CPU S3C2410A (id 0x32410002)
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 16256
Kernel command line: console=ttySAC0,115200 root=/dev/mtdblock3 rootfstype=jffs2
PID hash table entries: 256 (order: -2, 1024 bytes)
Dentry cache hash table entries: 8192 (order: 3, 32768 bytes)
Inode-cache hash table entries: 4096 (order: 2, 16384 bytes)
Memory: 59228K/65536K available (3936K kernel code, 237K rwdata, 1136K rodata, 180K init, 177K bss, 6308K r
eserved, 0K cma-reserved)
Virtual kernel memory layout:
    vector  : 0xffff0000 - 0xffff1000   (   4 kB)
    fixmap  : 0xffc00000 - 0xfff00000   (3072 kB)
```

烧写 NAND FLASH

下载 kernel 至 SDRAM：tftp 30000000 uImage

```
SMDK2410 # tftp 30000000 uImage3
Using CS8900-0 device
TFTP from server 192.168.0.116; our IP address is 192.168.0.166
Filename 'uImage3'.
Load address: 0x30000000
Loading: #################################################################
         #################################################################
         ###############################################################
         317.4 KiB/s
done
Bytes transferred = 2814088 (2af088 hex)
SMDK2410 #
```

擦除扇区：nand erase.part kernel

```
SMDK2410 # nand erase.part kernel

NAND erase.part: device 0 offset 0x100000, size 0x400000
Erasing at 0x4fc000 -- 100% complete.
OK
SMDK2410 #
```

烧写 Flash: nand write 30000000 kernel

```
SMDK2410 # nand write 30000000 kernel

NAND write: device 0 offset 0x100000, size 0x400000
 4194304 bytes written: OK
SMDK2410 #
```

文件系统

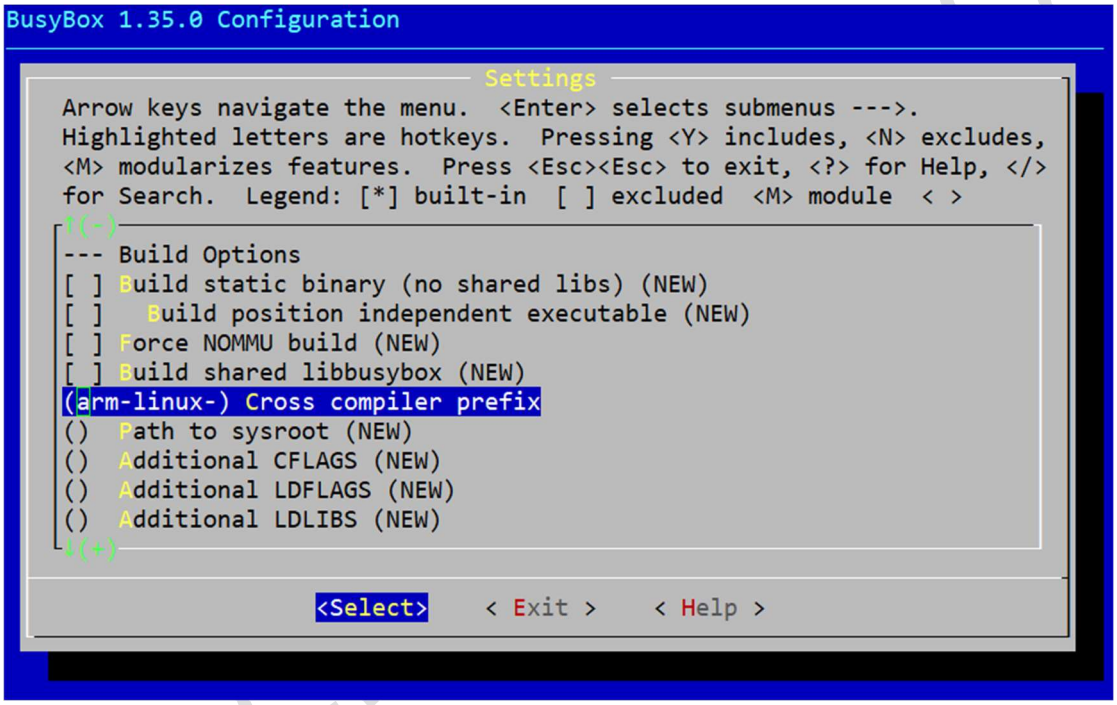
BusyBox

下载与编译

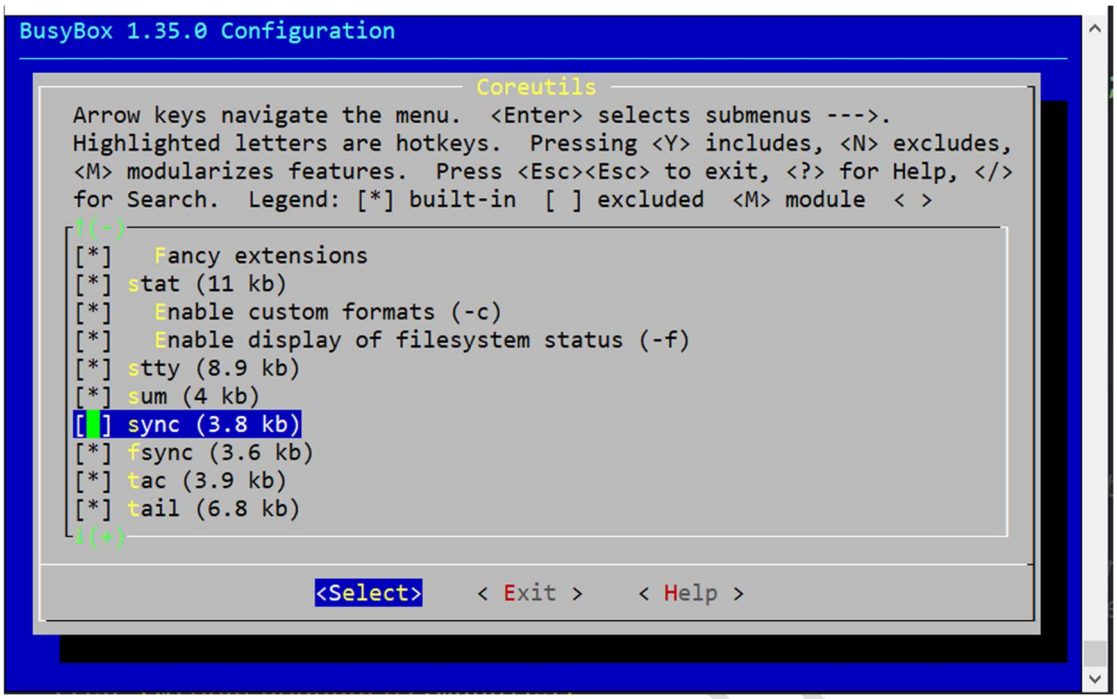
wget https://busybox.net/downloads/busybox-1.35.0.tar.bz2

make ARCH=arm menuconfig

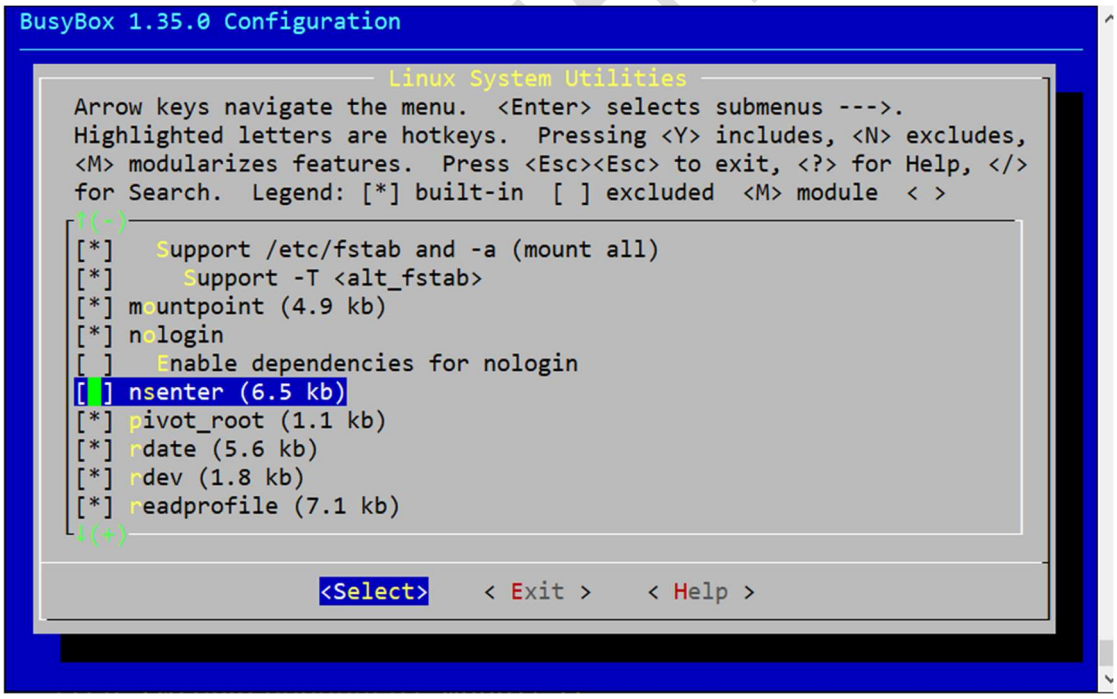
Settings->build Options->Cross compiler prefix:



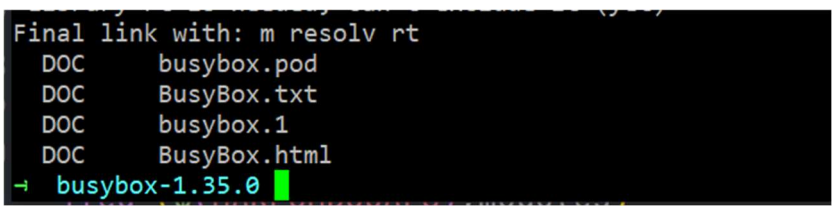
Core utils->disable sync:



Linux system utilty->disable nsenter:



Make ARCH=arm



拷贝生成的文件到目标目录：

```
→  busybox-1.35.0 make ARCH=arm install CONFIG_PREFIX=../../qrootfs/
```

```
→  qrootfs ll
总用量 12K
drwxrwxr-x 2 lab lab 4.0K 10月 17 23:57 bin
lrwxrwxrwx 1 lab lab   11 10月 17 23:57 linuxrc -> bin/busybox
drwxrwxr-x 2 lab lab 4.0K 10月 17 23:57 sbin
drwxrwxr-x 4 lab lab 4.0K 10月 17 23:57 usr
```

构造文件系统

cd qrootfs

mkdir lib

cp -d /usr/arm-linux-toolchains4.4.3/arm-none-linux-gnueabi/sys-root/lib/*so* lib

```
→  qrootfs ls lib/
ld-2.9.so                libmemusage.so          libpng.so.3
ld-linux.so.3            libm.so.6               libpng.so.3.35.0
libanl-2.9.so            libmudflap.so           libpthread-2.9.so
libanl.so.1              libmudflap.so.0         libpthread.so.0
libBrokenLocale-2.9.so   libmudflap.so.0.0.0     libresolv-2.9.so
libBrokenLocale.so.1     libmudflapth.so         libresolv.so.2
libc-2.9.so              libmudflapth.so.0       librt-2.9.so
libcrypt-2.9.so          libmudflapth.so.0.0.0   librt.so.1
libcrypt.so.1            libnsl-2.9.so           libSegFault.so
libc.so.6                libnsl.so.1             libssp.so
libdl-2.9.so             libnss_compat-2.9.so    libssp.so.0
libdl.so.2               libnss_compat.so.2      libssp.so.0.0.0
libgcc_s.so              libnss_dns-2.9.so       libstdc++.so
libgcc_s.so.1            libnss_dns.so.2         libstdc++.so.6
libgomp.so               libnss_files-2.9.so     libstdc++.so.6.0.13
libgomp.so.1             libnss_files.so.2       libthread_db-1.0.so
```

mkdir usr/lib

cp -d /usr/arm-linux-toolchains4.4.3/arm-none-linux-gnueabi/sys-root/usr/lib/*so* usr/lib

```
→  qrootfs ls usr/lib
libanl.so           libc.so_orig      libnss_files.so     libresolv.a
libbfd-2.20.so      libdl.so          libnss_hesiod.so    libresolv.so
libbfd.so           libm.so           libnss_nisplus.so   librt.so
libBrokenLocale.so  libnsl.so         libnss_nis.so       libthread_db.so
libcrypt.so         libnss_compat.so  libpthread.so       libutil.so
libc.so             libnss_dns.so     libpthread.so_orig
```

Mkdir etc

Vim etc/inittab

/etc/inittab

::sysinit:/etc/init.d/rcS

console::askfirst:-/bin/sh

::ctrlaltdel:/sbin/reboot

```
# /etc/inittab
::sysinit:/etc/init.d/rcS
console::askfirst:-/bin/sh
::ctrlaltdel:/sbin/reboot
~
```

Mkdir init.d

vim etc/init.d/rcS

#!/bin/sh

mount -a

mkdir /dev/pts

mount -t devpts devpts /dev/pts

echo /sbin/mdev > /proc/sys/kernel/hotplug

mdev -s

```
#!/bin/sh
mount -a
mkdir /dev/pts
mount -t devpts devpts /dev/pts
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
```

```
→  qrootfs sudo chmod +x etc/init.d/rcS
```

Vim etc/fstab

#device mount-point type options dump fsck order

proc /proc proc defaults 0 0

tmpfs /tmp tmpfs defaults 0 0

sysfs /sys sysfs defaults 0 0

tmpfs /dev tmpfs defaults 0 0

```
#device  mount-point    type    options dump    fsck    order
proc    /proc   proc    defaults        0       0
tmpfs   /tmp    tmpfs   defaults        0       0
sysfs   /sys    sysfs   defaults        0       0
tmpfs   /dev    tmpfs   defaults        0       0i
~
```

mkdir dev

cd dev

sudo mknod console c 5 1

sudo mknod null c 1 3

cd ..

mkdir proc mnt tmp sys root

mkdir lib/module/3.19.8,这里 3.19.8 是内核版本号，根据不同内核版本修改

生成文件系统

CRAMFS

下载 cramfs1.1:

wget https://udomain.dl.sourceforge.net/project/cramfs/cramfs/1.1/cramfs-1.1.tar.gz

修改 cramfsck.c mkcramfs.c 两处头文件定义：

```
39  //#include <sys/types.h>
40  #include <sys/sysmacros.h>
```

make

```
→  s3c2410 ./xrootfs/cramfs-1.1/mkcramfs qrootfs qrootfs.cramfs
Directory data: 10940 bytes
Everything: 5784 kilobytes
Super block: 76 bytes
CRC: 4be9bc03
warning: gids truncated to 8 bits (this may be a security concern)
```

JFFS2

mkfs.jffs2 -n -e 0x4000 -d qrootfs -o qrootfs.jffs2 --pad=0x800000

这里需要注意-n 和 -e 0x4000 两个参数，需要根据具体 flash 的参数来进行配置

YAFFS2

TBD

烧写文件系统

CRAMFS

```
Environment size: 525/131068 bytes
SMDK2410 # tftp 30000000 qrootfs.cramfs
Using CS8900-0 device
TFTP from server 192.168.0.116; our IP address is 192.168.0.166
Filename 'qrootfs.cramfs'.
Load address: 0x30000000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ###############
         335.9 KiB/s
done
Bytes transferred = 5922816 (5a6000 hex)
SMDK2410 # nand erase.part filesystem

NAND erase.part: device 0 offset 0x500000, size 0x3b00000
Erasing at 0x3ffc000 -- 100% complete.
OK
SMDK2410 # nand write 30000000 500000 5a6000

NAND write: device 0 offset 0x500000, size 0x5a6000
 5922816 bytes written: OK
SMDK2410 #
```

更新 bootargs:

```
SMDK2410 # set bootargs console=ttySAC0,115200 root=/dev/mtdblock3
SMDK2410 # saveenv
Saving Environment to NAND...
Erasing NAND...
Erasing at 0xfc000 -- 100% complete.
Writing to NAND... OK
```

```
VFS: Mounted root (cramfs filesystem) readonly on device 31:3.
Freeing unused kernel memory: 180K (c05fd000 - c062a000)

Please press Enter to activate this console.
/ # ls
bin           lib           proc          sbin          usr
dev           linuxrc       qrootfs.jffs2 sys
etc           mnt           root          tmp
/ # cat /proc/cpuinfo
processor       : 0
model name      : ARM920T rev 0 (v4l)
BogoMIPS        : 50.17
Features        : swp half
CPU implementer : 0x41
CPU architecture: 4T
CPU variant     : 0x1
CPU part        : 0x920
```

JFFS2

U-Boot 中执行:

tftp 30000000 qrootfs.jffs2

nand erase.part filesystem

nand write.jffs2 30000000 500000 800000

```
SMDK2410 # tftp 30000000 qrootfs.jffs2
Using CS8900-0 device
TFTP from server 192.168.0.116; our IP address is 192.168.0.166
Filename 'qrootfs.jffs2'.
Load address: 0x30000000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ##########################################
         331.1 KiB/s
done
Bytes transferred = 8388608 (800000 hex)
SMDK2410 # nand erase.part filesystem

NAND erase.part: device 0 offset 0x500000, size 0x3b00000
Erasing at 0x3ffc000 -- 100% complete.
OK
SMDK2410 # nand write.jffs2 30000000 500000 800000

NAND write: device 0 offset 0x500000, size 0x800000
 8388608 bytes written: OK
SMDK2410 #
```

Setup bootargs:

```
SMDK2410 # set bootargs console=ttySAC0,115200 root=/dev/mtdblock3 rootfstype=jf
fs2
```

saveenv and printenv:

```
SMDK2410 # saveenv
Saving Environment to NAND...
Erasing NAND...
Erasing at 0xfc000 -- 100% complete.
Writing to NAND... OK
SMDK2410 #
```

```
bootargs=console=ttySAC0,115200 root=/dev/mtdblock3 rootfstype=jffs2
```

```
VFS: Mounted root (jffs2 filesystem) on device 31:3.
Freeing unused kernel memory: 180K (c05fd000 - c062a000)

Please press Enter to activate this console.
/ #
/ #
/ # ls
bin             lib             proc            sbin            usr
dev             linuxrc         qrootfs.jffs2   sys
etc             mnt             root            tmp
/ #
```

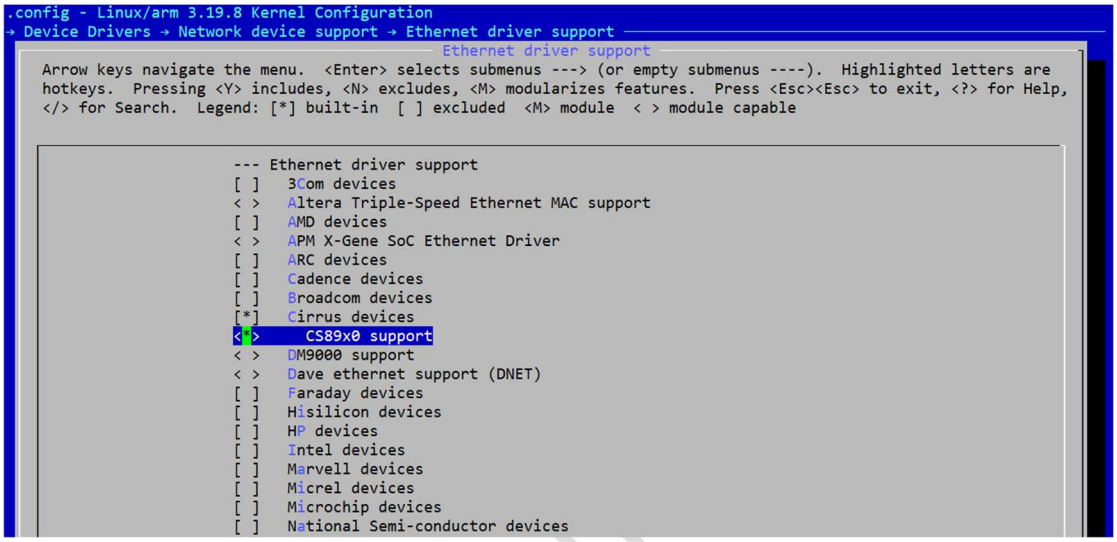
YAFFS2

TBD

修改配置网卡驱动

为了能更好的支持后续的调试，在 Kernel 中支持网络是非常有帮助的，这里参考了文章：
https://www.cnblogs.com/normalmanzhao2003/p/11355691.html

首先在 Menuconfig 中启动默认 CS89x0 驱动，Device Drivers→Network device support→Ethernet driver support→CS89x0 support:



由于 Kernel 3.19.8 与文章中用的 3.4.2 稍有不同：

在定义了 CONFIG_PLAT_S3C24XX 的情况下 CONFIG_NO_IOPORT_MAP 会被定义：

```
10  if ARCH_S3C24XX
11
12  config PLAT_S3C24XX
13      def_bool y
14      select ARCH_REQUIRE_GPIOLIB
15      select NO_IOPORT_MAP
16      select S3C_DEV_NAND
17      select IRQ_DOMAIN
18      help
19        Base platform code for any Samsung S3C24XX device
```

从而导致没有启用 CONFIG_HAS_IOPORT_MAP:

```
371 config HAS_IOPORT_MAP
372     boolean
373     depends on HAS_IOMEM && !NO_IOPORT_MAP
374     default y
375
```

从而导致：

1. CONFIG_CS89x0_PLATFORM 会被强制定义：

```
config CS89x0_PLATFORM
    bool "CS89x0 platform driver support" if HAS_IOPORT_MAP
    default !HAS_IOPORT_MAP
    depends on CS89x0
    help
      Say Y to compile the cs89x0 driver as a platform driver. This
      makes this driver suitable for use on certain evaluation boards
      such as the iMX21ADS.

      If you are unsure, say N.
```

2. ioport_map 函数没有定义:

```
#ifdef CONFIG_HAS_IOPORT_MAP
/* Create a virtual mapping cookie for an IO port range */
void __iomem *ioport_map(unsigned long port, unsigned int nr)
{
    if (port > PIO_MASK)
        return NULL;
    return (void __iomem *) (unsigned long) (port + PIO_OFFSET);
}
```

从而进一步导致后面驱动无法被正常的执行 probe，所以此处修改 drivers/net/ethernet/cirrus/Kconfig:

```
config CS89x0_PLATFORM
    bool "CS89x0 platform driver support"
#if HAS_IOPORT_MAP
    default !HAS_IOPORT_MAP
    depends on CS89x0
    help
      Say Y to compile the cs89x0 driver as a platform driver. This
      makes this driver suitable for use on certain evaluation boards
      such as the iMX21ADS.
```

menuconfig 中取消"CS89x0 platform driver support":

```
< >     Broadcom GENET internal MAC support
[*]   Cirrus devices
<*>     CS89x0 support
[ ]       CS89x0 platform driver support
< >   DM9000 support
```

修改 drivers/net/ethernet/cirrus/cs89x0.c:

```
#endif
#define DEBUGGING 1

#include "cs89x0.h"
```

```
#ifdef CONFIG_ARCH_S3C24XX
    #include <asm/irq.h>
    #include <../mach-s3c24xx/regs-mem.h>
    #define S3C24XX_PA_CS8900 0x19000000
    static unsigned int netcard_portlist[] __used __initdata = {0, 0};
    static unsigned int cs8900_irq_map[] = {IRQ_EINT9, 0, 0, 0};
#else
    #ifndef CONFIG_CS89x0_PLATFORM
    static unsigned int netcard_portlist[] __used __initdata = {
        0x300, 0x320, 0x340, 0x360, 0x200, 0x220, 0x240,
        0x260, 0x280, 0x2a0, 0x2c0, 0x2e0, 0};
    static unsigned int cs8900_irq_map[] = {
        10, 11, 12, 5};
    #endif
#endif
```

net_open 函数:

```
static int
net_open(struct net_device *dev)
{
    struct net_local *lp = netdev_priv(dev);
    int result = 0;
    int i;
    int ret;

#if !defined(CONFIG_SH_HICOSH4) && !defined(CONFIG_ARCH_PNX010X) && !defined(CONFIG_ARCH_S3C2
    if (dev->irq < 2) {
        /* Allow interrupts to be generated by the chip */
/* Cirrus' release had this: */
#if 0
        writereg(dev, PP_BusCTL, readreg(dev, PP_BusCTL) | ENABLE_IRQ);
```

```
            if (i >= CS8920_NO_INTS) {
                writereg(dev, PP_BusCTL, 0);    /* disable interrupts. */
                pr_err("can't get an interrupt\n");
                ret = -EAGAIN;
                goto bad_out;
            }
        }
        else
#endif
        {
#if !defined(CONFIG_CS89x0_PLATFORM) && !defined(CONFIG_ARCH_S3C24XX)
            if (((1 << dev->irq) & lp->irq_map) == 0) {
                pr_err("%s: IRQ %d is not in our map of allowable IRQs, which is %x\n",
                       dev->name, dev->irq, lp->irq_map);
                ret = -EAGAIN;
                goto bad_out;
            }
#endif
/* FIXME: Cirrus' release had this: */
            writereg(dev, PP_BusCTL, readreg(dev, PP_BusCTL)|ENABLE_IRQ);
/* And 2.3.47 had this: */
#if 0
```

```
#if 0
        writereg(dev, PP_BusCTL, ENABLE_IRQ | MEMORY_ON);
#endif
        write_irq(dev, lp->chip_type, dev->irq);
#if defined(CONFIG_ARCH_S3C24XX)
        ret = request_irq(dev->irq, &net_interrupt, IRQF_TRIGGER_RISING, dev->name, dev);
#else
        ret = request_irq(dev->irq, net_interrupt, 0, dev->name, dev);
#endif
        if (ret) {
            pr_err("request_irq(%d) failed\n", dev->irq);
            goto bad_out;
        }
    }
```

```
    /* check to make sure that they have the "right" hardware available */
    switch (lp->adapter_cnf & A_CNF_MEDIA_TYPE) {
    case A_CNF_MEDIA_10B_T:
        result = lp->adapter_cnf & A_CNF_10B_T;
        break;
    case A_CNF_MEDIA_AUI:
        result = lp->adapter_cnf & A_CNF_AUI;
        break;
    case A_CNF_MEDIA_10B_2:
        result = lp->adapter_cnf & A_CNF_10B_2;
        break;
    default:
        result = lp->adapter_cnf & (A_CNF_10B_T |
                    A_CNF_AUI |
                    A_CNF_10B_2);
    }
#if defined(CONFIG_ARCH_PNX010X) || defined(CONFIG_ARCH_S3C24XX)
    result = A_CNF_10B_T;
#endif
    if (!result) {
        pr_err("%s: EEPROM is configured for unavailable media\n",
               dev->name);
release_dma:
#if ALLOW_DMA
        free_dma(dev->dma);
release_irq:
        release_dma_buff(lp);
```

cs89x0_probe 函数:

```
struct net_device * __init cs89x0_probe(int unit)
{
    struct net_device *dev = alloc_etherdev(sizeof(struct net_local));
    unsigned *port;
    int err = 0;
    int irq;
    int io;
#if defined(CONFIG_ARCH_S3C24XX)
    unsigned int oldval_bwscon;
    unsigned int oldval_bankcon3;
#endif
```

```
1685
1686    sprintf(dev->name, "eth%d", unit);
1687    netdev_boot_setup_check(dev);
1688    io = dev->base_addr;
1689    irq = dev->irq;
1690 #if defined(CONFIG_ARCH_S3C24XX)
1691    if (netcard_portlist[0])
1692        return ERR_PTR(-ENODEV);
1693    netcard_portlist[0] = (unsigned int)ioremap(S3C24XX_PA_CS8900, SZ_1M) + 0x300;
1694    dev->dev_addr[0] = 0xB8;
1695    dev->dev_addr[1] = 0x27;
1696    dev->dev_addr[2] = 0xEB;
1697    dev->dev_addr[3] = 0x23;
1698    dev->dev_addr[4] = 0x5C;
1699    dev->dev_addr[5] = 0x11;
1700
1701    oldval_bwscon = *((volatile unsigned int *)S3C2410_BWSCON);
1702    *((volatile unsigned int *)S3C2410_BWSCON) = (oldval_bwscon & ~(3 << 12)) | \
1703                                                 (1 << 12) | \
1704                                                 (1 << 14) | \
1705                                                 (1 << 15);
1706    oldval_bankcon3 = *((volatile unsigned int *)S3C2410_BANKCON3);
1707    *((volatile unsigned int *)S3C2410_BANKCON3) = 0x1f7c;
1708 #endif
1709
1710    cs89_dbg(0, info, "cs89x0_probe(0x%x)\n", io);
1711
```

```
1710    cs89_dbg(0, info, "cs89x0_probe(0x%x)\n", io);
1711
1712    if (io > 0x1ff) {    /* Check a single specified location. */
1713 #ifdef CONFIG_ARCH_S3C24XX
1714        err = cs89x0_probe1(dev, (void *)io, 0);
1715 #else
1716        err = cs89x0_ioport_probe(dev, io, 0);
1717 #endif
1718    } else if (io != 0) {    /* Don't probe at all. */
1719        err = -ENXIO;
1720    } else {
1721        for (port = netcard_portlist; *port; port++) {
1722 #ifdef CONFIG_ARCH_S3C24XX
1723            if (cs89x0_probe1(dev, (void *)*port, 0) == 0)
1724 #else
1725            if (cs89x0_ioport_probe(dev, *port, 0) == 0)
1726 #endif
1727                break;
1728            dev->irq = irq;
1729        }
1730        if (!*port)
1731            err = -ENODEV;
```

```
1735    return dev;
1736 out:
1737 #if defined(CONFIG_ARCH_S3C24XX)
1738    iounmap((void *)netcard_portlist[0]);
1739    netcard_portlist[0] = 0;
1740
1741    *((volatile unsigned int *)S3C2410_BWSCON) = oldval_bwscon;
1742    *((volatile unsigned int *)S3C2410_BANKCON3) = oldval_bankcon3;
1743 #endif
1744    free_netdev(dev);
1745    pr_warn("no cs8900 or cs8920 detected.  Be sure to disable PnP with SETUP\n");
1746    return ERR_PTR(err);
1747 }
1748 #endif
1749 #endif
```

init_module 函数：如果不是编译 MODULE 可以不更改

```
1823  int __init init_module(void)
1824  {
1825      struct net_device *dev = alloc_etherdev(sizeof(struct net_local));
1826      struct net_local *lp;
1827      int ret = 0;
1828  #if defined(CONFIG_ARCH_S3C24XX)
1829      unsigned int oldval_bwscon;
1830      unsigned int oldval_bankcon3;
1831  #endif
1832
1833  #if DEBUGGING
```

```
1838      if (!dev)
1839          return -ENOMEM;
1840
1841  #if defined(CONFIG_ARCH_S3C24XX)
1842      dev->base_addr = io = (unsigned int)ioremap(S3C24XX_PA_CS8900, SZ_1M) + 0x300;
1843      dev->irq = irq = cs8900_irq_map[0];
1844
1845      dev->dev_addr[0] = 0xB8;
1846      dev->dev_addr[1] = 0x27;
1847      dev->dev_addr[2] = 0xEB;
1848      dev->dev_addr[3] = 0x23;
1849      dev->dev_addr[4] = 0x5C;
1850      dev->dev_addr[5] = 0x11;
1851
1852      oldval_bwscon = *((volatile unsigned int *)S3C2410_BWSCON);
1853      *((volatile unsigned int *)S3C2410_BWSCON) = (oldval_bwscon & ~(3 << 12)) | \
1854                                                   (1<<12) | \
1855                                                   (1<<14) | \
1856                                                   (1<<15);
1857      oldval_bankcon3 = *((volatile unsigned int *)S3C2410_BANKCON3);
1858      *((volatile unsigned int *)S3C2410_BANKCON3) = 0x1f7c;
1859  #else
1860      dev->irq = irq;
1861      dev->base_addr = io;
1862  #endif
1863
1864      lp = netdev_priv(dev);
```

```
1912      return 0;
1913  out:
1914  #if defined(CONFIG_ARCH_S3C24XX)
1915      iounmap(dev->base_addr);
1916      *((volatile unsigned int *)S3C2410_BWSCON) = oldval_bwscon;
1917      *((volatile unsigned int *)S3C2410_BANKCON3) = oldval_bankcon3;
1918  #endif
1919      free_netdev(dev);
1920      return ret;
1921  }
1922
```

执行编译后下载调试，启动日志中可以看到 CS8900A 的初始化信息：

```
0x000000500000-0x000004000000 : "rootfs"
cs89x0: cs89x0_probe(0x0)
cs89x0: v2.4.3-pre1 Russell Nelson <nelson@crynwr.com>, Andrew Morton
cs89x0: eth0: cs8900 rev K found at c4c00300
cs89x0: Extended EEPROM checksum bad and no Cirrus EEPROM, relying on command line
cs89x0: media  IRQ 57, programmed I/O, MAC b8:27:eb:23:5c:11
cs89x0: cs89x0_probe1() successful
ohci_hcd: USB 1.1 'Open' Host Controller (OHCI) Driver
ohci-s3c2410: OHCI S3C2410 driver
```

进入命令行后，探测 eth0:

```
/ # ifconfig eth0
eth0      Link encap:Ethernet  HWaddr B8:27:EB:23:5C:11
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:57
```

网络设备已经存在，设置 IP 的子网掩码:

```
/ # ifconfig eth0 add 192.168.1.2 netmask 255.255.255.0 up
cs89x0: eth0: using half-duplex 10Base-T (RJ-45)
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr B8:27:EB:23:5C:11
          inet addr:192.168.1.2  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:57
```

添加默认网关:

```
/ # route add default gw 192.168.1.1
/ # route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         192.168.1.1     0.0.0.0         UG    0      0        0 eth0
192.168.1.0     *               255.255.255.0   U     0      0        0 eth0
```

测试网络连接:

```
/ # ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: seq=0 ttl=64 time=3.709 ms
64 bytes from 192.168.1.1: seq=1 ttl=64 time=1.688 ms
64 bytes from 192.168.1.1: seq=2 ttl=64 time=1.605 ms
64 bytes from 192.168.1.1: seq=3 ttl=64 time=1.606 ms
^C
--- 192.168.1.1 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 1.605/2.152/3.709 ms
```

最后可以在/etc/init.d/rcS 中添加默认系统启动后初始化网卡的命令:

```
random: nonblocking pool is initialized
mount -a
mkdir /dev/pts
mount -t devpts devpts /dev/pts
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s

ifconfig eth0 add 192.168.1.2 netmask 255.255.255.0 up
route add default gw 192.168.1.1

~
```

跟目录下创建用户文件夹并修改权限，使用 tftp 命令测试服务是否正常：

```
/ # mkdir share
/ # ls
bin      etc      linuxrc  proc     sbin     sys      usr
dev      lib      mnt      root     share    tmp
/ # chmod 777 share/
/ # cd share/
/share # tftp -gr uImage 192.168.1.1
uImage              100% |*******************************|  2761k  0:00:00 ETA
/share #
```

启动日志

```
U-Boot 2016.05 (Oct 16 2022 - 22:27:25 +0800)

CPUID: 32410002
FCLK:  202.800 MHz
HCLK:  101.400 MHz
PCLK:   50.700 MHz
DRAM:  64 MiB
WARNING: Caches not enabled
Flash: 2 MiB
NAND:  64 MiB
In:    serial
Out:   serial
Err:   serial
Net:   CS8900-0
Hit any key to stop autoboot:  0

NAND read: device 0 offset 0x100000, size 0x400000
 4194304 bytes read: OK
## Booting kernel from Legacy Image at 30000000 ...
   Image Name:   Linux-3.19.8
   Created:      2022-10-16  17:38:13 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2814024 Bytes = 2.7 MiB
```

```
   Load Address: 30108000
   Entry Point:  30108000
   Verifying Checksum ... OK
   Loading Kernel Image ... OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
Booting Linux on physical CPU 0x0
Linux version 3.19.8 (lab@thelaptop) (gcc version 4.4.3
(ctng-1.6.1) ) #2 Mon Oct 17 01:38:08 CST 2022
CPU: ARM920T [41129200] revision 0 (ARMv4T), cr=0000717f
CPU: VIVT data cache, VIVT instruction cache
Machine: SMDK2410
Memory policy: Data cache writeback
CPU S3C2410A (id 0x32410002)
Built 1 zonelists in Zone order, mobility grouping
on.  Total pages: 16256
Kernel command line: console=ttySAC0,115200
root=/dev/mtdblock3
PID hash table entries: 256 (order: -2, 1024 bytes)
Dentry cache hash table entries: 8192 (order: 3, 32768
bytes)
Inode-cache hash table entries: 4096 (order: 2, 16384
bytes)
Memory: 59228K/65536K available (3936K kernel code, 237K
rwdata, 1136K rodata, 180K init, 177K bss, 6308K
reserved, 0K cma-reserved)
Virtual kernel memory layout:
    vector  : 0xffff0000 - 0xffff1000   (   4 kB)
    fixmap  : 0xffc00000 - 0xfff00000   (3072 kB)
    vmalloc : 0xc4800000 - 0xff000000   ( 936 MB)
    lowmem  : 0xc0000000 - 0xc4000000   (  64 MB)
    modules : 0xbf000000 - 0xc0000000   (  16 MB)
      .text : 0xc0108000 - 0xc05fc7c4   (5074 kB)
      .init : 0xc05fd000 - 0xc062a000   ( 180 kB)
      .data : 0xc062a000 - 0xc06657e0   ( 238 kB)
       .bss : 0xc06657e0 - 0xc0691f7c   ( 178 kB)
NR_IRQS:111
irq: clearing pending status 00000002
irq: clearing pending status 00000200
```

```
sched_clock: 16 bits at 1014kHz, resolution 986ns, wraps
every 64630177ns
Console: colour dummy device 80x30
Calibrating delay loop... 50.17 BogoMIPS (lpj=125440)
pid_max: default: 32768 minimum: 301
Mount-cache hash table entries: 1024 (order: 0, 4096
bytes)
Mountpoint-cache hash table entries: 1024 (order: 0, 4096
bytes)
CPU: Testing write buffer coherency: ok
Setting up static identity map for 0x304c2ed0 -
0x304c2f4c
NET: Registered protocol family 16
DMA: preallocated 256 KiB pool for atomic coherent
allocations
S3C Power Management, Copyright 2004 Simtec Electronics
S3C2410: Initialising architecture
SCSI subsystem initialized
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb
s3c-i2c s3c2410-i2c.0: slave address 0x10
s3c-i2c s3c2410-i2c.0: bus frequency set to 99 KHz
s3c-i2c s3c2410-i2c.0: i2c-0: S3C I2C adapter
Advanced Linux Sound Architecture Driver Initialized.
Switched to clocksource samsung_clocksource_timer
NET: Registered protocol family 2
TCP established hash table entries: 1024 (order: 0, 4096
bytes)
TCP bind hash table entries: 1024 (order: 0, 4096 bytes)
TCP: Hash tables configured (established 1024 bind 1024)
TCP: reno registered
UDP hash table entries: 256 (order: 0, 4096 bytes)
UDP-Lite hash table entries: 256 (order: 0, 4096 bytes)
NET: Registered protocol family 1
RPC: Registered named UNIX socket transport module.
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
RPC: Registered tcp NFSv4.1 backchannel transport module.
NetWinder Floating Point Emulator V0.97 (extended
precision)
```

```
futex hash table entries: 256 (order: -1, 3072 bytes)
jffs2: version 2.2. (NAND) (SUMMARY)  ? 2001-2006 Red
Hat, Inc.
romfs: ROMFS MTD (C) 2007 Red Hat, Inc.
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
s3c2410-lcd s3c2410-lcd: no platform data for lcd, cannot
attach
s3c2410-lcd: probe of s3c2410-lcd failed with error -22
Serial: 8250/16550 driver, 4 ports, IRQ sharing enabled
s3c2410-uart.0: ttySAC0 at MMIO 0x50000000 (irq = 74,
base_baud = 0) is a S3C2410
console [ttySAC0] enabled
s3c2410-uart.1: ttySAC1 at MMIO 0x50004000 (irq = 77,
base_baud = 0) is a S3C2410
s3c2410-uart.2: ttySAC2 at MMIO 0x50008000 (irq = 80,
base_baud = 0) is a S3C2410
lp: driver loaded but no devices found
ppdev: user-space parallel port driver
brd: module loaded
loop: module loaded
Uniform Multi-Platform E-IDE driver
ide-gd driver 1.18
ide-cd driver 5.00
s3c24xx-nand s3c2410-nand: Tacls=3, 29ns Twrph0=7 69ns,
Twrph1=3 29ns
s3c24xx-nand s3c2410-nand: NAND soft ECC
nand: device found, Manufacturer ID: 0x98, Chip ID: 0x76
nand: Toshiba NAND 64MiB 3,3V 8-bit
nand: 64 MiB, SLC, erase size: 16 KiB, page size: 512,
OOB size: 16
Scanning device for bad blocks
Creating 4 MTD partitions on "NAND":
0x000000000000-0x0000000e0000 : "loader"
0x0000000e0000-0x000000100000 : "params"
0x000000100000-0x000000500000 : "kernel"
0x000000500000-0x000004000000 : "rootfs"
ohci_hcd: USB 1.1 'Open' Host Controller (OHCI) Driver
ohci-s3c2410: OHCI S3C2410 driver
s3c2410-ohci s3c2410-ohci: OHCI Host Controller
```

```
s3c2410-ohci s3c2410-ohci: new USB bus registered,
assigned bus number 1
s3c2410-ohci s3c2410-ohci: irq 42, io mem 0x49000000
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 2 ports detected
usbcore: registered new interface driver usbserial
usbcore: registered new interface driver
usbserial_generic
usbserial: USB Serial support registered for generic
usbcore: registered new interface driver ftdi_sio
usbserial: USB Serial support registered for FTDI USB
Serial Device
usbcore: registered new interface driver pl2303
usbserial: USB Serial support registered for pl2303
mousedev: PS/2 mouse device common for all mice
s3c2410-wdt s3c2410-wdt: watchdog inactive, reset
disabled, irq disabled
Driver 'mmcblk' needs updating - please use bus_type
methods
TCP: cubic registered
NET: Registered protocol family 17
drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
ALSA device list:
  No soundcards found.
VFS: Mounted root (cramfs filesystem) readonly on device
31:3.
Freeing unused kernel memory: 180K (c05fd000 - c062a000)

Please press Enter to activate this console. random:
nonblocking pool is initialized
```

U-Boot 分析

U-Boot 编译

make smdk2410_config

Makefile 中按照顺序依次依赖 FORCE，outputmakefile, scripts_basic

```
475
476  %config: scripts_basic outputmakefile FORCE
477      $(Q)$(MAKE) $(build)=scripts/kconfig $@
478
```

FORCE:

```
1629  PHONY += FORCE
1630  FORCE:
1631
```

FORCE 作为一个没有依赖也没有任何行为的伪目标，每次编译中总是被更新，所以任何依赖 FORCE 的目标，在每次执行 make 的时候也将会因为其所依赖的 FORCE‘更新’而被重新执行。

outputmakefile:

```
403  PHONY += outputmakefile
404  # outputmakefile generates a Makefile in the output directory, if using a
405  # separate output directory. This allows convenient use of make in the
406  # output directory.
407  outputmakefile:
408  ifneq ($(KBUILD_SRC),)
409      $(Q)ln -fsn $(srctree) source
410      $(Q)$(CONFIG_SHELL) $(srctree)/scripts/mkmakefile \
411          $(srctree) $(objtree) $(VERSION) $(PATCHLEVEL)
412  endif
```

根据 KBUILD_SRC 的定义：

```
103  # kbuild supports saving output files in a separate directory.
104  # To locate output files in a separate directory two syntaxes are supported.
105  # In both cases the working directory must be the root of the kernel src.
106  # 1) O=
107  # Use "make O=dir/to/store/output/files/"
108  #
109  # 2) Set KBUILD_OUTPUT
110  # Set the environment variable KBUILD_OUTPUT to point to the directory
111  # where the output files shall be placed.
112  # export KBUILD_OUTPUT=dir/to/store/output/files/
113  # make
114  #
115  # The O= assignment takes precedence over the KBUILD_OUTPUT environment
116  # variable.
117
118  # KBUILD_SRC is set on invocation of make in OBJ directory
119  # KBUILD_SRC is not intended to be used by the regular user (for now)
120  ifeq ($(KBUILD_SRC),)
121
122  # OK, Make called in directory where kernel src resides
123  # Do we want to locate output files in a separate directory?
124  ifeq ("$(origin O)", "command line")
125    KBUILD_OUTPUT := $(O)
126  endif
```

如果在编译时输入了 make O=xxx, 那么 KBUILD_OUTPUT 就会被赋值为 xxx，最终导致 outputmakefile 具有实际执行的内容。

首先 srctree 切换执行目录：

```
135  ifneq ($(KBUILD_OUTPUT),)
136  # Invoke a second make in the output directory, passing relevant variables
137  # check that the output directory actually exists
138  saved-output := $(KBUILD_OUTPUT)
139  KBUILD_OUTPUT := $(shell mkdir -p $(KBUILD_OUTPUT) && cd $(KBUILD_OUTPUT) \
140                                  && /bin/pwd)
141  $(if $(KBUILD_OUTPUT),, \
142       $(error failed to create output directory "$(saved-output)"))
143
144  PHONY += $(MAKECMDGOALS) sub-make
145
146  $(filter-out _all sub-make $(CURDIR)/Makefile, $(MAKECMDGOALS)) _all: sub-make
```

并且对 srctree 赋值：

```
203  ifeq ($(KBUILD_SRC),)
204          # building in the source tree
205          srctree := .
206  else
207          ifeq ($(KBUILD_SRC)/,$(dir $(CURDIR)))
208                  # building in a subdirectory of the source tree
209                  srctree := ..
210          else
211                  srctree := $(KBUILD_SRC)
212          endif
213  endif
214  objtree     := .
215  src         := $(srctree)
216  obj         := $(objtree)
217
```

ln -fsn $(srctree) source 将会被之行为"ln -fsn .. source"， 意味着再 xxx 目录下创建了一个 uboot 根目录的软链接。

最后执行 scripts 下的 mkmakefile, 将再 xxx 目录下生成一个 makefile。

最后一个依赖是 scripts_basic:

```
394  # Basic helpers built in scripts/
395  PHONY += scripts_basic
396  scripts_basic:
397      $(Q)$(MAKE) $(build)=scripts/basic
398      $(Q)rm -f .tmp_quiet_recordmcount
399
```

MAKE 即 make 命令， $(build)来自于 Kbuild.include 并在 makefile 中被引入

```
327  # We need some generic definitions (do not try to remake the file).
328  scripts/Kbuild.include: ;
329  include scripts/Kbuild.include
330
```

```
177  ###
178  # Shorthand for $(Q)$(MAKE) -f scripts/Makefile.build obj=
179  # Usage:
180  # $(Q)$(MAKE) $(build)=dir
181  build := -f $(srctree)/scripts/Makefile.build obj
182
```

```
$(Q)$(MAKE) $(build)=scripts/basic
```

就可以转化为：@make -f ./scripts/Makefile.build obj=scripts/basic

也就是执行 scripts/Makefile.build, 传入 obj=scripts/basic

在 Makefile.build 中 parser $(src), 得到 src=scripts/basic

```
 8  # Modified for U-Boot
 9  prefix := tpl
10  src := $(patsubst $(prefix)/%,%,$(obj))
11  ifeq ($(obj),$(src))
12  prefix := spl
13  src := $(patsubst $(prefix)/%,%,$(obj))
14  ifeq ($(obj),$(src))
15  prefix := .
16  endif
17  endif
```

之后根据 src(scripts/basic)下内容执行 include：

```
57  # The filename Kbuild has precedence over Makefile
58  kbuild-dir := $(if $(filter /%,$(src)),$(src),$(srctree)/$(src))
59  kbuild-file := $(if $(wildcard $(kbuild-dir)/Kbuild),$(kbuild-dir)/Kbuild,$(kbuild-dir)
60  include $(kbuild-file)
61
```

此处引入 scripts/basic/Makefile:

```
13
14  hostprogs-y := fixdep
15  always      := $(hostprogs-y)
16
17  # fixdep is needed to compile other host programs
18  $(addprefix $(obj)/,$(filter-out fixdep,$(always))): $(obj)/fixdep
19
```

在 Makefile.build 中，执行默认目标（__build 在 Makefile.build 中有两处，第一处在文件的最开始处，并没有提供任何依赖和执行的目标，只是占用了默认目标这一个作用，第二处的__build 则提供了最终的依赖和执行目标:

```
116 # We keep a list of all modules in $(MODVERDIR)
117 $(info ">>params1=$(KBUILD_BUILTIN),$(builtin-target),$(lib-target),$(extra-y);")
118 $(info ">>params2=$(KBUILD_MODULES),$(obj-m),$(modorder-target);")
119 $(info ">>params3=$(subdir-ym),$(always);")
120 __build: $(if $(KBUILD_BUILTIN),$(builtin-target) $(lib-target) $(extra-y)) \
121      $(if $(KBUILD_MODULES),$(obj-m) $(modorder-target)) \
122      $(subdir-ym) $(always)
123     @:
124
```

```
">>params1=1,,,;"
">>params2=,,scripts/basic/modules.order;"
">>params3=,scripts/basic/fixdep;"
```

即_build 最终依赖为 scripts/basic/fixdep

scripts/basic 下只有 fixdep.c, 目标则为 fixdep, 所以使用 Makefile.build 下的规则:

```
83
84  # Do not include host rules unless needed
85  ifneq ($(hostprogs-y)$(hostprogs-m),)
86  include scripts/Makefile.host
87  endif
```

```
88  # Create executable from a single .c file
89  # host-csingle -> Executable
90  quiet_cmd_host-csingle  = HOSTCC  $@
91        cmd_host-csingle  = $(HOSTCC) $(hostc_flags) -o $@ $< \
92          $(HOST_LOADLIBES) $(HOSTLOADLIBES_$(@F))
93  $(host-csingle): $(obj)/%: $(src)/%.c FORCE
94      $(call if_changed_dep,host-csingle)
```

执行 Kbuild.include 中定义的 if_changed_dep 并传入参数 host-csingle:

```
262  # Execute the command and also postprocess generated .d dependencies file.
263 if_changed_dep = $(if $(strip $(any-prereq) $(arg-check) ),                \
264     @set -e;                                                              \
265     $(echo-cmd) $(cmd_$(1));                                              \
266     scripts/basic/fixdep $(depfile) $@ '$(make-cmd)' > $(dot-target).tmp;\
267     rm -f $(depfile);                                                     \
268     mv -f $(dot-target).tmp $(dot-target).cmd)
```

其中 host-csingle 被展开为 cmd_host-csingle:

```
91       cmd_host-csingle  = $(HOSTCC) $(hostc_flags) -o $@ $< \
92         $(HOST_LOADLIBES) $(HOSTLOADLIBES_$(@F))
```

编译可执行目标文件 fixdep 并执行 fixdep。fixdep 最终用于更新目标（如*.o）所需要的依赖文件（*.cmd）.

执行完此步后，完成%config 所有依赖的准备，执行目标:

```
476  %config: scripts_basic outputmakefile FORCE
477      $(Q)$(MAKE) $(build)=scripts/kconfig $@
```

类似于前面，会被转换为实际命令:

make -f ./scripts/Makefile.build obj=scripts/kconfig smdk2410_defconfig

继续在 Makefile.build 中执行:

```
56  # The filename Kbuild has precedence over Makefile
57  kbuild-dir := $(if $(filter /%,$(src)),$(src),$(srctree)/$(src))
58  kbuild-file := $(if $(wildcard $(kbuild-dir)/Kbuild),$(kbuild-dir)/Kbuild,$(kbuild-dir)/Mak
59  include $(kbuild-file)
```

即相当于执行了了 include scripts/kconfig/Makefile

而 scripts/kconfig/Makefile 中定义了目标:

```
113  %_defconfig: $(obj)/conf
114      $(Q)$< $(silent) --defconfig=arch/$(SRCARCH)/configs/$@ $(Kconfig)
115
116  # Added for U-Boot (backward compatibility)
117  %_config: %_defconfig
118      @:
```

(所以 make smdk2410_config 和 make smdk2410_defconfig 效果是完全一致的)

而这个目标的依赖则是$(obj)/conf。在这个 Makefile 中，并没有显式的声明目标 $(obj)/conf, 在 scripts/kconfig/Makefile 中定义了:

```
conf-objs   := conf.o  zconf.tab.o
mconf-objs      := mconf.o zconf.tab.o $(lxdialog)
nconf-objs      := nconf.o zconf.tab.o nconf.gui.o
kxgettext-objs  := kxgettext.o zconf.tab.o
qconf-cxxobjs   := qconf.o
qconf-objs  := zconf.tab.o
gconf-objs  := gconf.o zconf.tab.o

hostprogs-y := conf nconf mconf kxgettext qconf gconf
```

而在 scripts/Makefile.build 中，在 include $(kbuild-file)之后，由于 hostprogs-y 不为空：

```
# Do not include host rules unless needed
ifneq ($(hostprogs-y)$(hostprogs-m),)
include scripts/Makefile.host
endif
```

Makefile.host 则会把 hostprogs-y 内容赋值给__hostprogs 并进一步传入 host-cmulti 中，host-cmulti 与前面分析 fixdep 中的 host-csingle 的区别就是， host-csingle 是单个.c 生成目标， host-cmulti 则是多个.c 生成目标：

```
__hostprogs := $(sort $(hostprogs-y) $(hostprogs-m))

# C code
# Executables compiled from a single .c file
host-csingle    := $(foreach m,$(__hostprogs), \
            $(if $($(m)-objs)$($(m)-cxxobjs),,$(m)))

# C executables linked based on several .o files
host-cmulti := $(foreach m,$(__hostprogs),\
            $(if $($(m)-cxxobjs),,$(if $($(m)-objs),$(m))))
```

这里通过判断没有定义 xxx-cxxobjs 且定义了 xxx-objs 来控制是否传入 host-cmulti 中，在 scripts/kconfig/Makefile 中，前面分析过，定义了：

```
conf-objs   := conf.o  zconf.tab.o
mconf-objs      := mconf.o zconf.tab.o $(lxdialog)
nconf-objs      := nconf.o zconf.tab.o nconf.gui.o
kxgettext-objs  := kxgettext.o zconf.tab.o
qconf-cxxobjs   := qconf.o
qconf-objs  := zconf.tab.o
gconf-objs  := gconf.o zconf.tab.o

hostprogs-y := conf nconf mconf kxgettext qconf gconf
```

所以最后 host-cmulti = conf nconf mconf kxgettext gconf(实际中因为执行了$(sort), 目标的顺序需要排序一下)

在 Makefile.host 中，定义了：

```
96  # Link an executable based on list of .o files, all plain c
97  # host-cmulti -> executable
98  quiet_cmd_host-cmulti   = HOSTLD  $@
99        cmd_host-cmulti   = $(HOSTCC) $(HOSTLDFLAGS) -o $@ \
100               $(addprefix $(obj)/, $($(@F)-objs)) \
101               $(HOST_LOADLIBES) $(HOSTLOADLIBES_$(@F))
102 $(host-cmulti): FORCE
103     $(call if_changed,host-cmulti)
104 $(call multi_depend, $(host-cmulti), , -objs)
```

这里和前面分析的 host-csingle 类似但又有所不同，这里 cmd_host-cmulti 实际上是链接过程，将多个.o 链接为最终的可执行文件（host-cmulti 下就是通过 xxx-objs 的定义找到对应的.o 列表），例如 conf, 在 scripts/kconfig/Makefile 下：

```
182  conf-objs    := conf.o  zconf.tab.o
```

则最后的 cmd_host-cmulti 则实际执行的：cc -o conf conf.o zconf.tab.o

但整个链条中，通过 hostprogs-y 定义了 conf nconf mconf， 并告诉了 Makefile conf-objs=conf.o zconf.tab.o, 但是%_defconfig 所真正依赖的目标$(obj)/conf， 也就是 scripts/kconfig/conf: xxx 这样的目标仍然是缺失的。

这个关键的地方则是：

```
$(call multi_depend, $(host-cxxmulti), , -objs -cxxobjs)
```

multi_depend 是在 scripts/Makefile.lib 中定义的：

```
177 # Useful for describing the dependency of composite objects
178 # Usage:
179 #   $(call multi_depend, multi_used_targets, suffix_to_remove, suffix_to_add)
180 define multi_depend
181 $(foreach m, $(notdir $1), \
182     $(eval $(obj)/$m: \
183     $(addprefix $(obj)/, $(foreach s, $3, $($(m:%$(strip $2)=%$(s))))))
184 endef
```

Makefile 中$(eval)函数的作用就是将表达式展开并在 Makefile 中执行，可以理解为 Makefile 中动态展开的一小段 Makefile。

展开后，对于每一个 host-cmulti 下定义的目标，都有：

```
$(eval $(obj)/$m:$(addprefix $(obj)/, $($(m:%$(strip  )=%-objs))))
```

以 host-cmulti 下的 conf 为例，其展开结果为：

scripts/kconfig/conf: scripts/kconfig/conf.o scripts/kconfig/zconf.tab.o

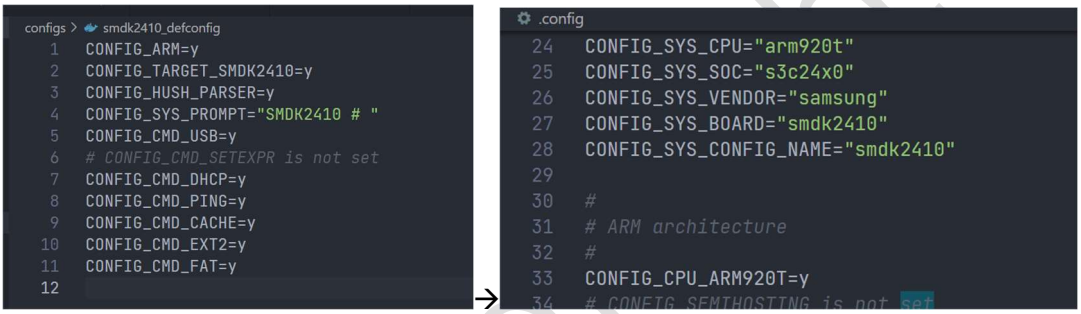
也就构造了%_defconfig 的所依赖的目标$(obj)/conf， 而对于每一个.o, 就会重新引入 host-csingle 进行编译，讲对应的.c 编译为.o:

```
  cc -Wp,-MD,scripts/kconfig/.conf.o.d -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer   -D_DEFAULT_SOURCE -D_XOPEN_S
OURCE=600 -DCURSES_LOC="<ncurses.h>" -DNCURSES_WIDECHAR=1 -DLOCALE   -c -o scripts/kconfig/conf.o scripts/kconfig/conf.c
  cat scripts/kconfig/zconf.tab.c_shipped > scripts/kconfig/zconf.tab.c
  cat scripts/kconfig/zconf.lex.c_shipped > scripts/kconfig/zconf.lex.c
  cat scripts/kconfig/zconf.hash.c_shipped > scripts/kconfig/zconf.hash.c
  cc -Wp,-MD,scripts/kconfig/.zconf.tab.o.d -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer   -D_DEFAULT_SOURCE -D_XO
PEN_SOURCE=600 -DCURSES_LOC="<ncurses.h>" -DNCURSES_WIDECHAR=1 -DLOCALE  -Iscripts/kconfig -c -o scripts/kconfig/zconf.t
ab.o scripts/kconfig/zconf.tab.c
In file included from scripts/kconfig/zconf.tab.c:2534:
```

生成 conf 后执行_defconfig 下的目标:

```
scripts/kconfig/conf  --defconfig=arch/../configs/smdk2410_defconfig Kconfig
```

smde2410_defconfig 结合 Kconfig 从而生成具体的.config 文件:



所以整体执行 make xx_defconfig 流程就是生成 fixdep->生成 conf->执行 conf configs/xx_defconfig Kconfig

make

编译时，输入命令:

make ARCH=arm CROSS_COMPILE=arm-linux-

随后系统自动完成编译链接，并产生最终的目标文件 u-boot.bin。在开始分析编译流程中，首先看编译过程中各个要素。

参与编译的要素

首先，在顶层的 makefile 中将会执行默认的目标 all:

```
128 # That's our default target when none is given on the command line
129 PHONY := _all
130 _all:
131
```

```
194  # If building an external module we do not care about the all: rule
195  # but instead _all depend on modules
196  PHONY += all
197  ifeq ($(KBUILD_EXTMOD),)
198  _all: all
199  else
200  _all: modules
201  endif
```

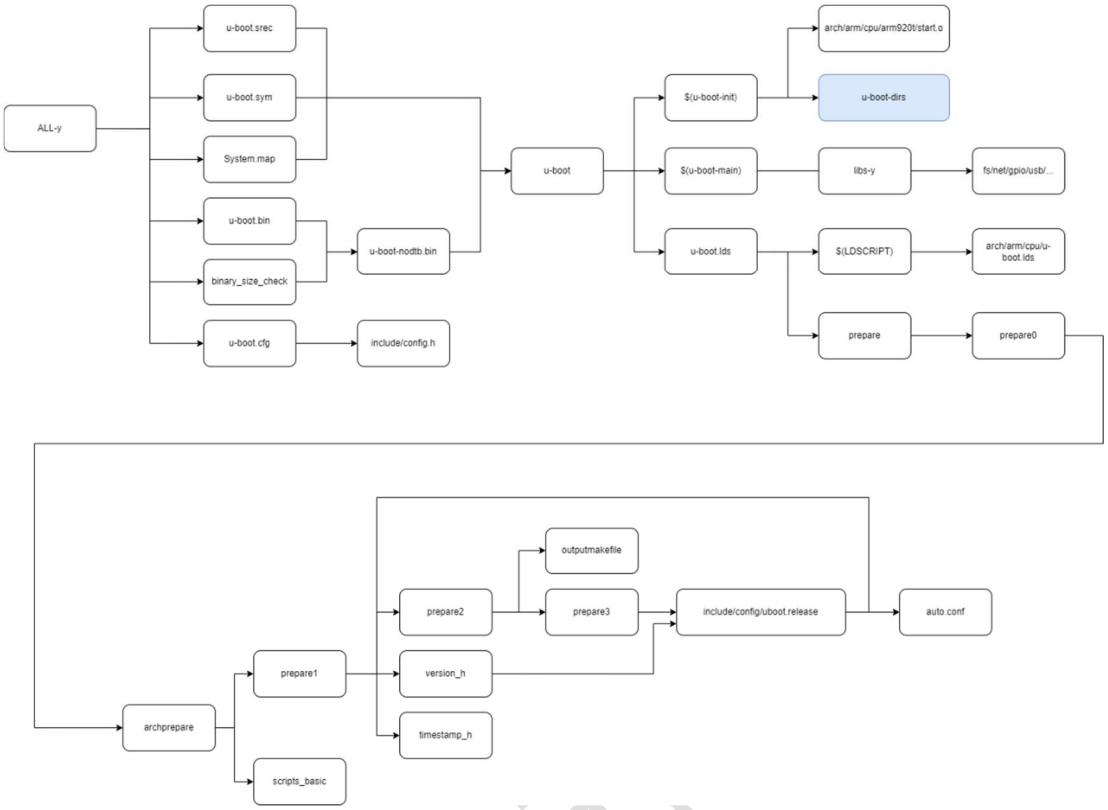
而 all 的依赖和目标：

```
805  all:        $(ALL-y)
806  ifneq ($(CONFIG_SYS_GENERIC_BOARD),y)
807      @echo "==================== WARNING ======================"
808      @echo "Please convert this board to generic board."
809      @echo "Otherwise it will be removed by the end of 2014."
810      @echo "See doc/README.generic-board for further information"
811      @echo "==================================================="
812  endif
813  ifeq ($(CONFIG_DM_I2C_COMPAT),y)
814      @echo "==================== WARNING ======================"
815      @echo "This board uses CONFIG_DM_I2C_COMPAT. Please remove"
816      @echo "(possibly in a subsequent patch in your series)"
817      @echo "before sending patches to the mailing list."
818      @echo "==================================================="
819  endif
```

目标 ALL-y:

```
733  # Always append ALL so that arch config.mk's can add custom ones
734  ALL-y += u-boot.srec u-boot.bin u-boot.sym System.map u-boot.cfg binary_size_check
735
736  ALL-$(CONFIG_ONENAND_U_BOOT) += u-boot-onenand.bin
737  ifeq ($(CONFIG_SPL_FSL_PBL),y)
738  ALL-$(CONFIG_RAMBOOT_PBL) += u-boot-with-spl-pbl.bin
739  else
740  ifneq ($(CONFIG_SECURE_BOOT), y)
741  # For Secure Boot The Image needs to be signed and Header must also
742  # be included. So The image has to be built explicitly
743  ALL-$(CONFIG_RAMBOOT_PBL) += u-boot.pbl
744  endif
745  endif
746  ALL-$(CONFIG_SPL) += spl/u-boot-spl.bin
747  ALL-$(CONFIG_SPL_FRAMEWORK) += u-boot.img
748  ALL-$(CONFIG_TPL) += tpl/u-boot-tpl.bin
749  ALL-$(CONFIG_OF_SEPARATE) += u-boot.dtb
750  ifeq ($(CONFIG_SPL_FRAMEWORK),y)
751  ALL-$(CONFIG_OF_SEPARATE) += u-boot-dtb.img
752  endif
753  ALL-$(CONFIG_OF_HOSTFILE) += u-boot.dtb
754  ifneq ($(CONFIG_SPL_TARGET),)
755  ALL-$(CONFIG_SPL) += $(CONFIG_SPL_TARGET:"%"=%)
756  endif
757  ALL-$(CONFIG_REMAKE_ELF) += u-boot.elf
758  ALL-$(CONFIG_EFI_APP) += u-boot-app.efi
759  ALL-$(CONFIG_EFI_STUB) += u-boot-payload.efi
```

这里只分析没有任何编译条件的第一行目标，可以得到依赖关系：



这个依赖关系中，u-boot-dirs 提供了一个对于各种子目录的方法：

```
1208  # Handle descending into subdirectories listed in $(vmlinux-dirs)
1209  # Preset locale variables to speed up the build process. Limit locale
1210  # tweaks to this spot to avoid wrong language settings when running
1211  # make menuconfig etc.
1212  # Error messages still appears in the original language
1213
1214  PHONY += $(u-boot-dirs)
1215  $(u-boot-dirs): prepare scripts
1216      $(Q)$(MAKE) $(build)=$@
1217
```

对于子目录（例如 xxx）展开命令就是 make -f $(srctree)/scripts/Makefile.build obj=xxx

在 Makefile.build 下构造了子目录下的 Makefile 所在路径，并将他们 include 进当前 makefile

```
56  # The filename Kbuild has precedence over Makefile
57  kbuild-dir := $(if $(filter /%,$(src)),$(src),$(srctree)/$(src))
58  kbuild-file := $(if $(wildcard $(kbuild-dir)/Kbuild),$(kbuild-dir)/Kbuild,$(kbuild-dir)/Makefile
59  include $(kbuild-file)
60
```

以 fs/yaffs2 为例子此处得到的 kbuild-file 就为 fs/Makefile:

```
#

ifdef CONFIG_SPL_BUILD
obj-$(CONFIG_SPL_FAT_SUPPORT) += fat/
obj-$(CONFIG_SPL_EXT_SUPPORT) += ext4/
else
obj-y                 += fs.o

obj-$(CONFIG_CMD_CBFS) += cbfs/
obj-$(CONFIG_CMD_CRAMFS) += cramfs/
obj-$(CONFIG_FS_EXT4) += ext4/
obj-y += fat/
obj-$(CONFIG_CMD_JFFS2) += jffs2/
obj-$(CONFIG_CMD_REISER) += reiserfs/
obj-$(CONFIG_SANDBOX) += sandbox/
obj-$(CONFIG_CMD_UBIFS) += ubifs/
obj-$(CONFIG_YAFFS2) += yaffs2/
obj-$(CONFIG_CMD_ZFS) += zfs/
endif

```

回到 Makefile.build, 构造了对于这个子目录的 builtin-target:

```
ifneq ($(strip $(obj-y) $(obj-m) $(obj-) $(subdir-m) $(lib-target)),)
builtin-target := $(obj)/built-in.o
endif

```

同时这个文件也定义了规则生成:

```
#
# Rule to compile a set of .o files into one .o file
#
ifdef builtin-target
quiet_cmd_link_o_target = LD      $@
# If the list of objects to link is empty, just create an empty built-in.o
cmd_link_o_target = $(if $(strip $(obj-y)),\
                $(LD) $(ld_flags) -r -o $@ $(filter $(obj-y), $^) \
                $(cmd_secanalysis),\
                rm -f $@; $(AR) rcs$(KBUILD_ARFLAGS) $@)

$(builtin-target): $(obj-y) FORCE
    $(call if_changed,link_o_target)

targets += $(builtin-target)
endif # builtin-target

```

最终会生成 fs 下的.built-in.o.cmd:

```
cmd_fs/built-in.o :=  arm-linux-ld      -r -o fs/built-in.o fs/fs.o fs/ext4/built-in.o fs/fat/built-in.o fs/ubifs/built-in.o fs/yaffs2/built-in.o
```

而在顶层 Makefile 中，同时将 libs-y 下的定义的路径添加目标 built-in.o:

```
676
677 libs-y      := $(patsubst %/, %/built-in.o, $(libs-y))
678
679 u-boot-init := $(head-y)
680 u-boot-main := $(libs-y)
```

而针对 obj-y 中的.o 则提供了编译的规则:

```
279
280 # Built-in and composite module parts
281 $(obj)/%.o: $(src)/%.c $(recordmcount_source) FORCE
282     $(call cmd,force_checksrc)
283     $(call if_changed_rule,cc_o_c)
284
```

最终的链接:

```
1327 # ---------------------------------------------------------------------------
1328 quiet_cmd_cpp_lds = LDS     $@
1329 cmd_cpp_lds = $(CPP) -Wp,-MD,$(depfile) $(cpp_flags) $(LDPPFLAGS) -ansi \
1330         -D__ASSEMBLY__ -x assembler-with-cpp -P -o $@ $<
1331
1332 u-boot.lds: $(LDSCRIPT) prepare FORCE
1333     $(call if_changed_dep,cpp_lds)
1334
```

所以对于 u-boot 的宏观上的编译过程就是根据 makefile 的一层一层的索引，找到所有激活的子目录下的源代码文件编译成.o, 并通过.build-in.o.cmd 而链接为更大的 built-in.o, 最后由链接器和 u-boot.lds 链接脚本链接为最终的二进制文件。

核心编译流程

根据前面的分析，u-boot 核心的编译围绕着 u-boot 这个目标展开:

```
1192 u-boot: $(u-boot-init) $(u-boot-main) u-boot.lds FORCE
1193     $(call if_changed,u-boot__)
1194 ifeq ($(CONFIG_KALLSYMS),y)
1195     $(call cmd,smap)
1196     $(call cmd,u-boot__) common/system_map.o
1197 endif
```

u-boot-main 其实就是各个子模块的合集:

```
677 libs-y      := $(patsubst %/, %/built-in.o, $(libs-y))
678
679 u-boot-init := $(head-y)
680 u-boot-main := $(libs-y)
```

```
>> u-boot-main= arch/arm/cpu/built-in.o  arch/arm/cpu/arm920t/built-in.o  arch/arm/lib/built-in.o  board/s
amsung/common/built-in.o  board/samsung/smdk2410/built-in.o  cmd/built-in.o  common/built-in.o  disk/built
-in.o  drivers/built-in.o  drivers/dma/built-in.o  drivers/gpio/built-in.o  drivers/i2c/built-in.o  driver
s/mmc/built-in.o  drivers/mtd/built-in.o  drivers/mtd/nand/built-in.o  drivers/mtd/onenand/built-in.o  dri
vers/mtd/spi/built-in.o  drivers/mtd/ubi/built-in.o  drivers/net/built-in.o  drivers/net/phy/built-in.o  d
rivers/pci/built-in.o  drivers/power/built-in.o  drivers/power/battery/built-in.o  drivers/power/fuel_gaug
e/built-in.o  drivers/power/mfd/built-in.o  drivers/power/pmic/built-in.o  drivers/power/regulator/built-i
n.o  drivers/serial/built-in.o  drivers/spi/built-in.o  drivers/usb/common/built-in.o  drivers/usb/dwc3/bu
ilt-in.o  drivers/usb/emul/built-in.o  drivers/usb/eth/built-in.o  drivers/usb/gadget/built-in.o  drivers/
usb/gadget/udc/built-in.o  drivers/usb/host/built-in.o  drivers/usb/musb-new/built-in.o  drivers/usb/musb/
built-in.o  drivers/usb/phy/built-in.o  drivers/usb/ulpi/built-in.o  fs/built-in.o  lib/built-in.o  net/bu
ilt-in.o  test/built-in.o  test/dm/built-in.o
```

在将 libs-y 加上后缀 built-in.o 之前，同时将这些路径保存在了 u-boot-dirs 内，并增加了 tools 和 examples 两个路径：

```
673  u-boot-dirs := $(patsubst %/,%,$(filter %/, $(libs-y))) tools examples
674
675  u-boot-alldirs  := $(sort $(u-boot-dirs) $(patsubst %/,%,$(filter %/, $(libs-))))
676
677  libs-y      := $(patsubst %/, %/built-in.o, $(libs-y))
```

```
>> u-boot-dirs=arch/arm/cpu arch/arm/cpu/arm920t arch/arm/lib board/samsung/common board/samsung/smdk2410
cmd common disk drivers drivers/dma drivers/gpio drivers/i2c drivers/mmc drivers/mtd drivers/mtd/nand driv
ers/mtd/onenand drivers/mtd/spi drivers/mtd/ubi drivers/net drivers/net/phy drivers/pci drivers/power driv
ers/power/battery drivers/power/fuel_gauge drivers/power/mfd drivers/power/pmic drivers/power/regulator dr
ivers/serial drivers/spi drivers/usb/common drivers/usb/dwc3 drivers/usb/emul drivers/usb/eth drivers/usb/
gadget drivers/usb/gadget/udc drivers/usb/host drivers/usb/musb-new drivers/usb/musb drivers/usb/phy drive
rs/usb/ulpi fs lib net test test/dm tools examples
```

u-boot-dirs 作为一个伪目标，提供了具体的编译行为：

```
1210  # Handle descending into subdirectories listed in $(vmlinux-dirs)
1211  # Preset locale variables to speed up the build process. Limit locale
1212  # tweaks to this spot to avoid wrong language settings when running
1213  # make menuconfig etc.
1214  # Error messages still appears in the original language
1215
1216  PHONY += $(u-boot-dirs)
1217  $(u-boot-dirs): prepare scripts
1218      $(Q)$(MAKE) $(build)=$@
1219
```

例如执行：make -f ./scripts/Makefile.build obj=arch/arm/cpu/arm920t，产生 built-in.o

再生成完各个子模块的 built-in.o 后，则执行最终的链接过程：

```
1178  # Rule to link u-boot
1179  # May be overridden by arch/$(ARCH)/config.mk
1180  quiet_cmd_u-boot__ ?= LD      $@
1181        cmd_u-boot__ ?= $(LD) $(LDFLAGS) $(LDFLAGS_u-boot) -o $@ \
1182        -T u-boot.lds $(u-boot-init)                              \
1183        --start-group $(u-boot-main) --end-group                  \
1184        $(PLATFORM_LIBS) -Map u-boot.map
1185
1186  quiet_cmd_smap = GEN     common/system_map.o
1187  cmd_smap = \
1188      smap=`$(call SYSTEM_MAP,u-boot) | \
1189          awk '$$2 ~ /[tTwW]/ {printf $$1 $$3 "\\\\000"}'` ; \
1190      $(CC) $(c_flags) -DSYSTEM_MAP="\"$${smap}\"" \
1191          -c $(srctree)/common/system_map.c -o common/system_map.o
1192
1193  u-boot: $(u-boot-init) $(u-boot-main) u-boot.lds FORCE
1194      $(call if_changed,u-boot__)
1195  ifeq ($(CONFIG_KALLSYMS),y)
1196      $(call cmd,smap)
1197      $(call cmd,u-boot__) common/system_map.o
1198  endif
```

完成链接后则通过 objcopy 生成最终的 u-boot.bin:

```
867
868  u-boot-nodtb.bin: u-boot FORCE
869      $(call if_changed,objcopy)
870      $(call DO_STATIC_RELA,$<,$@,$(CONFIG_SYS_TEXT_BASE))
871      $(BOARD_SIZE_CHECK)
872
```

U-Boot 启动

启动宏观流程

从上电的零时刻起，到跳转到 Linux 内核启动，大体的流程为芯片 ROM 代码执行，完成芯片最初始的启动，根据芯片的硬件（例如启动引脚）的配置，初始化关键硬件和跳转到对应外设地址执行，例如跳转到 NorFlash 执行。

在 NorFlash 中，则固化了 U-Boot 的 binary, U-Boot 开始执行，U-Boot 的执行可以大致分为四个阶段：

1. CPU 最底层早期初始化和初始化 C 语言环境（堆栈的初始化）
2. board_init_f，板级早期初始化，
3. relocate 和在 SDRAM 中继续执行 board_init_r 完成全部板级初始化
4. 命令的执行和内核启动

在 U-Boot 最后的执行阶段，加载完内核后则跳转到内核的入口地址处，开始执行内核的启动。

U-Boot 启动代码详细分析

早期启动

查看链接脚本 u-boot.lds:

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
 . = 0x00000000;
 . = ALIGN(4);
 .text :
 {
  *(.__image_copy_start)
  *(.vectors)
  arch/arm/cpu/arm920t/start.o (.text*)
  *(.text*)
 }
```

可知整个程序的入口是_start, 位于 arch/arm/lib/vectors.S, 同时也是中断向量表的位置:

```

_start:

#ifdef CONFIG_SYS_DV_NOR_BOOT_CFG
    .word   CONFIG_SYS_DV_NOR_BOOT_CFG
#endif

    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
```

执行的第一条指令便是跳转到 reset 函数，位于 arch/arm/cpu/arm920t/start.S

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

```

设置 CPSR 寄存器为 8'b11x10011，对应含义:

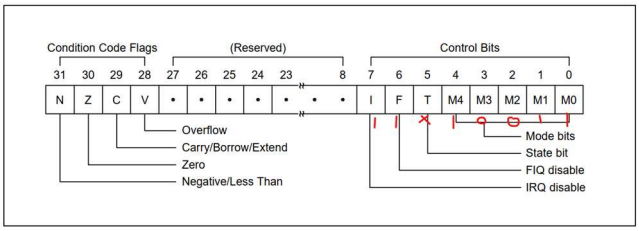


Figure 2-6. Program Status Register Format

关闭中断（IRQ+FIQ），让处理器工作在 Supervisor 模式：

Table 2-1. PSR Mode Bit Values

M[4:0]	Mode	Visible THUMB state registers	Visible ARM state registers
10000	User	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR
10001	FIQ	R7..R0, LR_fiq, SP_fiq PC, CPSR, SPSR_fiq	R7..R0, R14_fiq..R8_fiq, PC, CPSR, SPSR_fiq
10010	IRQ	R7..R0, LR_irq, SP_irq PC, CPSR, SPSR_irq	R12..R0, R14_irq, R13_irq, PC, CPSR, SPSR_irq
10011	Supervisor	R7..R0, LR_svc, SP_svc, PC, CPSR, SPSR_svc	R12..R0, R14_svc, R13_svc, PC, CPSR, SPSR_svc
10111	Abort	R7..R0, LR_abt, SP_abt, PC, CPSR, SPSR_abt	R12..R0, R14_abt, R13_abt, PC, CPSR, SPSR_abt
11011	Undefined	R7..R0 LR_und, SP_und, PC, CPSR, SPSR_und	R12..R0, R14_und, R13_und, PC, CPSR
11111	System	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR

之后执行代码：

```
52
53  #ifdef CONFIG_S3C24X0
54      /* turn off the watchdog */
55
56  # if defined(CONFIG_S3C2400)
57  #  define pWTCON     0x15300000
58  #  define INTMSK     0x14400008  /* Interrupt-Controller base addresses */
59  #  define CLKDIVN    0x14800014  /* clock divisor register */
60  #else
61  #  define pWTCON     0x53000000
62  #  define INTMSK     0x4A000008  /* Interrupt-Controller base addresses */
63  #  define INTSUBMSK  0x4A00001C
64  #  define CLKDIVN    0x4C000014  /* clock divisor register */
65  # endif
66
```

smdk2410.h 中定义了 CONFIG_S3C24X0 但是没有定义 CONFIG_S3C2400, 故 pWTCON 定义为 0x53000000, 也就是 2410 中看门狗的地址。关闭看门狗，放置系统因为后续的操作没有喂狗动作而导致的复位：

```
67      ldr r0, =pWTCON
68      mov r1, #0x0
69      str r1, [r0]
```

随后执行代码关闭中断：

```
71      /*
72       * mask all IRQs by setting all bits in the INTMR - default
73       */
74      mov r1, #0xffffffff
75      ldr r0, =INTMSK
76      str r1, [r0]
77  # if defined(CONFIG_S3C2410)
78      ldr r1, =0x3ff
79      ldr r0, =INTSUBMSK
80      str r1, [r0]
81  # endif
```

令人不太理解的一点是 2410 下，INTSUBMSK 的 BIT10，也就是 ADC 对应位被清除了：

Register	Address	R/W	Description	Reset Value
INTSUBMSK	0X4A00001C	R/W	Determine which interrupt source is masked. The masked interrupt source will not be serviced. 0 = Interrupt service is available. 1 = Interrupt service is masked.	0x7FF

INTSUBMSK	Bit	Description	Initial State
Reserved	[31:11]	Not used	0
INT_ADC	[10]	0 = Service available, 1 = Masked	1
INT_TC	[9]	0 = Service available, 1 = Masked	1
INT_ERR2	[8]	0 = Service available, 1 = Masked	1

继续执行代码：

```
83      /* FCLK:HCLK:PCLK = 1:2:4 */
84      /* default FCLK is 120 MHz ! */
85      ldr r0, =CLKDIVN
86      mov r1, #3
87      str r1, [r0]
```

根据初始状态下系统的时钟配置：

PLLCON	Bit	Description	Initial State
MDIV	[19:12]	Main divider control	0x5C / 0x28
PDIV	[9:4]	Pre-divider control	0x08 / 0x08
SDIV	[1:0]	Post divider control	0x0 / 0x0

根据公式：

PLL Control Register (MPLLCON and UPLLCON)

Mpll = (m * Fin) / (p * 2^s)
m = (MDIV + 8), p = (PDIV + 2), s = SDIV

可以计算得到 Mpll = ((92+8) * 12MHz)/((8+2)*2^0) = 120MHz

配置 CLKDIVN 寄存器让 F:H:P 频率比为 1：2：4， 也就是 HCLK=60MHz, PCLK=30MHz.

继续执行代码：

```
94   #ifndef CONFIG_SKIP_LOWLEVEL_INIT
95       bl  cpu_init_crit
96   #endif
97
```

执行：

```
120  cpu_init_crit:
121      /*
122       * flush v4 I/D caches
123       */
124      mov r0, #0
125      mcr p15, 0, r0, c7, c7, 0    /* flush v3/v4 cache */
126      mcr p15, 0, r0, c8, c7, 0    /* flush v4 TLB */
127
128      /*
```

mcr 指令为 cp15 协处理器指令，格式大概是 mcr p15, op1, Rd, CRn, CRm, op2

在 S3C2410 中对应的指令内容为：

Invalidate ICache & DCache	SBZ	MCR p15,0,Rd,c7,c7,0
Invalidate TLB(s)	SBZ	MCR p15,0,Rd,c8,c7,0

分别清除 I/D Cache 和 MMU 中的 TLB
TLB 是 MMU 中用于缓冲页表的查找表。
继续执行关闭 MMU 操作：

```
128     /*
129      * disable MMU stuff and caches
130      */
131     mrc p15, 0, r0, c1, c0, 0
132     bic r0, r0, #0x00002300 @ clear bits 13, 9:8 (--V- --RS)
133     bic r0, r0, #0x00000087 @ clear bits 7, 2:0 (B--- -CAM)
134     orr r0, r0, #0x00000002 @ set bit 1 (A) Align
135     orr r0, r0, #0x00001000 @ set bit 12 (I) I-Cache
136     mcr p15, 0, r0, c1, c0, 0
137
```

执行：

```
143     mov ip, lr
144
145     bl  lowlevel_init
146
147     mov lr, ip
148     mov pc, lr
```

ip 寄存器就是 R12 寄存器，arm 中 R15 就是 PC，R14 是 LR，R13 是 SP，R12，这里首先备份 LR 地址到 R12 中，再执行完 lowlevel_init 函数后再将 LR 恢复并返回到 LR 的地址。lowlevel_init 属于板级的定义，在 board/samsung/smdk2410/lowlevel_init.S 中：

```
111 .globl lowlevel_init
112 lowlevel_init:
113     /* memory control configuration */
114     /* make r0 relative the current location so that it */
115     /* reads SMRDATA out of FLASH rather than memory ! */
116     ldr     r0, =SMRDATA
117     ldr r1, =CONFIG_SYS_TEXT_BASE
118     sub r0, r0, r1
119     ldr r1, =BWSCON /* Bus Width Status Controller */
120     add     r2, r0, #13*4
121 0:
122     ldr     r3, [r0], #4
123     str     r3, [r1], #4
124     cmp     r2, r0
125     bne     0b
126
127     /* everything is fine now */
128     mov pc, lr
```

前面配置 SDRAM 时序的参数就在这个文件的 SMRDATA 中修改的，这组代码的的作用就是把 SMRDATA 中的内容依次写入内存控制器相关的寄存器中：

Table 1-4. S3C2410X Special Registers (Sheet 1 of 11)

Register Name	Address (B. Endian)	Address (L. Endian)	Acc. Unit	Read/ Write	Function
Memory Controller					
BWSCON	0x48000000	←	W	R/W	Bus Width & Wait Status Control
BANKCON0	0x48000004				Boot ROM Control
BANKCON1	0x48000008				BANK1 Control
BANKCON2	0x4800000C				BANK2 Control
BANKCON3	0x48000010				BANK3 Control
BANKCON4	0x48000014				BANK4 Control
BANKCON5	0x48000018				BANK5 Control
BANKCON6	0x4800001C				BANK6 Control
BANKCON7	0x48000020				BANK7 Control
REFRESH	0x48000024				DRAM/SDRAM Refresh Control
BANKSIZE	0x48000028				Flexible Bank Size
MRSRB6	0x4800002C				Mode register set for SDRAM
MRSRB7	0x48000030				Mode register set for SDRAM

初始化 C 语言环境

配置完后函数返回至 start.S 中的 reset 代码中执行跳转：

```
98      bl  _main
```

_main 的定义是在 arch/arm/lib/crt0.S 或者 crt0_64.S 中：

```
67  ENTRY(_main)
```

根据前面分析的 U-boot 编译流程，arch/arm/lib/Makefile 将会被引用，那么：

```
12  ifdef CONFIG_CPU_V7M
13  obj-y   += vectors_m.o crt0.o
14  else ifdef CONFIG_ARM64
15  obj-y   += crt0_64.o
16  else
17  obj-y   += vectors.o crt0.o
18  endif
```

由于没有定义 CONFIG_CPU_V7M 和 CONFIG_ARM64, 所以 crt0.o 对应的 crt0.S 会被编译。_main 函数中首先执行：

```
#if defined(CONFIG_SPL_BUILD) && defined(CONFIG_SPL_STACK)
    ldr sp, =(CONFIG_SPL_STACK)
#else
    ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
#endif
#if defined(CONFIG_CPU_V7M) /* v7M forbids using SP as BIC destinat
    mov r3, sp
    bic r3, r3, #7
    mov sp, r3
#else
    bic sp, sp, #7  /* 8-byte alignment for ABI compliance */
#endif
    mov r0, sp
    bl  board_init_f_alloc_reserve
    mov sp, r0
```

初始化堆栈指针并跳转到 board_init_f_alloc_reserve(sp):

```
ulong board_init_f_alloc_reserve(ulong top)
{
    /* Reserve early malloc arena */
#if defined(CONFIG_SYS_MALLOC_F)
    top -= CONFIG_SYS_MALLOC_F_LEN;
#endif
    /* LAST : reserve GD (rounded up to a multiple of 16 bytes) */
    top = rounddown(top-sizeof(struct global_data), 16);

    return top;
}
```

rounddown 在 include/linux/kernel.h 中定义：

```
#define rounddown(x, y) (               \
{                               \
    typeof(x) __x = (x);                \
    __x - (__x % (y));              \
}                               \
)
```

在 board_init.c 开头，定义了：

```
#include <common.h>

DECLARE_GLOBAL_DATA_PTR;
```

而 common.h include arm/global_data.h:

```
57
58  #include <asm-generic/global_data.h>
59
```

```
83
84  #ifdef CONFIG_ARM64
85  #define DECLARE_GLOBAL_DATA_PTR     register volatile gd_t *gd asm ("x18")
86  #else
87  #define DECLARE_GLOBAL_DATA_PTR     register volatile gd_t *gd asm ("r9")
88  #endif
89  #endif
90
```

在 asm-generic/global_data.h 中定义了结构体 global_data:

```
24  #include <membuff.h>
25  #include <linux/list.h>
26
27  typedef struct global_data {
28      bd_t *bd;
29      unsigned long flags;
30      unsigned int baudrate;
31      unsigned long cpu_clk;  /* CPU clock in Hz!     */
32      unsigned long bus_clk;
33      /* We cannot bracket this with CONFIG_PCI due to mpc5xxx */
34      unsigned long pci_clk;
35      unsigned long mem_clk;
36  #if defined(CONFIG_LCD) || defined(CONFIG_VIDEO)
37      unsigned long fb_base;  /* Base address of framebuffer mem */
```

所以 board_init_f_alloc_reserve(sp)函数的最终作用就是根据传入的堆栈指针，在堆栈中挖出一块 global_data 大小（并保证是 16 字节填充对齐的）的区域。
并在函数返回后，修改堆栈栈顶指针的位置，并把这个值同时赋给 r9, 也就是保证指针*gd（register volatile gd_t *gd asm ("r9")）指向刚从堆栈中保留的空间：

```
86      bl  board_init_f_alloc_reserve
87      mov sp, r0
88      /* set up gd here, outside any C code */
89      mov r9, r0
90      bl  board_init_f_init_reserve
```

然后继续跳转到 void board_init_f_init_reserve(ulong base)执行：
根据 AAPCS，base 就是 r0, 也就是 board_init_f_alloc_reserve 的返回值，也是刚从栈中挖出来的 global_data 的基地址。那么 board_init_f_init_reserve，结合其名字，输入值于头部注释可知用于初始化 global_data 与相关的 reserve 的内存空间。

```
68  /*
69   * Initialize reserved space (which has been safely allocated on the C
70   * stack from the C runtime environment handling code).
71   *
72   * Notes:
73   *
74   * Actual reservation was done by the caller; the locations from base
75   * to base+size-1 (where 'size' is the value returned by the allocation
```

```
110  void board_init_f_init_reserve(ulong base)
111  {
112      struct global_data *gd_ptr;
113  #ifndef _USE_MEMCPY
114      int *ptr;
115  #endif
116
```

```
14  /*
15   * It isn't trivial to figure out whether memcpy() exists. The arch-specific
16   * memcpy() is not normally available in SPL due to code size.
17   */
18  #if !defined(CONFIG_SPL_BUILD) || \
19          (defined(CONFIG_SPL_LIBGENERIC_SUPPORT) && \
20          !defined(CONFIG_USE_ARCH_MEMSET))
21  #define _USE_MEMCPY
22  #endif
```

初始化 global_data 区为 0:

```
122      gd_ptr = (struct global_data *)base;
123      /* zero the area */
124  #ifdef _USE_MEMCPY
125      memset(gd_ptr, '\0', sizeof(*gd));
126  #else
127      for (ptr = (int *)gd_ptr; ptr < (int *)(gd_ptr + 1); )
128          *ptr++ = 0;
129  #endif
```

后面的代码:

```
134      /* next alloc will be higher by one GD plus 16-byte alignment */
135      base += roundup(sizeof(struct global_data), 16);
136
137      /*
138       * record early malloc arena start.
139       * Use gd as it is now properly set for all architectures.
140       */
141
142  #if defined(CONFIG_SYS_MALLOC_F)
143      /* go down one 'early malloc arena' */
144      gd->malloc_base = base;
145      /* next alloc will be higher by one 'early malloc arena' size */
146      base += CONFIG_SYS_MALLOC_F_LEN;
147  #endif
148  }
```

因该结合 board_init_f_alloc_reserve 中:

```
59  #if defined(CONFIG_SYS_MALLOC_F)
60      top -= CONFIG_SYS_MALLOC_F_LEN;
61  #endif
62      /* LAST : reserve GD (rounded up to
```

也就是如果定义了 CONFIG_SYS_MALLOC_F, 那么在 board_init_f_alloc_reserve 中，从栈顶首先挖出来一块 CONFIG_SYS_MALLOC_F 的空间，随后再挖出来 global_data 的保留空

间，随后在 board_init_f_init_reserve 中将这个 malloc_f 的基地址，写到 global_data->malloc_base, 这个空间应该是早期的 malloc-heap, 并没有初始化为 0 而是保持了上电后的随机状态。
此后，基本的 C 语言环境已经建立（就是初始化堆和栈的指针）。

board_init_f

返回 crt0.S 中，执行 board_init_f(0):

```
92      mov r0, #0
93      bl  board_init_f
```

由于在.config 中配置了 CONFIG_SYS_GENERIC_BOARD=y，那么在 Common/Makefile 下:

```
28  # boards
29  obj-$(CONFIG_SYS_GENERIC_BOARD) += board_f.o
30  obj-$(CONFIG_SYS_GENERIC_BOARD) += board_r.o
31  obj-$(CONFIG_DISPLAY_BOARDINFO) += board_info.o
32  obj-$(CONFIG_DISPLAY_BOARDINFO_LATE) += board_info.o
33
```

board_f.c 和 board_r.c 都会被编译，board_inif_f()函数就定义在 board_f.c 中:

```
1056      gd->flags = boot_flags;
1057      gd->have_console = 0;
1058
1059      if (initcall_run_list(init_sequence_f))
1060          hang();
1061
```

initcall_run_list()被定义在 initcall.c, 是 lib/Makefile 下一个默认的编译目标:

```
33  obj-y += initcall.o
34  obj-$(CONFIG_LMB) += lmb.o
```

精简后的代码:

```
13  int initcall_run_list(const init_fnc_t init_sequence[])
14  {
15      const init_fnc_t *init_fnc_ptr;
16      for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
17          unsigned long reloc_ofs = 0;
18          int ret;
19          if (gd->flags & GD_FLG_RELOC)
20              reloc_ofs = gd->reloc_off;
21          ret = (*init_fnc_ptr)();
22          if (ret) {
23              return -1;
24      }
25      return 0;
26  }
```

这个函数的作用是依次执行 init_sequence[]中的函数指针所指向的函数，直到最后一个 NULL 指针为止，执行过程中如果其中一个函数返回值不为 0 则立即退出执行并报错与返回-1。
那么 initcall_run_list(init_sequence_f)就是要执行 init_sequence_f 中的函数:

```
829  static init_fnc_t init_sequence_f[] = {
830  #ifdef CONFIG_SANDBOX
831      setup_ram_buf,
832  #endif
833      setup_mon_len,
834  #ifdef CONFIG_OF_CONTROL
835      fdtdec_setup,
836  #endif
837  #ifdef CONFIG_TRACE
838      trace_early_init,
839  #endif
840      initf_malloc,
841      initf_console_record,
842  #if defined(CONFIG_MPC85xx) || defined(CONFIG_MPC86xx)
843      /* TODO: can this go into arch_cpu_init()? */
844      probecpu,
845  #endif
846  #if defined(CONFIG_X86) && defined(CONFIG_HAVE_FSP)
847      x86_fsp_init,
848  #endif
849      arch_cpu_init,      /* basic arch cpu dependent setup */
850      initf_dm,
```

这个数组很长，只截取了一小段。

由于初始化函数太多，不再记录逐行分析，一下是 smdk2410 需要执行的函数和大概的功能分析表（针对 SMDK2410 配置）：

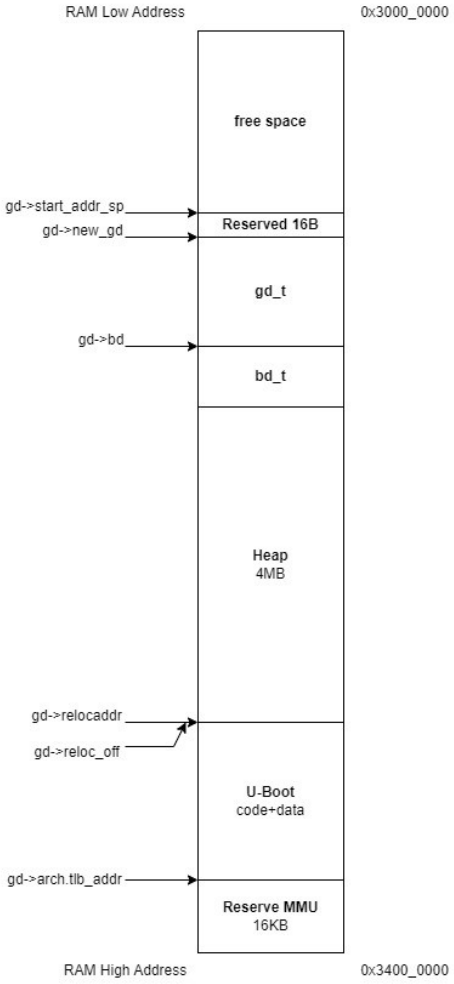
函数名	位置	说明
setup_mon_len	common/board_f.c	在 global_data->mon_len 的写入值 TEXT+RODATA+DATA+BSS 的空间大小。
initf_malloc	common/dlmalloc.c	由于没有定义 CONFIG_SYS_MALLOC_F_LEN，故此函数没有实际作用
initf_console_record	common/board_f.c	由于没有定义 CONFIG_CONSOLE_RECORD 和 CONFIG_SYS_MALLOC_F_LEN，这个函数没有实际作用。这个函数中由调用了 membuf 相关的 API，一个循环 buffer
arch_cpu_init	common/board_f.c	board_f.c 中 arch_cpu_init 被定义为 weak 函数，在 arm/cpu 下并没有具体实现，故此函数没有实际作用
initf_dm	common/board_f.c	由于没有定义 CONFIG_DM，CONFIG_SYS_MALLOC_F_LEN 和 CONFIG_TIMER_EARLY， 故此函数没有实际作用
arch_cpu_init_dm	common/board_f.c	被定义为弱函数且没有实际其他实现，故此函数没有实际作用
mark_bootstage	common/board_f.c	将 BOOTSTAGE_ID_START_UBOOT_F 和信息"board_init_f"记录到数组 struct bootstage_record record[BOOTSTAGE_ID_COUNT]中去

		这里有个地怪的地方，mark_bootstage 中最终会调用到 get_timer()这个 API，而此时 timer 还没有初始化，smdk2410 中在后面的位置才会实行 timer_init 完成定时器的初始化
board_early_init_f	board/samsung/smdk2410/smdk2410.c	初始化 FCLK 为 202.8MHz，UPLL 为 48MHz 并初始化所有 IO 口到为一组固定的值，如果是其他修改过的板子，这组数值要根据具体硬件进行修改
timer_init	arch/arm/cpu/arm920t/s3c24x0/timer.c	初始化 S3C2410 片内 Timer4，并配置为 10ms 自动重载
env_init	common/env_nand.c	此处只实际赋值 gd->env_addr = (ulong)&default_environment[0]; gd->env_valid = 1; 在前面配置过 include/configs/smdk2410.h 中配置过的 CONFIG_BOOTARGS 或者 CONFIG_BOOTCOMMAND，都是在 include/env_default.h 中被赋值给 default_environment
init_baud_rate	common/board_f.c	通过 gd->baudrate = getenv_ulong("baudrate", 10, CONFIG_BAUDRATE); 获取配置于 inlclude/configs/smdk2410.h 中，最终被 env_init 赋值到环境变量中的串口波特率
serial_init	drivers/series/serial.c	通过调用 get_current(), 进而通过 drivers/serial/serial_s3c24x0.c: default_serial_console()最终执行 serial_init_dev(0)完成串口 0 的初始化
console_init_f	common/console.c	由于没有定义 CONFIG_PRE_CONSOLE_BUFFER，此函数没有实际作用
display_options	lib/display_options.c	打印 version_string，参见 U-Boot 中的版本号和编译时间管理
display_text_info	common/board_f.c	打印 text 地址，bss 地址信息
print_cpuinfo	arch/arm/cpu/arm920t/s3c24x0/cpu_info.c	读取并显示 CPUID，并通过 arch/arm/cpu/arm920t/s3c24x0/speed.c 获取 PLL 配置信息并计算具体时钟
init_func_watchdog_init	common/board_f.c	此函数没有实际作用
init_func_watchdog_reset	common/board_f.c	此函数没有实际作用
announce_dram_init	common/board_f.c	打印"DRAM: "
dram_init	board/samsung/smdk	配置 gd->ram_size= PHYS_SDRAM_1_SIZE, smdk2410.h 中根据硬件配置修改为 64MB

	2410/smdk2410.c	
setup_dest_addr	common/board_f.c	计算并配置 gd->ram_top 和 gd->relocaddr
reserve_round_4k	common/board_f.c	修改 gd->relocaddr，确保地址时 4KB 对齐的
reserve_mmu	common/board_f.c	修改 gd_relocaddr, 默认挖掉 4KB*4 的空间预留给 MMU 的 TLB，并确保地址 64KB 对齐
reserve_trace	common/board_f.c	此函数没有实际作用 值得提到的是，在 lib/trace.c 下，提供了一个函数执行追踪库，通过使用 gcc 提供的-finstrument-functions, 每个函数的入口和出口都会额外的增加__cyg_profile_func_enter(void *func_ptr, void *caller) 和 __cyg_profile_func_exit(void *func_ptr, void *caller)，用来提供函数追踪的功能
reserve_uboot	common/board_f.c	从 SDRAM 底部预留出 UBOOT 大小的空间，并向上 4KB 对齐，并且初始化 gd->start_addr_sp
reserve_malloc	common/board_f.c	从 gd->start_addr_sp 再预留出 4MB 的空间给 malloc 使用
reserve_board	common/board_f.c	从 gd->start_addr_sp 预留 u-boot.h 下定义了板子信息的 bd_t 的空间
setup_machine	common/board_f.c	此函数没有实际作用
reserve_global_data	common/board_f.c	从 gd->start_addr_sp 中预留 global_data.h 中 gd_t 的空间
reserve_fdt	common/board_f.c	此函数没有实际作用
reserve_arch	common/board_f.c	此函数没有实际作用
reserve_stacks	common/board_f.c	从 gd->start_addr_sp 中预留 16 字节，并地址 16 字节对齐
setup_dram_config	common/board_f.c	初始化 gd->bd->bi_dram 信息
show_dram_config	common/board_f.c	打印配置的 SDRAM 信息
display_new_sp	common/board_f.c	打印最后更新的 gd->start_addr_sp 地址
reloc_fdt	common/board_f.c	此函数没有实际作用
setup_reloc	common/board_f.c	**将 gd 拷贝到 gd->new_gd 的位置**（reserve_global_data 阶段从 start_addr_sp 中分配的），在完成 relocate 后，后面的程序可以继续使用 board_init_f 阶段配置的 gd 中的各种信息

完成 board_init_f 后的内存布局

程序执行早期是在 Nor Flash 或者 Nand Flash 上执行的，在完成 board_init_f 后会基本完成了内存的布局和分配，随后就会执行依次 realloc 的动作，将程序搬移到 SDRAM 中并跳转执行。根据前面对于 board_init_f 中列表 init_sequence_f 后半部分的分析，多个 reserve_xxx 相关的函数执行完成后，大概的内存布局和几个关键指针的位置：



Relocate

在执行完 board_init_f 后，返回 crt0.S 中的_main 函数继续执行 relocate 过程，首先配置栈顶指针并 8 字节对齐，并更新 gd 指针(r9 寄存器)指向的位置：

```
103     ldr sp, [r9, #GD_START_ADDR_SP] /* sp = gd->start_addr_sp */
104 #if defined(CONFIG_CPU_V7M) /* v7M forbids using SP as BIC destination */
105     mov r3, sp
106     bic r3, r3, #7
107     mov sp, r3
108 #else
109     bic sp, sp, #7  /* 8-byte alignment for ABI compliance */
110 #endif
111     ldr r9, [r9, #GD_BD]        /* r9 = gd->bd */
112     sub r9, r9, #GD_SIZE        /* new GD is below bd */
```

随后执行:

```
114     adr lr, here
115     ldr r0, [r9, #GD_RELOC_OFF]     /* r0 = gd->reloc_off */
116     add lr, lr, r0
117 #if defined(CONFIG_CPU_V7M)
118     orr lr, #1              /* As required by Thumb-only */
119 #endif
120     ldr r0, [r9, #GD_RELOCADDR]     /* r0 = gd->relocaddr */
121     b   relocate_code
122 here:
123 /*
124  * now relocate vectors
125  */
126
127     bl  relocate_vectors
```

上面的代码主要目的就是更新 lr 和跳转到 relocate_code 函数执行。"adr lr, here"是个伪指令，目的是为了将 lr 更新到 here 标签所在的位置，它会被编译成:

```
543     774:   e24990a8    sub r9, r9, #168    ; 0xa8
544     778:   e28fe00c    add lr, pc, #12
545     77c:   e5990040    ldr r0, [r9, #64]   ; 0x40
546     780:   e08ee000    add lr, lr, r0
547     784:   e599002c    ldr r0, [r9, #44]   ; 0x2c
548     788:   ea00001b    b   7fc <relocate_code>
549
550 0000078c <here>:
551     78c:   eb000010    bl  7d4 <relocate_vectors>
```

也就是 lr 会被更新为相对于当前 pc+12 的位置，arm9 中 PC 为当前执行指令+8，那么 lr 的地址就 0x780+0xc = 0x78c，即 here 标签的位置。

随后 lr 又做了一次偏移:

```
115     ldr r0, [r9, #GD_RELOC_OFF]     /* r0 = gd->reloc_off */
116     add lr, lr, r0
```

这里 gd->reloc_off 就是指向 SDRAM 中 u-boot 的代码段的首地址。也就是说此时 lr 里面存储的地址从 flash 地址改编为了 SDRAM 地址。

这里更新 lr 的目的也是配合后面的“b relocate_code”使用，B 指令并不会更新 lr 寄存器，在执行完 relocate_code 后：

```
123
124 #ifdef __ARM_ARCH_4__
125     mov pc, lr
126 #else
127     bx  lr
128 #endif
```

程序直接返回此处配置的 lr 地址。这样做就使得 u-boot 执行完 relocate_code 后，**代码实质上从 Flash 中已经搬移到了 SDRAM 中，那么就跳转到'here'标签所在的 SDRAM 中的地址继续执行。**

relocate_code 函数位于 arch/arm/lib/relocate.S 中，_main 中执行的 relocate_code 的入参是 relocate_code(gd->relocaddr)

```
120     ldr r0, [r9, #GD_RELOCADDR]     /* r0 = gd->relocaddr */
121     b   relocate_code
```

在开始分析 u-boot 的 relocate 之前，需要一些背景信息。首先修改根目录 Makefile:

```
930 u-boot.dis: u-boot
931         $(OBJDUMP) -D $< > $@
932
```

并执行 make ARCH=arm CROSS_COMPILE=arm-linux- u-boot.dis 产生反汇编文件

u-boot 在编译时，提供了-mword-relocations -fno-pic 来编译地址无关代码，并在链接时使用-pie 最终生成地址无关代码：

arch/arm/config.mk:

```
115 PLATFORM_CPPFLAGS += $(call cc-option, -mword-relocations)
116 PLATFORM_CPPFLAGS += $(call cc-option, -fno-pic)
117 endif
```

Makefile:

```
1178  # May be overridden by arch/$(ARCH)/config.mk
1179  quiet_cmd_u-boot__ ?= LD      $@
1180        cmd_u-boot__ ?= $(LD) $(LDFLAGS) $(LDFLAGS_u-boot) -o $@ \
1181        -T u-boot.lds $(u-boot-init)                              \
1182        --start-group $(u-boot-main) --end-group                  \
1183        $(PLATFORM_LIBS) -Map u-boot.map
1184
```

arch/arm/config.mk:

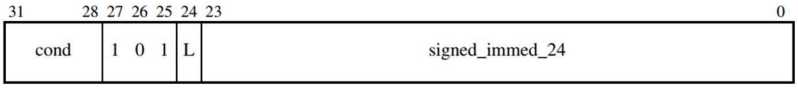
```
88   # needed for relocation
89   LDFLAGS_u-boot += -pie
90
```

这样编译后生成的代码就是地址无关代码，以 print_cpuinfo 为例，查看 u-boot.dis，其函数的调用：

```
283  000003b0 <print_cpuinfo>:
284       3b0:   e92d41f0    push    {r4, r5, r6, r7, r8, lr}
285       3b4:   e59f305c    ldr r3, [pc, #92]   ; 418 <print_cpuinfo+0x68>
286       3b8:   e24dd020    sub sp, sp, #32
287       3bc:   e5931000    ldr r1, [r3]
288       3c0:   e59f0054    ldr r0, [pc, #84]   ; 41c <print_cpuinfo+0x6c>
289       3c4:   eb0173e6    bl  5d364 <printf>
290       3c8:   e59f7050    ldr r7, [pc, #80]   ; 420 <print_cpuinfo+0x70>
291       3cc:   e59f6050    ldr r6, [pc, #80]   ; 424 <print_cpuinfo+0x74>
292       3d0:   e3a04000    mov r4, #0
```

BL 指令是地址无关指令：

A4.1.5 B, BL



B (Branch) and BL (Branch and Link) cause a branch to a target address, and provide both conditional and unconditional changes to program flow.

BL also stores a return address in the link register, R14 (also known as LR).

Syntax

B{L}{<cond>} <target_address>

where:

L — Causes the L bit (bit 24) in the instruction to be set to 1. The resulting instruction stores a return address in the link register (R14). If L is omitted, the L bit is 0 and the instruction simply branches without storing a return address.

<cond> — Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If <cond> is omitted, the AL (always) condition is used.

<target_address>

Specifies the address to branch to. The branch target address is calculated by:

1. Sign-extending the 24-bit signed (two's complement) immediate to 30 bits.
2. Shifting the result left two bits to form a 32-bit value.
3. Adding this to the contents of the PC, which contains the address of the branch instruction plus 8 bytes.

The instruction can therefore specify a branch of approximately ±32MB (see *Usage* on page A4-11 for precise range).

对于全局变量的访问，则通过类似 ldr Rd, [pc, imm]这样的方式，例如访问 freq_c[]:

```
3c4:   eb0173e6    bl  5d364 <printf>
3c8:   e59f7050    ldr r7, [pc, #80]   ; 420 <print_cpuinfo+0x70>
3cc:   e59f6050    ldr r6, [pc, #80]   ; 424 <print_cpuinfo+0x74>
3d0:   e3a04000    mov r4, #0
3d4:   e1a0500d    mov r5, sp
3d8:   e7d78004    ldrb    r8, [r7, r4]
```

那么访问的地址就是 0x3c8+0x8+0x50=0x420:

```
311        41c:   00064bbe    undefined instruction 0x00064bbe
312        420:   000639d0    ldrdeq  r3, [r6], -r0
313        424:   000639d4    ldrdeq  r3, [r6], -r4
314        428:   00064bcb    andeq   r4, r6, fp, asr #23
```

0x420 的数据是 0x639d0, 忽略后面的错误的反汇编提示，这个就是 freq_c[]的真实地址，最后配合 ldrb r8,[r7,r4]这样的方式把 0x639d0 位置的数据加载到 r8 中:

```
000639d0 <freq_c>:
   639d0:   00504846    subseq  r4, r0, r6, asr #16
```

而前面提到的，编译器和链接器的地址无关选项最终会生成.rel_dyn 段, u-boot.map:

```
5727 .rel_dyn_start  0x00078b08        0x0
5728   *(.__rel_dyn_start)
5729 .__rel_dyn_start
5730                 0x00078b08        0x0 arch/arm/lib/built-in.o
5731
5732 .rel.dyn        0x00078b08     0x96f0
5733   *(.rel*)
5734   .rel.got       0x00000000        0x0 arch/arm/cpu/arm920t/start.o
5735   .rel.plt       0x00000000        0x0 arch/arm/cpu/arm920t/start.o
```

这个段的作用很简单，前面提到了函数之间的调用都是使用了相对地址关系的指令，那么这些代码的执行的位置是不重要的，无论是 0x0 处执行还是 0x3000_0000 处执行，调用某个子函数都是去某个相对便宜地址去跳转，那么代码段整体搬移就可以了。

但是对于 data 的访问，程序首先在 pc+offset（后面称为 lable）处寻找这个 data 真实的地址，然后再去访问这个地址而获取数据，那么当代码段整体搬移时，这个 label 处的实际内容并不会改变，上面的例子，无论程序搬移到任何位置，pc+80(0x420)这个地址的内容永远都是 0x639d0, 但是因为程序与数据时整体一起搬移的，0x639d0 也应当做相应的 offset。那么问题来了，面对数以千计的函数，这些 lable 所在的位置也是根据各个函数分散再 text 段内，bootloader 无法知道每一个 label 的具体位置，这个时候生成的.rel.dyn 段

的作用就体现了，这个段里面的内容存储了这些 label 所在的位置，比如上面的例子，再 rel_dyn 段内：

```
Disassembly of section .rel.dyn:

00078b08 <__rel_dyn_end-0x96f0>:
   78b08:   00000020    andeq   r0, r0, r0, lsr #32
   78b0c:   00000017    andeq   r0, r0, r7, lsl r0
   78b10:   00000024    andeq   r0, r0, r4, lsr #32
   78b14:   00000017    andeq   r0, r0, r7, lsl r0
   78b18:   00000028    andeq   r0, r0, r8, lsr #32
   78b1c:   00000017    andeq   r0, r0, r7, lsl r0
   78b20:   0000002c    andeq   r0, r0, ip, lsr #32
   78b24:   00000017    andeq   r0, r0, r7, lsl r0
   78b28:   00000030    andeq   r0, r0, r0, lsr r0
   78b2c:   00000017    andeq   r0, r0, r7, lsl r0
   78b30:   00000034    andeq   r0, r0, r4, lsr r0
   78b34:   00000017    andeq   r0, r0, r7, lsl r0
   78b38:   00000038    andeq   r0, r0, r8, lsr r0
   78b3c:   00000017    andeq   r0, r0, r7, lsl r0
   78b40:   0000041c    andeq   r0, r0, ip, lsl r4
   78b44:   00000017    andeq   r0, r0, r7, lsl r0
   78b48:   00000420    andeq   r0, r0, r0, lsr #8
   78b4c:   00000017    andeq   r0, r0, r7, lsl r0
   78b50:   00000424    andeq   r0, r0, r4, lsr #8
   78b54:   00000017    andeq   r0, r0, r7, lsl r0
```

这个段内，每个条目由两个 DWORD 组成，一个是 label 的地址，另一个是固定的标签 0x17, 上面的例子中，0x420 就是存储着数据的位置。

所以 bootloader 需要搬移代码时需要做的两件事情：

1. 根据源地址和目标地址，把 text/bss/data 等段搬移到对应地址
2. 根据.rel.dyn 段寻找散落再搬移后的 text 段内的各个 label，并将其中的地址也做对应的偏移并写回

relocate_code 函数：

```
ENTRY(relocate_code)
    ldr r1, =__image_copy_start /* r1 <- SRC &__image_copy_start */
    subs    r4, r0, r1      /* r4 <- relocation offset */
    beq relocate_done       /* skip relocation */
    ldr r2, =__image_copy_end   /* r2 <- SRC &__image_copy_end */
```

加载 linker 中的__image_copy_start 地址，也就是.text 的首地址（0x0），如果和 r0（gd->relocaddr, 也就是目标地址）相等则跳过 relocation。

开始拷贝直到 r1 的地址和__image_copy_end 相等：

```
85 copy_loop:
86     ldmia   r1!, {r10-r11}      /* copy from source address [r1]    */
87     stmia   r0!, {r10-r11}      /* copy to   target address [r0]    */
88     cmp r1, r2          /* until source end address [r2]    */
89     blo copy_loop
```

这里就完成的第一步代码和数据的拷贝。

加载.rel.dyn 段的起始与结束地址：

```
94     ldr r2, =__rel_dyn_start    /* r2 <- SRC &__rel_dyn_start */
95     ldr r3, =__rel_dyn_end  /* r3 <- SRC &__rel_dyn_end */
```

读取地址的双 DWORD 并存储再 r0,r1 中，并判断 r1 中的固定标签是不是 0x17, 如果不匹配则跳转到 fixnext：

```
96 fixloop:
97     ldmia   r2!, {r0-r1}        /* (r0,r1) <- (SRC location,fixup) */
98     and r1, r1, #0xff
99     cmp r1, #23         /* relative fixup? */
100    bne fixnext
```

再 fixnext 中判断是不是.rel.dyn 段已经读完了，如果不是则返回前面的 fixloop 继续读下一条，若是则 relocate 结束：

```
107 fixnext:
108    cmp r2, r3
109    blo fixloop
110
111 relocate_done:
```

如果标签 0x17 匹配上，那么执行：

```
102    /* relative fix: increase location by offset */
103    add r0, r0, r4
104    ldr r1, [r0]
105    add r1, r1, r4
106    str r1, [r0]
107 fixnext:
```

首先将 r0 内的地址做偏移，如前面的例子，代码从 0x0 搬移到 0x3000_0000, 那么 offset=0x3000_0000, 对应的，r0=0x420 就是搬移前的地址，则需要更新为 0x3000_0420, r4 中存储的就是这个偏移，在最开始代码段搬移中获取的。

把 0x3000_0420 中的 0x639d0 读取出来，再次加上 offset，就是对应第二个 add 指令，此时 r1=0x3006_39d0。最后把 0x3006_39d0 这个新的数据地址，储存回 0x3000_0420 中去。

当 print_cpuinfo 中再次执行到“ldr　r7, [pc, #80] “这样的指令时，就能从正确的 lable 地址加载到正确的全局变量地址了。

当 relocate_code 中最后执行”bx lr”时，前面分析过，程序就会跳转到 SDRAM 中继续执行剩下的程序。

回到 crt0.S 中，继续执行 relocate_vectors, 根据 CP15 中 c1 寄存器 Vbit 的内容确定中断向量表的位置：

```
46      /*
47       * Copy the relocated exception vectors to the
48       * correct address
49       * CP15 c1 V bit gives us the location of the vectors:
50       * 0x00000000 or 0xFFFF0000.
51       */
52      ldr r0, [r9, #GD_RELOCADDR] /* r0 = gd->relocaddr */
53      mrc p15, 0, r2, c1, c0, 0   /* V bit (bit[13]) in CP15 c1 */
54      ands    r2, r2, #(1 << 13)
55      ldreq   r1, =0x00000000     /* If V=0 */
56      ldrne   r1, =0xFFFF0000     /* If V=1 */
57      ldmia   r0!, {r2-r8,r10}
58      stmia   r1!, {r2-r8,r10}
59      ldmia   r0!, {r2-r8,r10}
60      stmia   r1!, {r2-r8,r10}
```

B3.4.1 Control register

This register contains:

- Enable/disable bits for the caches, MMUs, and other memory system blocks that are primarily controlled by other CP15 registers. This allows these memory system blocks to be programmed correctly before they are enabled.
- Various configuration bits for memory system blocks and for the ARM processor itself.

Note

Extra bits of both varieties might be added in the future. Because of this, this register should normally be updated using read/modify/write techniques, to ensure that currently unallocated bits are not needlessly modified. Failure to observe this rule might result in code which has unexpected side effects on future processors.

31 … 27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UNP/SBZP	L2	EE	VE	XP	U	FI		L4	RR	V	I	Z	F	R	S	B	L	D	P	W	C	A	M

When a control bit in CP15 register 1 is not applicable to a particular implementation, it reads as the value that most closely reflects that implementation, and ignores writes. (Specific examples of this general rule are documented in the individual bit descriptions below.) Apart from bits that read as 1 according to this rule, all bits in CP15 register 1 are set to 0 on reset.

V (bit[13]) This bit is used to select the location of the exception vectors:

0 = Normal exception vectors selected (address range 0x00000000-0x0000001C)

1 = High exception vectors selected (address range 0xFFFF0000-0xFFFF001C).

An implementation can provide an input signal that determines the state of this bit after reset.

继续执行 c_runtime_cpu_setup:

```
102     .globl  c_runtime_cpu_setup
103 c_runtime_cpu_setup:
104
105     mov pc, lr
106
```

但是什么也没有做..这里不知道原作者有何种考虑。

返回 crt0.S 中后继续执行代码将整个 bss 段清零:

```
141     ldr r0, =__bss_start    /* this is auto-relocated! */
142
143 #ifdef CONFIG_USE_ARCH_MEMSET
144     ldr r3, =__bss_end      /* this is auto-relocated! */
145     mov r1, #0x00000000     /* prepare zero to clear BSS */
146
147     subs    r2, r3, r0      /* r2 = memset len */
148     bl  memset
149 #else
150     ldr r1, =__bss_end      /* this is auto-relocated! */
151     mov r2, #0x00000000     /* prepare zero to clear BSS */
152
153 clbss_l:cmp r0, r1          /* while not at end of BSS */
154 #if defined(CONFIG_CPU_V7M)
155     itt lo
156 #endif
157     strlo   r2, [r0]        /* clear 32-bit BSS word */
158     addlo   r0, r0, #4      /* move to next */
159     blo clbss_l
160 #endif
```

最后准备入参 r0, r1 后跳转执行 board_init_r(gd_t *new_gd, ulong dest_addr)

board_init_r

board_init_r 与前面的 board_init_f 类似，最终会执行再 init_sequence_r 中注册的各个驱动初始化函数，不一样的是再 init_sequence_r 的最后会执行 u-boot 的主函数并不会再返回:

执行的初始化:

函数名	位置	说明
initr_trace	common/board_r.c	trace buffer 初始化

initr_reloc	common/board_r.c	写入全局标志表明 reloc 完成
initr_caches	arch/arm/lib/cache.c	无实际作用
initr_reloc_global_data	common/board_r.c	monitor_flash_len 设置为从.text 开始到.end 结束位置的大小
initr_barrier	common/board_r.c	无用
initr_malloc	common/board_r.c	初始化 malloc 功能，heap 空间起始位置指向 gd->relocaddr – 4MB
initr_console_record	common/board_r.c	无用
bootstage_relocate	common/bootstage.c	record[i].name 拷贝到新的空间
initr_bootstage	common/board_r.c	无用
board_init	board/samsung/smdk2410/smdk2410.c	配置 arch_number, 配置启动参数地址 0x3000_0100, 使能 I/D cache 与 MMU，参见补充：CACHE 与 MMU 的初始化
stdio_init_tables	common/stdio.c	参见补充：打印函数
initr_serial	common/board_r.c	参见补充：打印函数
initr_announce	common/board_r.c	打印"Now running in RAM - U-Boot at: %08lx\n", gd->relocaddr
power_init_board	common/board_r.c	没用
initr_flash	common/board_r.c	驱动，Nor Flash 初始化，参见补充：Nor Flash 初始化
initr_nand	common/board_r.c	初始化 Nand Flash 驱动
initr_env	common/env_nand.c	参见补充：环境变量初始化
initr_secondary_cpu	common/board_r.c	没用
stdio_add_devices	common/stdio.c	参见补充：打印函数
initr_jumptable		
console_init_r	common/console.c	参见补充：打印函数
interrupt_init	arch/arm/lib/interrupts.c	没用
initr_enable_interrupts	arch/arm/lib/interrupts.c	没用

initr_ethaddr	board_r.c	从 env 中获取 ethaddr 信息并存储在 bd->bi_enetaddr 中
initr_net	board_r.c	参见补充：网卡初始化

当 board_init_r 中完成了所有的有关设备的注册或者初始化后，则执行 run_main_loop，并调用 main_loop()执行 U-Boot 主函数。

主函数

main_loop 经过简化后的流程：

```
void main_loop(void)
{
    ...
    cli_init(); <-- 初始化cli

    ...

    s = bootdelay_process(); <-- 获取bootdelay和bootcmd

    ...

    autoboot_command(s); <-- 判断bootdelay时间内是否有用户输入:
                             如有则打断boot, 进入hush shell
                             如果没有, 则实行bootcmd里面的内容再前面的配置中:
                             bootcmd="nand read 30000000 kernel 0x400000;"
                                     "bootm 30000000"

    cli_loop(); <==>    parse_file_outer(); for (;;); <-- 进入hush shell执行输入命令
}
```

bootdelay_process()函数中，主要过程就是获取环境变量中的 bootdelay 和 bootcmd：

```
s = getenv("bootdelay");
```

```
        s = getenv("bootcmd");
```

通过 getenv()获取环境变量内容的细节参见补充：环境变量初始化

随后执行 autoboot_command(), 其精简后：

```
1  void autoboot_command(const char *s)
2  {
3      if (stored_bootdelay != -1 && s && !abortboot(stored_bootdelay)) {
4          run_command_list(s, -1, 0);
5      }
6  }
7
```

abortboot()就是通过获取 timer, 检查 uart 内是否有数据和比较 bootdelay 的剩余时间来确定是否有按键:

```
while ((bootdelay > 0) && (!abort)) {
    --bootdelay;
    /* delay 1000 ms */
    ts = get_timer(0);
    do {
        if (tstc()) {   /* we got a key press   */
            abort  = 1; /* don't auto boot  */
            bootdelay = 0;  /* no more delay    */
            (void) getc();  /* consume input    */
            break;
        }
        udelay(10000);
    } while (!abort && get_timer(ts) < 1000);

    printf("\b\b\b%2d ", bootdelay);
}
```

如果没有按键输入，超时后则执行:

```
        run_command_list(s, -1, 0);
```

如果有，autoboot_command()则会退出，继续执行 main_loop 中的的 cli_loop()，进入 hush_shell 中。

也就是 U-Boot 在此处根据是否有按键输入进去了两个分支:

1. 规定时间内没有按键，通过 run_command_list 执行环境变量中 bootcmd 下定义的命令;
2. 规定时间内有按键触发，则执行 cli_loop()进入 U-Boot 的 Hush Shell;

大概分析一下 Hush Shell，代码不长但是牵扯到的流程比较多，不进行详细的分析了:

cli_loop()->parse_file_outer()

```
1  int parse_file_outer(void)
2  {
3      int rcode;
4      struct in_str input;
5
6      setup_file_in_str(&input); <-- 初始化input, 并注册file_peek, file_get函数指针
7      rcode = parse_stream_outer(&input, FLAG_PARSE_SEMICOLON); <-- Hush Shell主体
8      return rcode;
9  }
```

```
static int parse_stream_outer(struct in_str *inp, int flag)
{
    ...
    do {
        ctx.type = flag;
        initialize_context(&ctx);       <-- 初始化ctx
        ...
        rcode = parse_stream(   &temp,  \  <-- 获取用户输入
                                &ctx,   \
                                inp,    \
                                flag & FLAG_CONT_ON_NEWLINE ? -1 : '\n');
        ...
        if (rcode != 1 && ctx.old_flag == 0) {
            done_word(&temp, &ctx);
            done_pipe(&ctx,PIPE_SEQ);
            code = run_list(ctx.list_head); <-- 执行用户输入
            ...
        } else {
            ...
        }
        b_free(&temp);
    /* loop on syntax errors, return on EOF */
    } while (   rcode != -1 && \
                !(flag & FLAG_EXIT_FROM_LOOP) && \
                (inp->peek != static_peek || b_peek(inp)));

    return (code != 0) ? 1 : 0;
}
```

上面代码中，run_list 中最终会调用执行用户输入命令的函数，其执行流程：

run_list()→run_list_real()→run_pipe_real()→cmd_process()→find_cmd()→ll_entry_start()/ll_entry_count()/find_cmd_tbl()

也就是最终会根据用户的键盘输入，在补充：环境变量初始化中提到的 link list 中搜索名称匹配的表项，并获取其中的函数指针，这也是 hush shell 中的命令如何与代码中定义的命令关联上。

回到 autoboot_command()中，如果在限定时间内没有任何键盘输入，则会执行 run_command_list(), 这里面实际执行的命令：

```
bootargs=console=ttySAC0,115200 root=/dev/mtdblock3
bootcmd=nand read 30000000 kernel 0x400000;bootm 30000000
bootdelay=5
```

也就是启动的时候，分成两条指令：

1. nand read 30000000 kernel 0x400000;
2. bootm 30000000

第一条指令的作用是执行 nand 命令：

在 cmd/nand.c 中定义了：

```
U_BOOT_CMD(
    nand, CONFIG_SYS_MAXARGS, 1, do_nand,
    "NAND sub-system", nand_help_text
);
```

“nand”命令的入口函数就是 do_nand(), 不再记录这个函数的分析细节，这部分比较直接，就是执行 nand read 操作，从 flash 中 kernel 地址（0x10_0000）读取 4M 大小（0x40_0000）到 SDRAM 地址 0x3000_0000 中去。

随后执行第二个命令”bootm 0x30000000”, 即从 SDRAM 地址 0x3000_0000 处启动。

bootm 指令在 cmd/bootm.c 中定义：

```
U_BOOT_CMD(
    bootm,  CONFIG_SYS_MAXARGS, 1,  do_bootm,
    "boot application image from memory", bootm_help_text
);
```

也就是会通过执行:

```
int do_bootm(cmd_tbl_t *cmdtp, int flag, int argc, char * const argv[])
```

这里 argc=2, argv[0]=”bootm”, argv[1]=”0x30000000”, 至此，U-Boot 开始在内存 0x3000_0000 处开始读取 kernel image header, 从而进行 Linux 引导和启动。

补充：CACHE 与 MMU 的初始化

在 board/Samsung/smdk2410/smdk2410.c 中的 board_init():

```
icache_enable();
dcache_enable();
```

等同于在 arch/arm/lib/cache-cp15.c 中执行了 cache_enable(CR_I), cache_enable(CR_C)

```
187  /* cache_bit must be either CR_I or CR_C */
188  static void cache_enable(uint32_t cache_bit)
189  {
190      uint32_t reg;
191
192      /* The data cache is not active unless the mmu is enabled too */
193      if ((cache_bit == CR_C) && !mmu_enabled())
194          mmu_setup();
195      reg = get_cr(); /* get control reg. */
196      cp_delay();
197      set_cr(reg | cache_bit);
198  }
```

get_cr()就是使用"mcr"指令的内联汇编操作 CP15 协处理器（参见 arch/arm/include/asm/system.h）

CP15 register 1:

I (bit[12]) If separate L1 caches are used, this is the enable/disable bit for the L1 instruction cache:

0 = L1 instruction cache disabled

1 = L1 instruction cache enabled.

If an L1 unified cache is used or the L1 instruction cache is not implemented, this bit reads as 0 and ignores writes. If the L1 instruction cache cannot be disabled, this bit reads as 1 and ignores writes.

The state of this bit does not affect further levels of cache in the system.

C (bit[2]) If a L1 unified cache is used, this is the enable/disable bit for the unified cache. If separate L1 caches are used, this is the enable/disable bit for the data cache. In either case:

0 = L1 unified/data cache disabled

1 = L1 unified/data cache enabled.

If the L1 cache is not implemented, this bit reads as 0 and ignores writes. If the L1 cache cannot be disabled, this bit reads as 1 and ignores writes.

The state of this bit does not affect other levels of cache in the system.

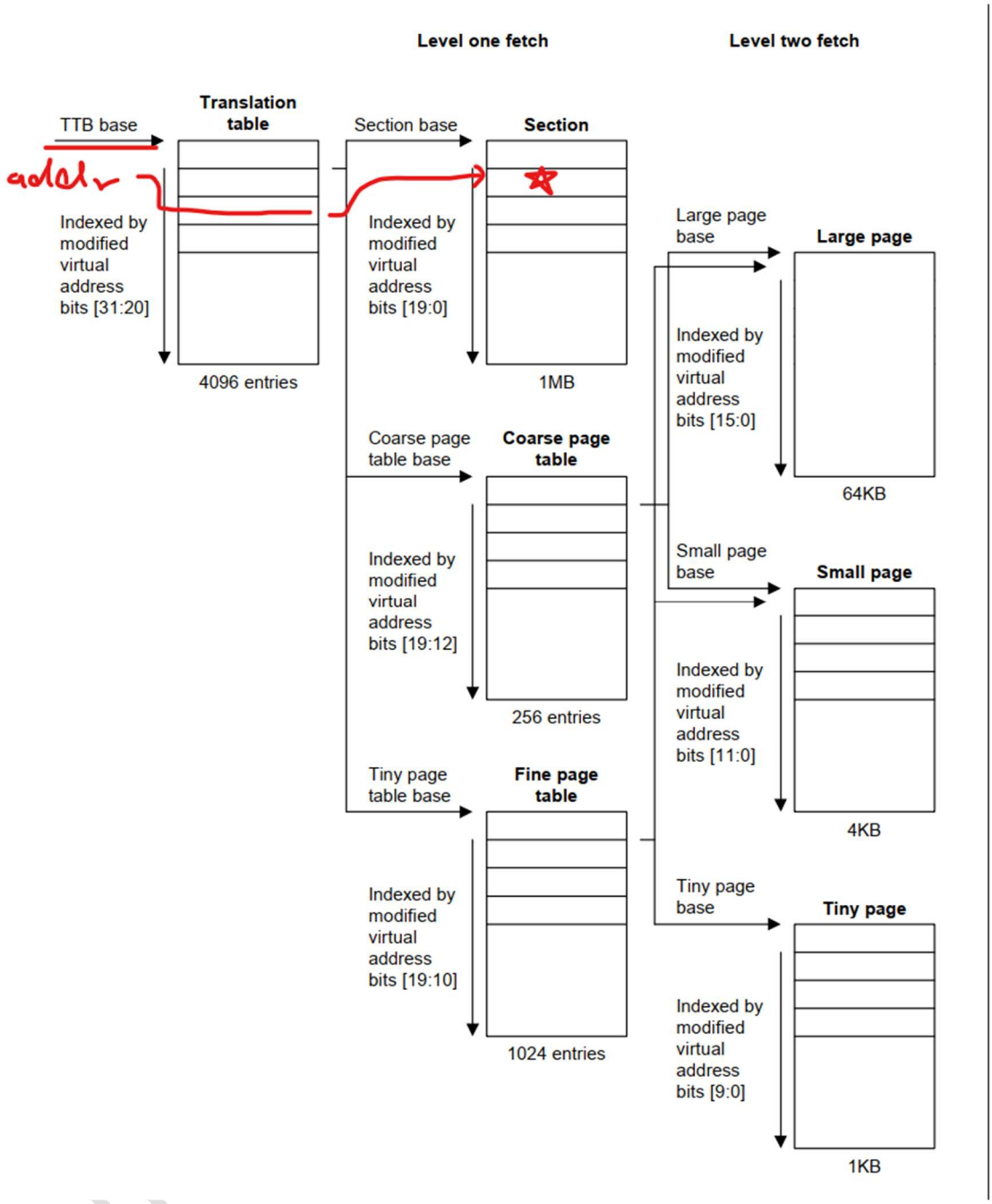
初始化 data cache 之前，如果 MMU 没有使能，则开始配置 MMU。

ARM920T 的 MMU 介绍：

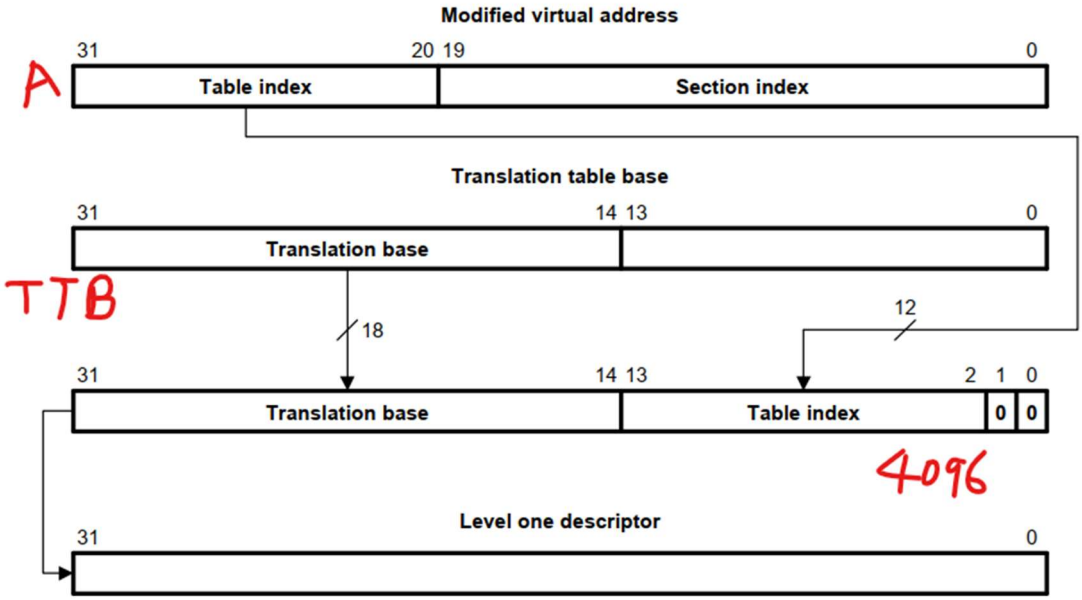
MMU 中，首先具有一个 64 条的 TLB，这个 TLB 类似 MMU 的 CACHE，会缓存命中的页表存在此处，下次地址转换时就不需要进行 table walk 了。 如果没有命中，那么硬件就会执行 table walk, 这里可以分成两级，level one fetch 与 level two fetch。

在 MMU 中，其内部有一个 TTB base 寄存器，其前 18 比特与访问的虚拟地址的前 12 比特完成了第一级查找表的索引。

MMU table walk 宏观流程：



level one fetch:



A 就是虚拟地址，这里的虚拟地址是经过 arm4 中的 FCSE（Fast Context Switch Extension）修改过的虚拟地址，此处默认就是 modified virtual address = virtual address

TTB 就是 MMU 中的 TTB 寄存器，提供了前 18BIT，结合虚拟地址的 BIT[31:20]就能确定具有 4096 个条目的查找表中的具体位置，而 BIT[1:0]则表明 level one descriptior 的种类，是一个 section 还是需要继续二级查找表：

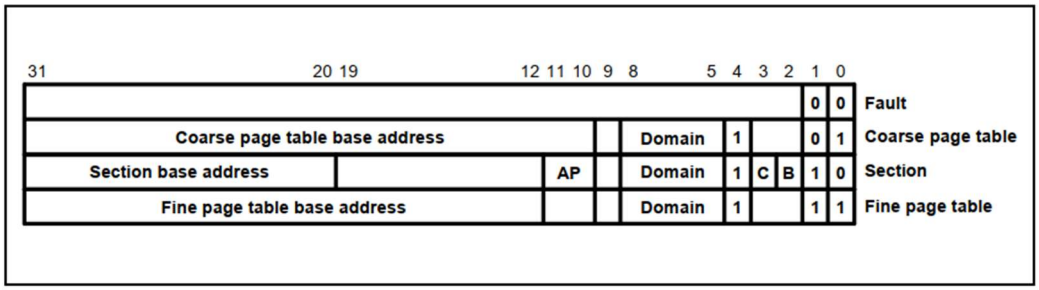


Figure 3-4. Level One Descriptors

The two least significant bits indicate the descriptor type

Table 3-2. Interpreting Level One Descriptor Bits [1:0]

Value	Meaning	Notes
00	Invalid	Generates a section translation fault.
01	Coarse page table	Indicates that this is a coarse page table descriptor.
10	Section	Indicates that this is a section descriptor.
11	Fine page table	Indicates that this is a fine page table descriptor.

如果 BIT[1:0]=2b10, 那么就是 section, section 中 BIT[3:2]就是表明 CACHE 类型的，可分为 cache-writeback, cache-writethough, non-cached non-buffered 和 non-cached buffered。

writethough cache 就是直写模式，在数据更新是，同时将数据写入 cache 和 RAM 中，这种逻辑简单可靠，但是每次写 RAM，消耗 CPU 速度

writeback cache 为回写模式，当 CPU 写新数据时并不会立刻把数据写入 RAM 中去，直到 cache 内容要被替换是才会写回 RAM，这样 CPU 效率最高，但是实现复杂。

Ctt and Ccr	Btt	Data cache, write buffer and memory access behavior
0 (1)	0	Non-cached, non-buffered (NCNB) Reads and writes are not cached and always perform accesses on the ASB and may be externally aborted. Writes are not buffered. The CPU halts until the write is completed on the ASB. Cache hits should never occur. (2)
0	1	Non-cached buffered (NCB) Reads and writes are not cached, and always perform accesses on the ASB. Cache hits should never occur. Writes are placed in the write buffer and will appear on the ASB. The CPU continues execution as soon as the write is placed in the write buffer. Reads may be externally aborted. Writes can not be externally aborted.
1	0	Cached, write-through mode (WT) Reads which hit in the cache will read the data from the cache and do not perform an access on the ASB. Reads which miss in the cache cause a linefill. All writes are placed in the write buffer and will appear on the ASB. The CPU continues execution as soon as the write is placed in the write buffer. Writes which hit in the cache update the cache. Writes cannot be externally aborted.
1	1	Cached, write-back mode (WB) Reads which hit in the cache will read the data from the cache and do not perform an ASB access. Reads which miss in the cache cause a linefill. Writes which miss in the cache are placed in the write buffer and will appear on the ASB. The CPU continues execution as soon as the write is placed in the write buffer. Writes which hit in the cache update the cache and mark the appropriate half of the cache line as dirty, and do not cause an ASB access. Cache write-backs are buffered. Writes (Cache write-misses and cache write-backs) cannot be externally aborted.

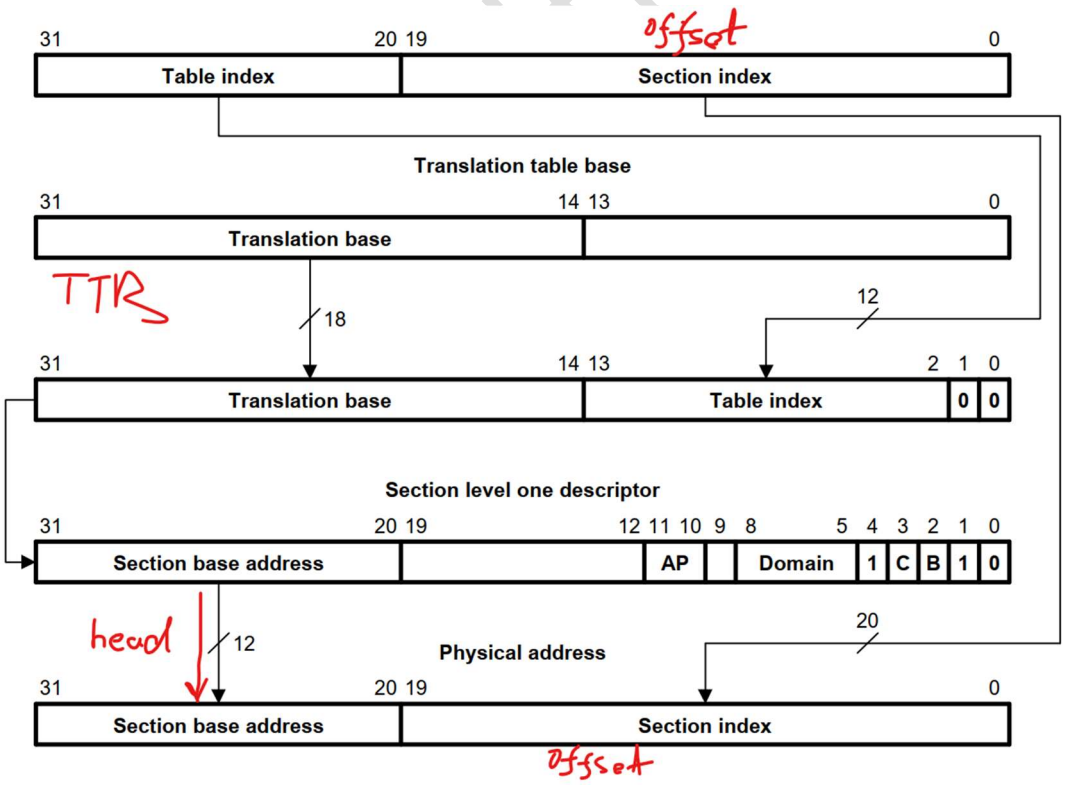
BIT[11:10]就是 AP 位，access permission, 其含义：

Table 3-6 shows how to interpret the access permission (AP) bits and how their interpretation is dependent upon the S and R bits (control register bits 8 and 9).

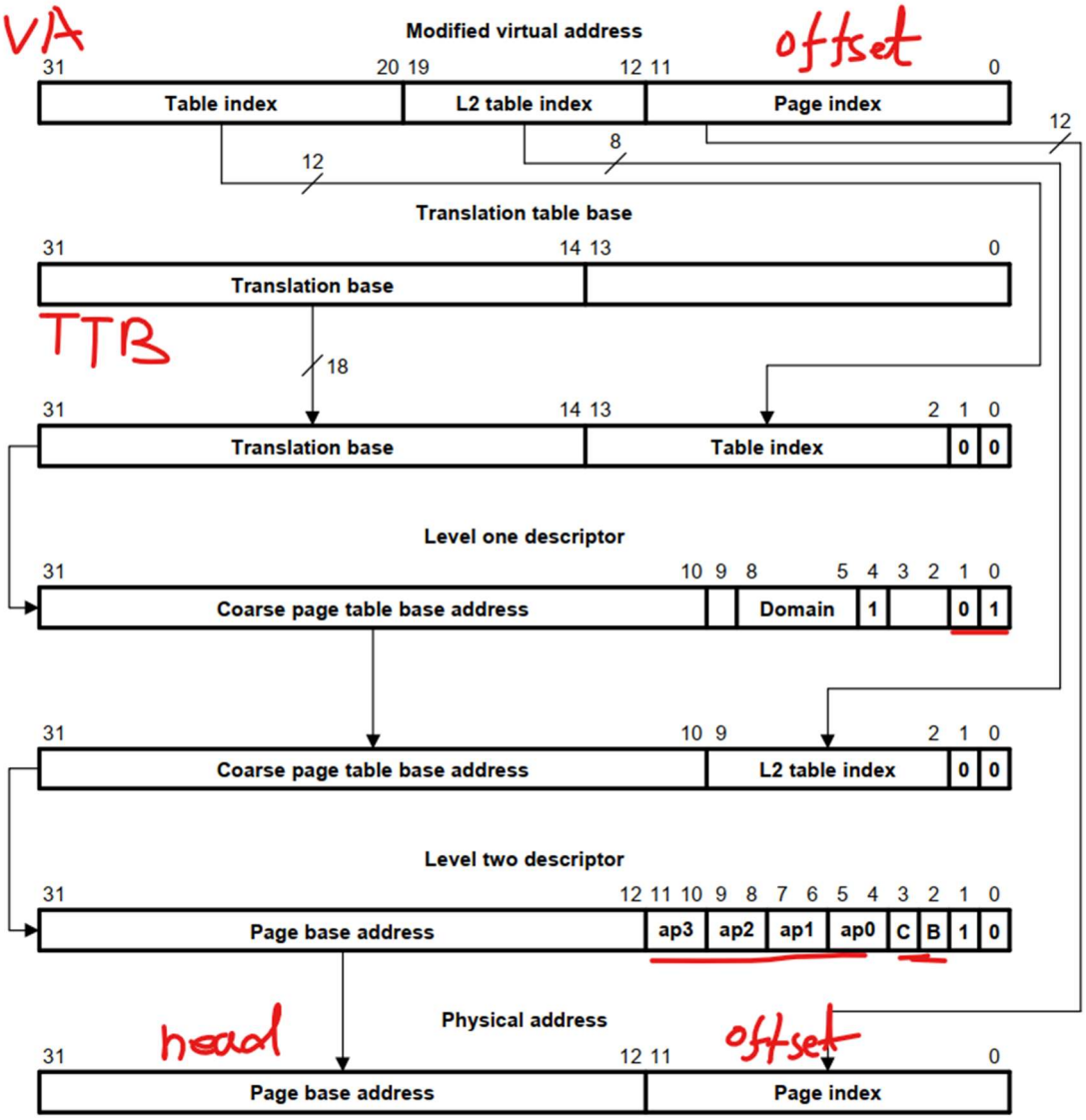
Table 3-6. Interpreting Access Permission (AP) Bits

AP	S	R	Supervisor Permissions	User Permissions	Notes
00	0	0	No access	No access	Any access generates a permission fault
00	1	0	Read only	No access	Supervisor read only permitted
00	0	1	Read only	Read only	Any write generates a permission fault
00	1	1	Reserved		
01	x	x	Read/write	No access	Access allowed only in supervisor mode
10	x	x	Read/write	Read only	Writes in user mode cause permission fault
11	x	x	Read/write	Read/write	All access types permitted in both modes.
xx	1	1	Reserved		

level one descriptor 的物理地址转换，如果 descriptiorBIT[1:0]=section, 那么物理地址的转换就是:



如果 level one descriptor != section, 那么 table walk 过程就会继续，进行第二级查找，可以根据页的大小继续分为 large page(64KB), small page(4KB), tiny page(1KB), 以 small page 为例:



这里需要注意的是，level one descriptor 中可以指向两种大小的 table: 256 条的 coarse page table 和 1024 条的 fine page table, 如果是 fine page table 指向的全都是 1KB 的 tiny page, 那么整个 MMU 下的查找表覆盖的范围就是 4096*1024*1KB=4GB。而如果是 fine page table 指向的全都是 64KB 的 large page, 那么能转换的空间范围则达到了 4096*1024*64KB=256GB？手册中同时提到了：

TRANSLATING LARGE PAGE REFERENCES

Figure 3-7 on page 3-13 illustrates the complete translation sequence for a 64KB large page.

As the upper four bits of the page index and low-order four bits of the coarse page table index overlap, each coarse page table entry for a large page must be duplicated 16 times (in consecutive memory locations) in the coarse page table.

If a large page descriptor is included in a fine page table the upper six bits of the page index and low-order six bits of the fine page table index overlap, each fine page table entry for a large page must therefore be duplicated 64 times.

类似 fine page table+small page(4KB)也是：

> Figure 3-8 illustrates the complete translation sequence for a 4KB small page. If a small page descriptor is included in a fine page table, the upper two bits of the page index and low-order two bits of the fine page table index overlap. Each fine page table entry for a small page must therefore be duplicated four times.

所以无论 coarse/fine table 如何组合 large/small/tiny page, 虚实地址转换的空间都不会超过 4GB。

回到 cache-cp15.c 的代码中，经过精简后的 mmu_setup():

```
97  static inline void mmu_setup(void)
98  {
99      int i;
100     u32 reg;
101
102     arm_init_before_mmu();
103     /* Set up an identity-mapping for all 4GB, rw for everyone */
104     for (i = 0; i < ((4096ULL * 1024 * 1024) >> MMU_SECTION_SHIFT); i++)
105         set_section_dcache(i, DCACHE_OFF);
106
107     for (i = 0; i < CONFIG_NR_DRAM_BANKS; i++) {
108         dram_bank_mmu_setup(i);
109     }
110
111     /* Copy the page table address to cp15 */
112     asm volatile("mcr p15, 0, %0, c2, c0, 0"
113                  : : "r" (gd->arch.tlb_addr) : "memory");
114
115     /* Set the access control to all-supervisor */
116     asm volatile("mcr p15, 0, %0, c3, c0, 0"
117                  : : "r" (~0));
118
119     arm_init_domains();
120
121     /* and enable the mmu */
122     reg = get_cr(); /* get control reg. */
123     cp_delay();
124     set_cr(reg | CR_M);
125 }
```

MMU_SECTION_SHIFT=20，第一个 for 循环就是循环 4096 次。

```c
void set_section_dcache(int section, enum dcache_option option)
{

    u32 *page_table = (u32 *)gd->arch.tlb_addr;
    u32 value = TTB_SECT_AP;

    /* Add the page offset */
    value |= ((u32)section << MMU_SECTION_SHIFT);

    /* Add caching bits */
    value |= option;

    /* Set PTE */
    page_table[section] = value;
}
```

```c
#define TTB_SECT_AP     (3 << 10)
/* options available for data cache on each page */
enum dcache_option {
    DCACHE_OFF = 0x12,
    DCACHE_WRITETHROUGH = 0x1a,
    DCACHE_WRITEBACK = 0x1e,
    DCACHE_WRITEALLOC = 0x16,
};
```

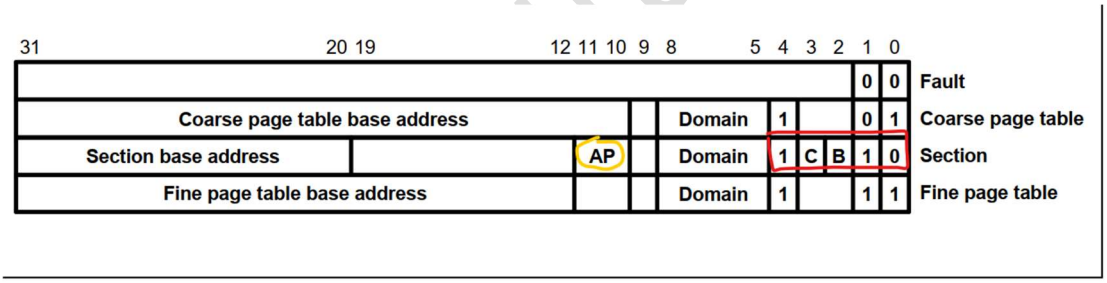


Figure 3-4. Level One Descriptors

结合前面的分析和 level one descriptors 的结构，第一个 for 循环就是把 4096 个 section 都配置为缓存策略为 NCNB 的，all accessble 的空间，每个 section 索引的范围是 1MB，正好遍历完成的 4GB 空间。

对于第二个 for 循环，由于没有额外定义 dram_init_banksize 函数，所以

bi_dram[0].start = 0x3000_0000

bi_dram[0].size = 0x0400_0000

> 也就是实际的物理内存只有 64MB, 那么就循环 64 次设置 RAM 所对应的 section 下 (768-832)的 cache 策略为 writethrough:

```
__weak void dram_bank_mmu_setup(int bank)
{
    bd_t *bd = gd->bd;
    int i;

    debug("%s: bank: %d\n", __func__, bank);
    for (i = bd->bi_dram[bank].start >> MMU_SECTION_SHIFT;
         i < (bd->bi_dram[bank].start >> MMU_SECTION_SHIFT) +
         (bd->bi_dram[bank].size >> MMU_SECTION_SHIFT);
         i++) {
        set_section_dcache(i, DCACHE_WRITETHROUGH);

    }
}
```

随后配置 TTB，把 translate table 的首地址写入 TTB 中：

The following instructions can be used to access the TTB:

MRC p15, 0, Rd, c2, c0, 0; read TTB register

MCR p15, 0, Rd, c2, c0, 0; write TTB register

配置访问属性为全可访问：

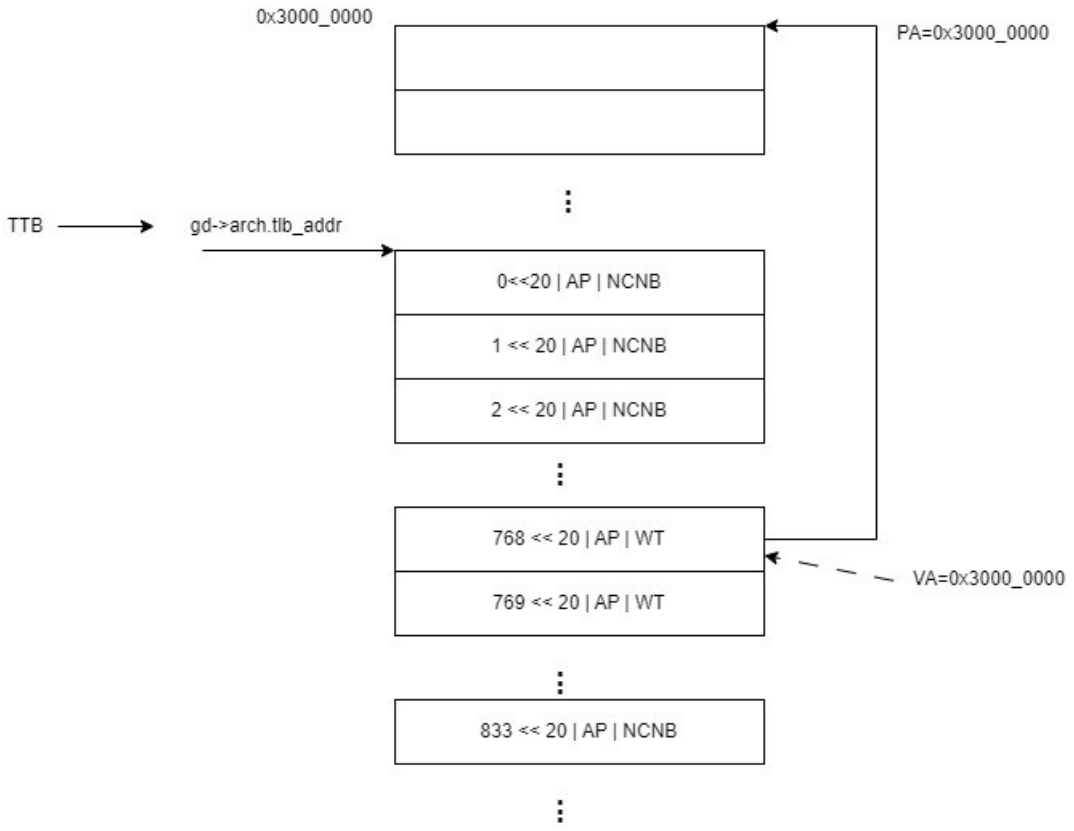
The encoding of the two bit domain access permission field is given in Table 3-5 on page 3-19. The following instructions can be used to access the domain access control register:

MRC p15, 0, Rd, c3, c0, 0; read domain 15:0 access permissions

MCR p15, 0, Rd, c3, c0, 0; write domain 15:0 access permissions

最后设置 CP15 Register1 CR_M 位使能 MMU。

最后初始化的 MMU 的页表大概结构就是：



也就是虚拟地址等于物理地址，RAM 范围内的地址 CACE 策略全是 writethough。 大致来看就是在 U-Boot 下，MMU 的功能用了，但是用的不多。

补充：打印函数

打印函数是非常基础的组件之一，其功能贯穿整个 u-boot 工作周期。

在 lib/Makefile 下，默认将 vsprintf.c 引入编译：

```
90  else
91  # Main U-Boot always uses the full printf support
92  obj-y += vsprintf.o panic.o strto.o
93  endif
```

而 vsprintf.c 中则定义了， 其中…为 C 语言中的可变参数列表

```c
int printf(const char *fmt, ...)
{
    va_list args;
    uint i;
    char printbuffer[CONFIG_SYS_PBSIZE];

    va_start(args, fmt);

    /*
     * For this to work, printbuffer must be larger than
     * anything we ever want to print.
     */
    i = vscnprintf(printbuffer, sizeof(printbuffer), fmt, args);
    va_end(args);

    /* Print the string */
    puts(printbuffer);
    return i;
}
```

va_start()于 va_end()配合可变参数列表使用 va_args, vscnprintf()则将字串模板和参数表内容展开并写入 printbuffer[]中，而 puts(printbuffer)则最终将数组内的字符串输出。

在 common/console.c 下面定义了 puts(const char *s):

```c
void puts(const char *s)
{
    if (!gd->have_console)
        return pre_console_puts(s);

    if (gd->flags & GD_FLG_DEVINIT) {
        /* Send to the standard output */
        fputs(stdout, s);
    } else {
        /* Send directly to the handler */
        pre_console_puts(s);
        serial_puts(s);
    }
}
```

gd->have_console 在后面板级初始化分析中会提到，在 board_init_f()下会首先初始化为 0，只有在第一次执行了 console_init_f()中时，才会配置为 1。所以在完成 console_init_f()之前调用 puts()都会被强制执行 pre_console_puts(s)，这个函数在没有定义 CONFIG_PRE_CONSOLE_BUFFER 时默认为空函数:

```
454  }
455  #else
456  static inline void pre_console_putc(const char c) {}
457  static inline void pre_console_puts(const char *s) {}
458  static inline void print_pre_console_buffer(int flushpoint) {}
459  #endif
460
```

等效于 puts（）无效。

而判断 GD_FLG_DEVINIT 则表示，如果已经完成 console_init_f(), 但尚未完成 console_init_r()，则直接调用驱动 API serial_puts(s). serial_puts()则指向了 drivers/serial/serial_s3c24x0.c: serial_puts_dev(0, s)。

```
497  void serial_puts(const char *s)
498  {
499      get_current()->puts(s);
500  }
501
```

```
382  static struct serial_device *get_current(void)
383  {
384      struct serial_device *dev;
385
386      if (!(gd->flags & GD_FLG_RELOC))
387          dev = default_serial_console();
388      else if (!serial_current)
389          dev = default_serial_console();
390      else
391          dev = serial_current;
392
```

```
190  __weak struct serial_device *default_serial_console(void)
191  {
192  #if defined(CONFIG_SERIAL1)
193      return &s3c24xx_serial0_device;
194  #elif defined(CONFIG_SERIAL2)
195      return &s3c24xx_serial1_device;
196  #elif defined(CONFIG_SERIAL3)
197      return &s3c24xx_serial2_device;
198  #else
199  #error "CONFIG_SERIAL? missing."
200  #endif
201  }
```

```
179
180  DECLARE_S3C_SERIAL_FUNCTIONS(0);
181  struct serial_device s3c24xx_serial0_device =
182  INIT_S3C_SERIAL_STRUCTURE(0, "s3ser0");
```

```
51      } \
52      void s3serial##port##_puts(const char *s) \
53      { \
54          serial_puts_dev(port, s); \
55      }
56
```

而对于 fputs()则相对复杂一些，其初始化流程大概是 board_init_r()->stdio_add_devices()->drv_system_init()->console_init_r()->stdio_get_list()->console_setfile()->stdio_devices[]以上顺序并不是严格的子函数调用顺序，而是大概的实际执行流程。

完成初始化后，就会把 stdio_serial_puts()注册在 stdio.c 下的全局结构体 static struct stdio_dev devs 的链表下。

```
66
67  static void stdio_serial_puts(struct stdio_dev *dev, const char *s)
68  {
69      serial_puts(s);
70  }
71
```

也就是当执行 fputs()时：

fputs()->console_puts()->stdio_devices[]:

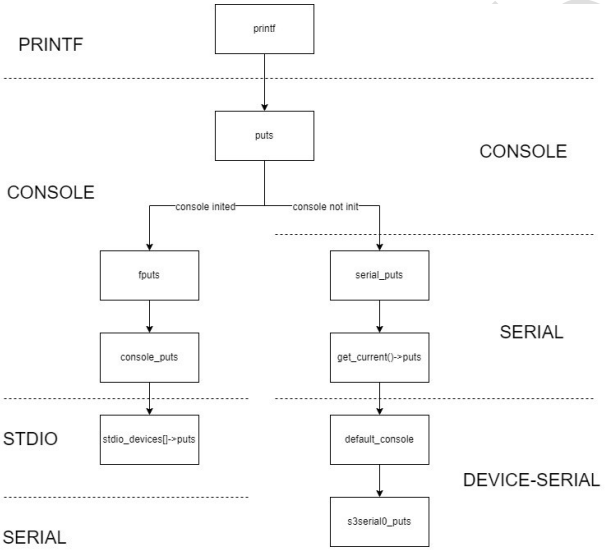
```
171  static inline void console_puts(int file, const char *s)
172  {
173      stdio_devices[file]->puts(stdio_devices[file], s);
174  }
```

而 serial_puts()在前面分析过，最终会执行：

```
497  void serial_puts(const char *s)
498  {
499      get_current()->puts(s);
500  }
501
```

```
168  static void _serial_puts(const char *s, const int dev_index)
169  {
170      while (*s) {
171          _serial_putc(*s++, dev_index);
172      }
173  }
174
175  static inline void serial_puts_dev(int dev_index, const char *s)
176  {
177      _serial_puts(s, dev_index);
178  }
```

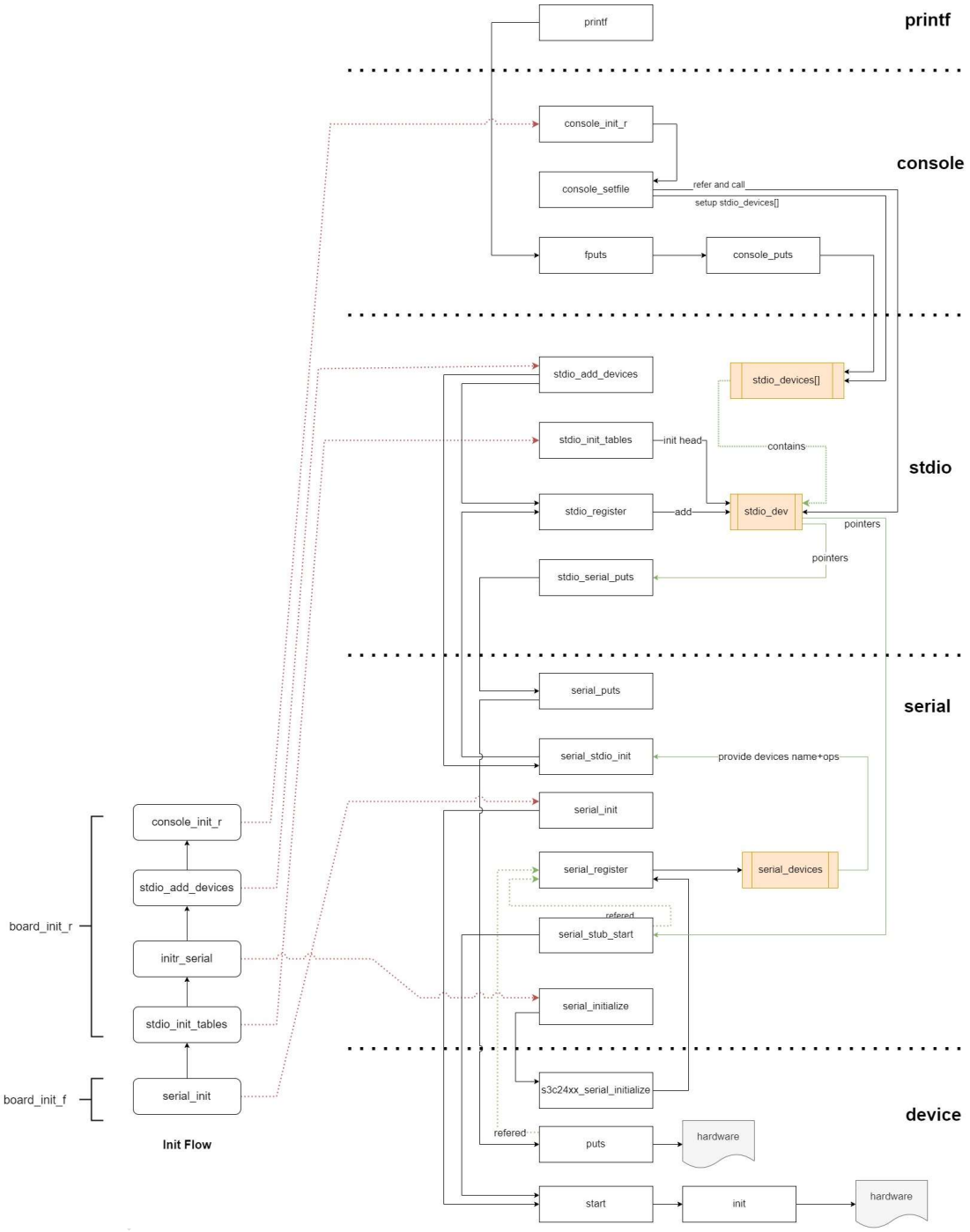
从执行过程的角度去分析打印这间简单的事情，u-boot 里牵扯到了多个功能组件，如 console，stdio 等，稍显混乱，重新整理下：



执行的过程中牵扯到多个软件层面的不同实体的交互，printf<->console<->stdio<->serial<->hardware, 而在 console 中，根据软件初始化的状态决定是利用 console->stdio->serial 这样的抽象层去使用串口（通过调用 stdio_devices[]->puts），或者直接调用 serial 部分的 API 从而执行硬件的打印函数。

通过后者直接调用的，其行为简单很多，串口的最早初始化是在 board_init_f 早期的 serial_init()中实现的。

而使用 stdio 方式实现打印的就相对要复杂的多了，其初始化流程和各个软件层的交互示意，左侧为初始化的流程，右侧为各个模块之间的调用或者索引关系：



补充：判断宏定义是否使能

在 include/linux/kconfig.h 下，U-Boot 引用了来自 Linux 的一些宏定义，用来检查一些宏是否被使能：

```
/*
 * Getting something that works in C and CPP for an arg that may or may
 * not be defined is tricky.  Here, if we have "#define CONFIG_BOOGER 1"
 * we match on the placeholder define, insert the "0," for arg1 and generate
 * the triplet (0, 1, 0).  Then the last step cherry picks the 2nd arg (a one).
 * When CONFIG_BOOGER is not defined, we generate a (... 1, 0) pair, and when
 * the last step cherry picks the 2nd arg, we get a zero.
 */
#define __ARG_PLACEHOLDER_1 0,
#define config_enabled(cfg) _config_enabled(cfg)
#define _config_enabled(value) __config_enabled(__ARG_PLACEHOLDER_##value)
#define __config_enabled(arg1_or_junk) ___config_enabled(arg1_or_junk 1, 0)
#define ___config_enabled(__ignored, val, ...) val

```

这个地方较为巧妙，当 C 语言中判断一个宏定义是否被定义，可以用#ifdef, #if defined(), 但是如果要判断一个宏定义是否为某个值，则用的是 #if (xxx==1)。

但是当使用#if (xxx==1)时，则 xxx 必须被定义有具体值，比如“#define xxx 1”，如果定义为”#define xxx”则编译器会报错。

有意思的是如果写成 #if (defined(xxx) && (xxx == 1))，有些编译器就不会报错，但是 GCC 仍然会报错。

而 U-Boot/Linux 提供的这组宏函数可以解决这样的问题，只有当这个被检测的宏被定义为 1 时才会返回 1，否则，无论是定义为一个空或者其他任何数值，都会返回 0。

这里面巧妙的地方是利用了 __ARG_PLACEHOLDER_1 为”0,”。

假设有三种情况：

第一种 #define testm 1

config_enabled(testm) ➔ _config_enabled(1) ➔__config_enabled(__ARG_PLACEHOLDER_1) ➔ __config_enabled(0,) ➔ ___config_enabled(0, 1, 0) ➔ 1

第二种 #define testm

config_enabled(testm) ➔ _config_enabled() ➔__config_enabled(__ARG_PLACEHOLDER_) ➔ __config_enabled(__ARG_PLACEHOLDER_) ➔ ___config_enabled(__ARG_PLACEHOLDER_ 1, 0) ➔ 0

第三种 #define testm 0

config_enabled(testm) ➔ _config_enabled(0) ➔__config_enabled(__ARG_PLACEHOLDER_0) ➔ __config_enabled(__ARG_PLACEHOLDER_0) ➔ ___config_enabled(__ARG_PLACEHOLDER_0, 1, 0) ➔ 0

补充：U-Boot 中的版本号和编译时间管理

U-Boot 打印版本和编译时间等信息时，会引用：

```
16  const char __weak version_string[] = U_BOOT_VERSION_STRING;
17
```

```
12
13  #ifndef DO_DEPS_ONLY
14  #include "generated/version_autogenerated.h"
15  #endif
16
17  #ifndef CONFIG_IDENT_STRING
18  #define CONFIG_IDENT_STRING ""
19  #endif
20
21  #define U_BOOT_VERSION_STRING U_BOOT_VERSION " (" U_BOOT_DATE " - " \
22      U_BOOT_TIME " " U_BOOT_TZ ")" CONFIG_IDENT_STRING
23
```

而 U_BOOT_VERSION 则来自于 version_autogenerated.h, U_BOOT_DATE 等则来自于 timestamp_autogenerated.h

version/timestamp_autogenerated.h 的生成：

前面分析过 U-Boot 编译时的各种依赖，其中：

```
1259  prepare1: prepare2 $(version_h) $(timestamp_h) \
1260                     include/config/auto.conf
```

```
422  version_h := include/generated/version_autogenerated.h
423  timestamp_h := include/generated/timestamp_autogenerated.h
424
```

而两个目标的具体执行则为：

```
1321  $(version_h): include/config/uboot.release FORCE
1322      $(call filechk,version.h)
1323
1324  $(timestamp_h): $(srctree)/Makefile FORCE
1325      $(call filechk,timestamp.h)
1326
```

执行函数 scripts/Kbuild.include: filechk:

```
define filechk
    $(Q)set -e;                 \
    $(kecho) '  CHK     $@';        \
    mkdir -p $(dir $@);         \
    $(filechk_$(1)) < $< > $@.tmp;      \
    if [ -r $@ ] && cmp -s $@ $@.tmp; then  \
        rm -f $@.tmp;           \
    else                        \
        $(kecho) '  UPD     $@';    \
        mv -f $@.tmp $@;        \
    fi
endef
```

也就是$(call filechk,version.h)则要执行 filechk_version.h, $(call filechk,timestamp.h)执行 filechk_timestamp.h, 生成对应的.tmp 文件并比较二者，如果不同则用新的.tmp 替换上次生成的头文件：

```
define filechk_version.h
    (echo \#define PLAIN_VERSION \"$(UBOOTRELEASE)\"; \
    echo \#define U_BOOT_VERSION \"U-Boot \" PLAIN_VERSION; \
    echo \#define CC_VERSION_STRING \"$$($(CC) --version | head -n 1)\"; \
    echo \#define LD_VERSION_STRING \"$$($(LD) --version | head -n 1)\"; )
endef
```

并更新到 filechk 中的$@, 也就是 Makefile 中 version_h 定义的 include/generated/version_autogenerated.h 和 timestamp_h 定义的 include/generated/timestamp_autogenerated.h

补充：Nor Flash 初始化

由于 Nor Flash 各个厂家的产品繁多，不同厂家之间 Flash 的特点也不近相同，为了便于统一，JEDEC 提出了 CFI 的标准接口，在特定的地址，按照特定的顺序写入特定的内容，可以对 Flash 芯片发送一些标准命令和读取 ID, 这项主控程序就能判断正在操作的 FLASH 的生产厂家和具体型号，进而更好的实现驱动。所以在 Nor Flash 初始化的早期，需要读取到 Flash 芯片的 ID。

Nor Flash 初始化的入口：

```
#if !defined(CONFIG_SYS_NO_FLASH)
static int initr_flash(void)
{
    ulong flash_size = 0;
    bd_t *bd = gd->bd;

    puts("Flash: ");

    if (board_flash_wp_on())
        printf("Uninitialized - Write Protect On\n");
    else
        flash_size = flash_init();

    print_size(flash_size, "");
```

flash_init 中首先尝试获取 non-CFI flash 信息：

```
    /* Init: no FLASHes known */
    for (i = 0; i < CONFIG_SYS_MAX_FLASH_BANKS; ++i) {
        flash_info[i].flash_id = FLASH_UNKNOWN;

        /* Optionally write flash configuration register */
        cfi_flash_set_config_reg(cfi_flash_bank_addr(i),
                                 cfi_flash_config_reg(i));

        if (!flash_detect_legacy(cfi_flash_bank_addr(i), i))
            flash_get_size(cfi_flash_bank_addr(i), i);
        size += flash_info[i].size;
        if (flash_info[i].flash_id == FLASH_UNKNOWN) {
#ifndef CONFIG_SYS_FLASH_QUIET_TEST
            printf ("## Unknown flash on Bank %d "
                "- Size = 0x%08lx = %ld MB\n",
                i+1, flash_info[i].size,
                flash_info[i].size >> 20);
#endif /* CONFIG_SYS_FLASH_QUIET_TEST */
        }
```

而 flash_detect_lagacy 中，board_flash_get_lagacy 位于板级驱动 smdk2410.c 中，配置了板子上具体用到的 flash 的位宽等信息，

```
    if (board_flash_get_legacy(base, banknum, info)) {
        /* board code may have filled info completely. If not, we
           use JEDEC ID probing. */
```

随后构造 unlock sequence 并读取 flash size:

```
if (info->portwidth == FLASH_CFI_8BIT
    && info->interface == FLASH_CFI_X8X16) {
    info->addr_unlock1 = 0x2AAA;
    info->addr_unlock2 = 0x5555;
} else {
    info->addr_unlock1 = 0x5555;
    info->addr_unlock2 = 0x2AAA;
}
flash_read_jedec_ids(info);
```

这里有一点迷惑了一会，根据 flash 的手册，其命令大概的格式：

Command Definitions

Table 9. Am29LV160D Command Definitions

Command Sequence (Note 1)			Cycles	Bus Cycles (Notes 2–5)											
				First		Second		Third		Fourth		Fifth		Sixth	
				Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Read (Note 6)			1	RA	RD										
Reset (Note 7)			1	XXX	F0										
Autoselect (Note 8)	Manufacturer ID	Word	4	555	AA	2AA	55	555	90	X00	01				
		Byte		AAA		555		AAA							
	Device ID, Top Boot Block	Word	4	555	AA	2AA	55	555	90	X01	22C4				
		Byte		AAA		555		AAA		X02	C4				
	Device ID, Bottom Boot Block	Word	4	555	AA	2AA	55	555	90	X01	2249				
		Byte		AAA		555		AAA		X02	49				
	Sector Protect Verify (Note 9)	Word	4	555	AA	2AA	55	555	90	(SA) X02	XX00 XX01				
		Byte		AAA		555		AAA		(SA) X04	00 01				

获取 MID 的顺序是 0x555=0xaa → 0x2aa=0x55 → 0x555 = 0x90

但是代码中用到的定义则是 0x2AAA, 似乎并不匹配，研究了一下发现手册中同时提到：

4. *Data bits DQ15–DQ8 are don't cares for unlock and command cycles.*
5. *Address bits A19–A11 are don't cares for unlock and command cycles, unless SA or PA required.*
6. *No unlock or command cycles required when reading array data.*

11. T… B…
12. T… re… m…
13. T…

那么 0x2AAA 忽略 A19-A11, 就成了 0x2AA!

```
1686  static void cmdset_amd_read_jedec_ids(flash_info_t *info)
1687  {
1688      ushort bankId = 0;
1689      uchar  manuId;
1690
1691      flash_write_cmd(info, 0, 0, AMD_CMD_RESET);
1692      flash_unlock_seq(info, 0);
1693      flash_write_cmd(info, 0, info->addr_unlock1, FLASH_CMD_READ_ID);
1694      udelay(1000); /* some flash are slow to respond */
```

首先构造 RESET 命令:

Command Definitions

Table 9. Am29LV160D Command Definitions

Command Sequence (Note 1)		Cycles	Bus Cycles (Notes 2–5)											
			First		Second		Third		Fourth		Fifth		Sixth	
			Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Read (Note 6)		1	RA	RD										
Reset (Note 7)		1	XXX	F0										
Manufacturer ID	Word	4	555	AA	2AA	55	555	90	X00	01				

发送 0x555,0x2aa, 0x555 序列读取 Manufacturer ID 和 Device ID:

Table 9. Am29LV160D Command Definitions

Command Sequence (Note 1)			Cycles	Bus Cycles (Notes 2–5)											
				First		Second		Third		Fourth		Fifth		Sixth	
				Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Read (Note 6)			1	RA	RD										
Reset (Note 7)			1	XXX	F0										
Autoselect (Note 8)	Manufacturer ID	Word	4	555	AA	2AA	55	555	90	X00	01				
		Byte		AAA		555		AAA							
	Device ID, Top Boot Block	Word	4	555	AA	2AA	55	555	90	X01	22C4				
		Byte		AAA		555		AAA		X02	C4				
	Device ID, Bottom Boot Block	Word	4	555	AA	2AA	55	555	90	X01	2249				
		Byte		AAA		555		AAA		X02	49				

```
flash_unlock_seq(info, 0);
flash_write_cmd(info, 0, info->addr_unlock1, FLASH_CMD_READ_ID);
udelay(1000); /* some flash are slow to respond */

manuId = flash_read_uchar (info, FLASH_OFFSET_MANUFACTURER_ID);
/* JEDEC JEP106Z specifies ID codes up to bank 7 */
```

```
info->device_id = flash_read_word (info,
                FLASH_OFFSET_DEVICE_ID);
if ((info->device_id & 0xff) == 0x7E) {
    /* AMD 3-byte (expanded) device ids */
    info->device_id2 = flash_read_uchar (info,
                FLASH_OFFSET_DEVICE_ID2);
    info->device_id2 <<= 8;
    info->device_id2 |= flash_read_uchar (info,
                FLASH_OFFSET_DEVICE_ID3);
}
```

读取完 ID 后，执行 jedec_flash_match, 这个函数会搜索预编译好的 jedec_table[]用来匹配 MID 和 DID：

```
493 int jedec_flash_match(flash_info_t *info, ulong base)
494 {
495     int ret = 0;
496     int i;
497     ulong mask = 0xFFFF;
498     if (info->chipwidth == 1)
499         mask = 0xFF;
500
501     for (i = 0; i < ARRAY_SIZE(jedec_table); i++) {
502         if ((jedec_table[i].mfr_id & mask) == (info->manufacturer_id & mask) &&
503             (jedec_table[i].dev_id & mask) == (info->device_id & mask)) {
504             fill_info(info, &jedec_table[i], base);
505             ret = 1;
506             break;
507         }
508     }
509     return ret;
510 }
```

jedec_table 的信息在 U-Boot 移植的 NorFlash 中介绍过：

```
{
    .mfr_id     = (u16)AMD_MANUFACT,
    .dev_id     = AM29LV160DB,
    .name       = "AMD AM29LV160DB",
    .uaddr      = {
        [1] = MTD_UADDR_0x0555_0x02AA /* x16 */
    },
    .DevSize    = SIZE_2MiB,
    .CmdSet     = CFI_CMDSET_AMD_LEGACY,
    .NumEraseRegions= 4,
    .regions    = {
        ERASEINFO(0x04000,1),
        ERASEINFO(0x02000,2),
        ERASEINFO(0x08000,1),
        ERASEINFO(0x10000,31),
    }
}
```

由于匹配到了 MID 和 DID，最后会通过 fill_info 把这些在 jedec_table 中预先配置的信息配置到全局结构体 flash_info 中。

补充：环境变量初始化

环境变量部分是 u-boot 的关键部分之一，组成了其运转必不可少的启动命令参数读取、解析过程和 hush shell 命令的构造、解析。

env 的初始化分成两部分，第一部分在 board_init_f 中调用 env_init, 将 gd->env_valid 置为 1：

```
115 #else /* ENV_IS_EMBEDDED || CONFIG_NAND_ENV_DST */
116     gd->env_addr    = (ulong)&default_environment[0];
117     gd->env_valid   = 1;
118 #endif /* ENV_IS_EMBEDDED || CONFIG_NAND_ENV_DST */
```

第二部分则是当程序搬移至 SDRAM 后，执行 board_init_r，调用 initr_env():

```
494 static int initr_env(void)
495 {
496     /* initialize environment */
497     if (should_load_env())
498         env_relocate();
499     else
500         set_default_env(NULL);
```

由于 should_load_env()总是返回 1，所以 env_relocate()总是被执行：

```
252 void env_relocate(void)
253 {
254 #if defined(CONFIG_NEEDS_MANUAL_RELOC)
255     env_reloc();
256     env_htab.change_ok += gd->reloc_off;
257 #endif
258     if (gd->env_valid == 0) {
259 #if defined(CONFIG_ENV_IS_NOWHERE) || defined(CONFIG_SPL_BUILD)
260         /* Environment not changable */
261         set_default_env(NULL);
262 #else
263         bootstage_error(BOOTSTAGE_ID_NET_CHECKSUM);
264         set_default_env("!bad CRC");
265 #endif
266     } else {
267         env_relocate_spec();
268     }
269 }
```

这里 gd_env_valid 在 board_init_f 阶段被初始化为 1，尽管中间 U-Boot 进行了一次 relocate, 但是 U-Boot 仍然将原始的 gd 结构体中的信息拷贝到了新的 new_gd 所在的位置。也就是说 gd->env_valid 这个 flag 也仍然保持为 1，所以此处会执行 env_relocate_spec()。

这个函数在 env_nand.c 中，其主要过程分为两个分支：

```
402
403     ret = readenv(CONFIG_ENV_OFFSET, (u_char *)buf);
404     if (ret) {
405         set_default_env("!readenv() failed");
406         return;
407     }
408
409     env_import(buf, 1);
410 #endif /* ! ENV_IS_EMBEDDED */
```

readenv 是从 Nand flash 中的固定偏移位置中读取信息到内存中，如果无法从 flash 中读取足够数量的数据，那么将会报错并执行 set_default_env，加载 env_default.h 中定义的 default_environment[] 信息，并执行 import。

如果读取到了足够的信息，则执行 env_import, 首先检查 CRC，如果 CRC 检查不通过，则报错并执行 set_default_env()。如果 CRC 检查通过，则当前的数据合法，则开始执行 himport_r():

```
215     if (himport_r(&env_htab, (char *)ep->data, ENV_SIZE, '\0', 0, 0,
216             0, NULL)) {
217         gd->flags |= GD_FLG_ENV_READY;
218         return 1;
219     }
```

这个函数的主要作用就是扫描 key 和 data, 比如”bootdelay=5”，key=”bootdelay”, data=”5”, 并把这些信息都存储在全局结构体 env_htab 中：

```
26 struct hsearch_data env_htab = {
27     .change_ok = env_flags_validate,
28 };
```

组合后，其大概结构：

```
struct hsearch_data {
    struct _ENTRY {
        int used;          ===> occupied, 0=not used, -1=deleted, other indicate hash
        typedef struct
        {
            char *key;     ===> key string
            void *data;    ===> data string
        } entry;
    } table[hash_table_size]; ===> items array
    unsigned int size;     ===> number of hash table items
    unsigned int filled;   ===> how many items were used
    int (*change_ok)() ------> env_flags_validate()
};
```

而 env_htab 中保存这些信息的方式是哈希表，这样用来平衡内存消耗和搜索时间（如果内存无限大，可以让 name 直接成为数组的索引，这样几乎没有查找时间，如果保持内存最小，每个条目顺序存储，则需要顺序查找，时间为 O(n)）。

不过此处疑问是，整体看环境变量的数量并不多，一般也就是几十个，就算按照 256 个条目甚至 512 个条目来算，遍历这样的数组代价并不是很大，构造和维护哈希表是否值得？

在 himport_r 中，将会构造存储这些环境变量的键和键值，首先计算哈希表的大小，申请空间并初始化这张表：

```
331     if (!htab->table) {
332         int nent = CONFIG_ENV_MIN_ENTRIES + size / 8;
333
334         if (nent > CONFIG_ENV_MAX_ENTRIES)
335             nent = CONFIG_ENV_MAX_ENTRIES;
336
337         debug("Create Hash Table: N=%d\n", nent);
338
339         if (hcreate_r(nent, htab) == 0) {
340             free(data);
341             return 0;
342         }
343     }
```

值得注意的是，首次计算出来的表的大小(nent)并不等于真正创建的表的大小，在 hcreate_r()中，表的大小被扩展成为素数 + 1：

```
    /* Change nel to the first prime number not smaller as nel. */
    nel |= 1;       /* make odd */
    while (!isprime(nel))
        nel += 2;

    htab->size = nel;
    htab->filled = 0;

    /* allocate memory and zero out */
    htab->table = (_ENTRY *) calloc(htab->size + 1, sizeof(_ENTRY));
    if (htab->table == NULL)
        return 0;

```

申请了哈希表的空间后，开始从整体的 buffer 中搜索'name=value'这样的配对，value 可以是空，每个键值的分隔用换行符表示。其格式就是 env_default.h 中的样式：

```
    ENV_FLAGS_VAR "=" CONFIG_ENV_FLAGS_LIST_DEFAULT "\0"
#endif
#ifdef  CONFIG_BOOTARGS
    "bootargs=" CONFIG_BOOTARGS         "\0"
#endif
#ifdef  CONFIG_BOOTCOMMAND
    "bootcmd="  CONFIG_BOOTCOMMAND      "\0"
#endif
#ifdef  CONFIG_RAMBOOTCOMMAND
    "ramboot="  CONFIG_RAMBOOTCOMMAND       "\0"
#endif
```

搜索到键和键值串后，进行下一步的 parser 并把这个键加入到哈希表中：

```
        /* enter into hash table */
        e.key = name;
        e.data = value;

        hsearch_r(e, ENTER, &rv, htab, flag);
```

在 hsearch_r 中首先构造第一个哈希键值，u-boot 中用到的就是简单的左移 4 然后自加键值中的字符的值：

```
    /* Compute an value for the given string. Perhaps use a better method. */
    hval = len;
    count = len;
    while (count-- > 0) {
        hval <<= 4;
        hval += item.key[count];
    }

    /*
     * First hash function:
     * simply take the modul but prevent zero.
     */
    hval %= htab->size;
    if (hval == 0)
        ++hval;

    /* The first index tried. */
    idx = hval;
```

但是这样算出来的不同的键的哈希值有可能是重复的，那么在获得第一个哈希值后，根据 used 来判断这个哈希表对应位置是否被使用了：

如果被使用，则进一步在_compare_and_overwrite_entry()中判断是否被同名的键所使用，若是则进行替换，若不是则计算第二个哈希并根据这个哈希不断的枚举新的 index,判断是否有空余或者重复键的位置：

```
        do {
            /*
             * Because SIZE is prime this guarantees to
             * step through all available indices.
             */
            if (idx <= hval2)
                idx = htab->size + idx - hval2;
            else
                idx -= hval2;

            /*
             * If we visited all entries leave the loop
             * unsuccessfully.
             */
            if (idx == hval)
                break;

            /* If entry is found use it. */
            ret = _compare_and_overwrite_entry(item, action, retval,
                htab, flag, hval, idx);
            if (ret != -1)
                return ret;
        }
        while (htab->table[idx].used);
```

这里并不会判断这个键是否真的获取到了对应位置的哈希表，可能出现的情况就是表满了且以前没有同名的键使用过的位置，那么在后续的代码中，就会遇到哈希表满而返回错误：

```
        if (htab->filled == htab->size) {
            __set_errno(ENOMEM);
            *retval = NULL;
            return 0;
        }
```

之后由于获得了有效的哈希表的位置，那么开始拷贝键和键值到哈希表中去，并开始搜素是否回[调函数列表]中是否有对应的回调函数：

```
    htab->table[idx].used = hval;
    htab->table[idx].entry.key = strdup(item.key);
    htab->table[idx].entry.data = strdup(item.data);
    if (!htab->table[idx].entry.key ||
        !htab->table[idx].entry.data) {
        __set_errno(ENOMEM);
        *retval = NULL;
        return 0;
    }

    ++htab->filled;

    /* This is a new entry, so look up a possible callback */
    env_callback_init(&htab->table[idx].entry);
```

回调函数列表可以是在环境变量中通过变量名为".callbacks"的方式定义，或者 u-boot 中默认的定义(env_callback.h):

```
/*
 * This list of callback bindings is static, but may be overridden by defining
 * a new association in the ".callbacks" environment variable.
 */
#define ENV_CALLBACK_LIST_STATIC ENV_DOT_ESCAPE ENV_CALLBACK_VAR ":callbacks," \
    ENV_DOT_ESCAPE ENV_FLAGS_VAR ":flags," \
    "baudrate:baudrate," \
    NET_CALLBACKS \
    "loadaddr:loadaddr," \
    SILENT_CALLBACK \
    SPLASHIMAGE_CALLBACK \
    "stdin:console,stdout:console,stderr:console," \
    CONFIG_ENV_CALLBACK_LIST_STATIC
```

以 smdk2410 为例，这个静态回调函数列表会被预编译为:

\.callbacks:callbacks,\.flags:flags,baudrate:baudrate,bootfile:bootfile,ipaddr:ipaddr,gatewayip:gatewayip,netmask:netmask,serverip:serverip,nvlan:nvlan,vlan:vlan,eth\d?addr:ethaddr,loadaddr:loadaddr,stdin:console,stdout:console,stderr:console,

这个列表通过逗号分隔不同的配对，每个配对通过冒号区分名称和回调函数名，大概格式就是:

name1:callback1, name2:callback2

至于这些回调函数的具体定义后面再分析。

通过调用 env_attr_loopup(), 可以获得是否再回调函数列表中有匹配的回调函数:

```
    /* look in the ".callbacks" var for a reference to this variable */
    if (callback_list != NULL)
        ret = env_attr_lookup(callback_list, var_name, callback_name);

    /* only if not found there, look in the static list */
    if (ret)
    {
        ret = env_attr_lookup(ENV_CALLBACK_LIST_STATIC, var_name, callback_name);
    }
```

```
int env_attr_lookup(const char *attr_list, const char *name, char *attributes)
{
    ...

    priv.searched_for = name;
    priv.regex = NULL;          <== 构造搜索目标
    priv.attributes = NULL;
    retval = env_attr_walk(attr_list, regex_callback, &priv); <==   搜索回调函数列表中是否
                                                                    有定义对应的回调函数
    if (retval)
        return retval; /* error */

    if (priv.regex) {
        strcpy(attributes, priv.attributes);    <== 搜索到匹配的名称和对应的回调函数名
        ...
        /* success */
        return 0;
    }
    return -ENOENT; /* not found in list */
}
```

上面会用到两个函数，一个是 env_attr_walk，用来分隔逗号并匹配每一个键：键值对，然后将获取到的键，键值和搜索目标键名送到 regex_callback 中用来检查名称是否匹配：

```
            if (strlen(name) != 0) {
                int retval = 0;

                retval = callback(name, attributes, priv);
                if (retval) {
                    free(entry_cpy);
                    return retval;
                }
            }
```

regex_callback 中则会用到一个正则表达式的库去匹配待检查的键(name)和搜索目标(priv->searched_for)是否一致：

```c
static int regex_callback(const char *name, const char *attributes, void *priv)
{
    int retval = 0;
    struct regex_callback_priv *cbp = (struct regex_callback_priv *)priv;
    struct slre slre;
    char regex[strlen(name) + 3];

    /* Require the whole string to be described by the regex */
    sprintf(regex, "^%s$", name);
    if (slre_compile(&slre, regex)) {
        struct cap caps[slre.num_caps + 2];

        if (slre_match(&slre, cbp->searched_for,
                       strlen(cbp->searched_for), caps)) {
            free(cbp->regex);
            cbp->regex = malloc(strlen(regex) + 1);
            if (cbp->regex) {
                strcpy(cbp->regex, regex);
```

成功匹配到后会把匹配到的函数名称逐次返回到上级调用者，最后根据这个回调函数的名称搜索对应的函数指针，并把这个指针存储在哈希表中：

```c
    /* if an association was found, set the callback pointer */
    if (!ret && strlen(callback_name)) {
        clbkp = find_env_callback(callback_name);
        if (clbkp != NULL)
#if defined(CONFIG_NEEDS_MANUAL_RELOC)
            var_entry->callback = clbkp->callback + gd->reloc_off;
#else
            var_entry->callback = clbkp->callback;
#endif
    }
```

find_env_calback()函数的实现：

```c
static struct env_clbk_tbl *find_env_callback(const char *name)
{
    struct env_clbk_tbl *clbkp;
    int i;
    int num_callbacks = ll_entry_count(struct env_clbk_tbl, env_clbk);
```

首先执行 ll_entry_count(), 这个函数有两重作用：第一个是计算回调函数结构体数组的大小，第二个是编译链接阶段定义这个数组的起始和结束标志：

```c
#define ll_entry_count(_type, _list)                    \
    ({                                                  \
        _type *start = ll_entry_start(_type, _list);    \
        _type *end = ll_entry_end(_type, _list);        \
        unsigned int _ll_result = end - start;          \
        _ll_result;                                     \
    })
```

ll_entry_start 和 ll_entry_end 会被扩展为：

```
#define ll_entry_start(_type, _list)                    \
({                                      \
    static char start[0] __aligned(4) __attribute__((unused,    \
        section(".u_boot_list_2_"#_list"_1")));         \
    (_type *)&start;                            \
})
```

```
#define ll_entry_end(_type, _list)                  \
({                                      \
    static char end[0] __aligned(4) __attribute__((unused,      \
        section(".u_boot_list_2_"#_list"_3")));         \
    (_type *)&end;                              \
})
```

也就是一个静态的的变量，被分别链接到.u_boot_list_2_env_clbk_1 和.u_boot_list_2_env_clbk_3 的位置。查看链接脚本 u-boot.lds:

```
. = ALIGN(4);
.rodata : { *(SORT_BY_ALIGNMENT(SORT_BY_NAME(.rodata*))) }
. = ALIGN(4);
.data : {
 *(.data*)
}
. = ALIGN(4);
. = .;
. = ALIGN(4);
.u_boot_list : {
 KEEP(*(SORT(.u_boot_list*)));
}
. = ALIGN(4);
.__efi_runtime_start : {
```

也就是所有名称前缀为.u_boot_list 的链接目标，都要按照排序被放在.u_boot_list 这个 section 内。

而具体的命令或者回调函数是如何定义的呢？再 linker_lists.h 中：

```
#define ll_entry_declare(_type, _name, _list)               \
    _type _u_boot_list_2_##_list##_2_##_name __aligned(4)       \
            __attribute__((unused,              \
            section(".u_boot_list_2_"#_list"_2_"#_name)))
```

命令被定义为.u_boot_list_2_xxx_2_yyy, 比如具体来说, env_callabck.h 中定义：

```
#define U_BOOT_ENV_CALLBACK(name, callback) \
    ll_entry_declare(struct env_clbk_tbl, name, env_clbk) = \
    {#name, callback}
#endif
```

对于具体的命令：

```
95  }
96  U_BOOT_ENV_CALLBACK(baudrate, on_baudrate);
97
```

就会被扩展为：

```
7  struct env_clbk_tbl {
8      const char *name;       /* Callback name */
9      int (*callback)(const char *name, const char *value, enum env_op op, int flags);
10 };
11
12
13 struct env_clbk_tbl     \
14         _u_boot_list_2_env_clbk_2_baudrate \
15             __aligned(4) __attribute__((unused, section(".u_boot_list_2_env_clbk_2_baudrate")))
16     {"baudrate", on_baudrate};
```

这个名为_u_boot_list_2_env_clbk_2_baudrate 的结构体会被链接器链接到.u_boot_list section 中，编译后查看 u-boot.map：

```
5648 .u_boot_list_2_env_clbk_1
5649                 0x00078b20        0x0 common/built-in.o
5650 .u_boot_list_2_env_clbk_2_baudrate
5651                 0x00078b20        0x8 drivers/serial/built-in.o
5652                 0x00078b20                _u_boot_list_2_env_clbk_2_baudrate
5653 .u_boot_list_2_env_clbk_2_bootfile
5654                 0x00078b28        0x8 net/built-in.o
5655                 0x00078b28                _u_boot_list_2_env_clbk_2_bootfile
5656 .u_boot_list_2_env_clbk_2_callbacks
5657                 0x00078b30        0x8 common/built-in.o
5658                 0x00078b30                _u_boot_list_2_env_clbk_2_callbacks
5659 .u_boot_list_2_env_clbk_2_console
5660                 0x00078b38        0x8 common/built-in.o
5661                 0x00078b38                _u_boot_list_2_env_clbk_2_console
5662 .u_boot_list_2_env_clbk_2_ethaddr
5663                 0x00078b40        0x8 net/built-in.o
```

```
5685                 0x00078b78                _u_boot_list_2_env_clbk_2_serverip
5686 .u_boot_list_2_env_clbk_2_vlan
5687                 0x00078b80        0x8 net/built-in.o
5688                 0x00078b80                _u_boot_list_2_env_clbk_2_vlan
5689 .u_boot_list_2_env_clbk_3
5690                 0x00078b88        0x0 common/built-in.o
5691 .u_boot_list_2_part_driver_1
```

可以看到.u_boot_list2_env_clbk_1 为起始位置，而所有命令都被放到了它的后面，最后则是.u_boot_list_2_env_clbk_3。

所以通过这样的 linker-list 的方式，可以再各个功能块内申明各自的回调函数和结构体，会被链接器自动的链接至同一个 section 中，从 C 代码的实现看是离散的，但是实际物理上的访问却可以是连续的。

所以再 find_env_callback 中，首先通过 ll_entry_count，计算.u_boot_list_2_env_1 -> .u_boot_list2_env_3 之间实际的大小，

然后通过搜索匹配寻找回调函数结构体和里面的指针：

```
    /* look up the callback in the linker-list */
    for (i = 0, clbkp = ll_entry_start(struct env_clbk_tbl, env_clbk);
         i < num_callbacks;
         i++, clbkp++) {
        if (strcmp(name, clbkp->name) == 0)
            return clbkp;
    }
```

再搜索完回调函数后，通过同样的方法搜索环境变量或默认 flags_list 中的".flags"。

随后进行一次 permission check, 判断当前的插入操作是否违反 flag 中限定的行为。最后会执行所匹配到的回调函数，例如如果环境变量中定义了"baudrate", 那么就会执行对应的回调函数：

```
static int on_baudrate(const char *name, const char *value, enum env_op op,
    int flags)
{
```

最后，以 baudrate 为例，回调函数匹配的大致过程：

1. 链接器把结构体放在对应的段内：

   {.name = "baudrate", .callback=on_baudrate}

2. flash 中读取带 CRC 信息的环境变量表，匹配后把数据送入 himport_r(), 这个数据中包含了"baudrate=115200"

3. env_callback_init()中获得了"baudrate:baudrate" callback list, 名称匹配上后，进一步搜索名字为"baudrate"的回调函数结构体

4. 最后搜索到 1 中定义的结构体，获得回调函数指针 on_baudrate

5. 最后执行 on_baudrate("baudrate", "115200", 0, 0)

补充：网卡初始化

board_r.c 中通过调用 eth_initialize()开始进行网卡的初始化，这个函数位于 eth_legacy.c 中：

首先执行：

```
    if (board_eth_init != __def_eth_init) {
        if (board_eth_init(gd->bd) < 0)
            printf("Board Net Initialization Failed\n");
    } else if (cpu_eth_init != __def_eth_init) {
        if (cpu_eth_init(gd->bd) < 0)
            printf("CPU Net Initialization Failed\n");
    } else {
        printf("Net Initialization Skipped\n");
    }
```

U-Boot 中默认定义了：

```
static int __def_eth_init(bd_t *bis)
{
    return -1;
}
int cpu_eth_init(bd_t *bis) __attribute__((weak, alias("__def_eth_init")));
int board_eth_init(bd_t *bis) __attribute__((weak, alias("__def_eth_init")));

```

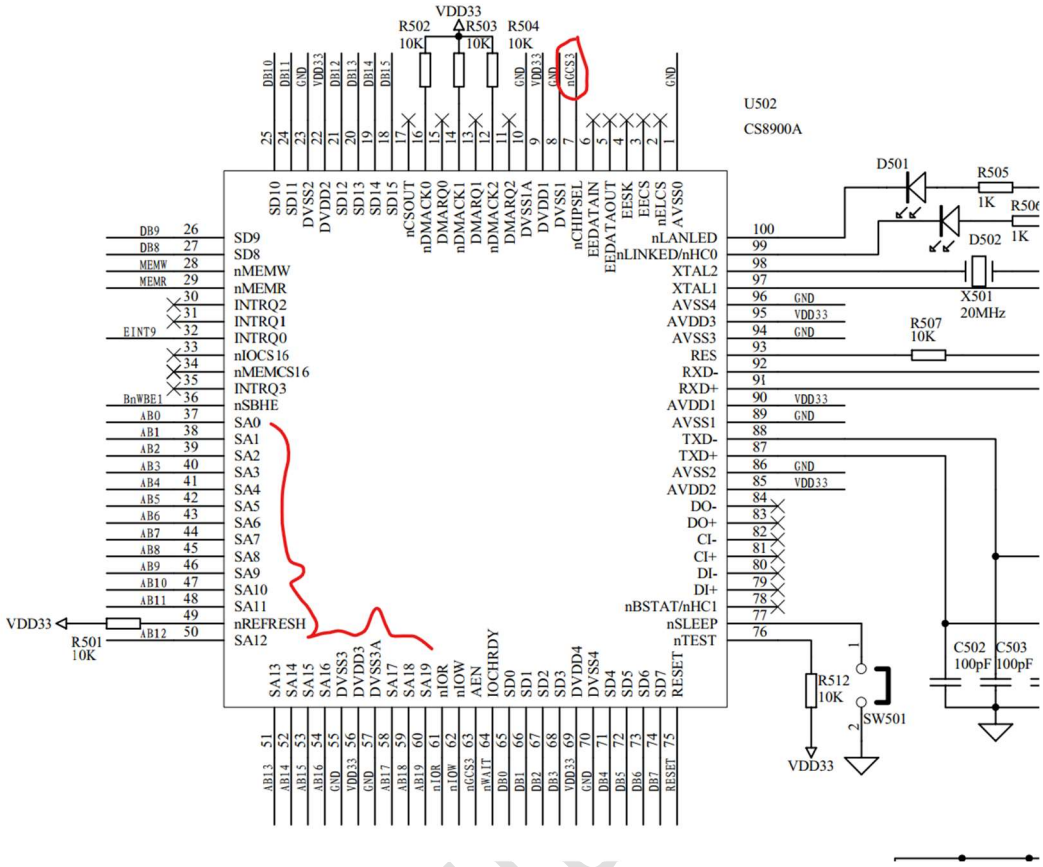
smdk2410 开发板上使用的 CS8900A 网卡，并在 smdk2410.c 中定义了 board_eth_init()：

```
#ifdef CONFIG_CMD_NET
int board_eth_init(bd_t *bis)
{
    int rc = 0;
#ifdef CONFIG_CS8900
    rc = cs8900_initialize(0, CONFIG_CS8900_BASE);
#endif
    return rc;
}
#endif
```

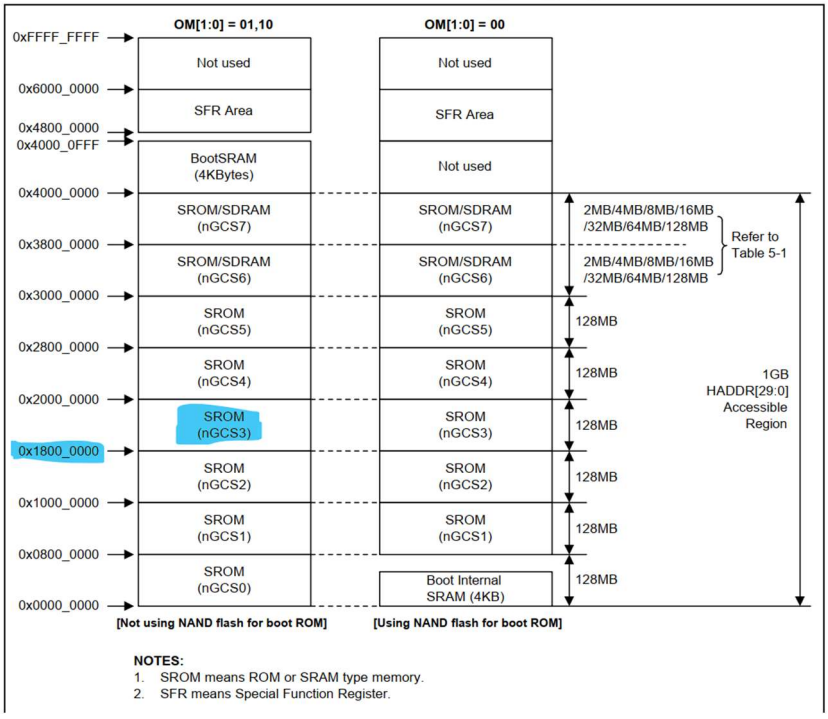
在 smdk2410.h 中，CONFIG_CS8900_BASE 被定义为：

```
#define CONFIG_CS8900       /* we have a CS8900 on-board */
#define CONFIG_CS8900_BASE  0x19000300
#define CONFIG_CS8900_BUS16 /* the Linux driver does accesses as shorts */
```

根据板子的原理图：



CS8900A 被挂载在总线上，并使用 nGCS3 为片选，根据 s3c2410 手册：



CS8900A 所在的地址空间为 0x1800_0000, 但是怎么基地址被定义为了 0x1900_0300?

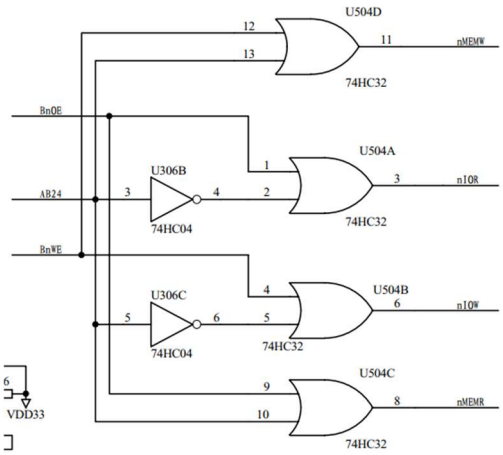
一，根据 CS8900A 手册：

4.10 I/O Space Operation

In I/O Mode, PacketPage memory is accessed through eight 16-bit I/O ports that are mapped into 16 contiguous I/O locations in the host system's I/O space. I/O Mode is the default configuration for the CS8900A and is always enabled. On power up, the default value of the I/O base address is set at 300h. (Note that 300h is typically assigned to LAN peripherals). The I/O base address may be changed to any available XXX0h location, either by loading configuration data from the EEPROM, or during system setup. Table 17 shows the CS8900A I/O Mode mapping.

Offset	Type	Description
0000h	Read/Write	Receive/Transmit Data (Port 0)
0002h	Read/Write	Receive/Transmit Data (Port 1)

二，原理图上，A24 用来驱动 nMEMW, nIOR 等信号：



所以 CS8900A 工作是的基地址就是 0x1800_0000 + 0x0100_0000 + 0x300 = 0x1900_0300

后面对于网络设备的初始化，过程比较直接，不再详细记录，基本流程就是填充连个结构体：

```
288     struct eth_device *dev;
289     struct cs8900_priv *priv;
```

填入对应的函数指针：

```c
struct eth_device {
    char name[16];
    unsigned char enetaddr[6];
    phys_addr_t iobase;
    int state;

    int (*init)(struct eth_device *, bd_t *);
    int (*send)(struct eth_device *, void *pac
    int (*recv)(struct eth_device *);
    void (*halt)(struct eth_device *);
#ifdef CONFIG_MCAST_TFTP
    int (*mcast)(struct eth_device *, const u8
#endif
    int (*write_hwaddr)(struct eth_device *);
    struct eth_device *next;
    int index;
    void *priv;
};
```

```c
memset(priv, 0, sizeof(*priv));
priv->regs = (struct cs8900_regs *)base_addr;

dev->iobase = base_addr;
dev->priv = priv;
dev->init = cs8900_init;
dev->halt = cs8900_halt;
dev->send = cs8900_send;
dev->recv = cs8900_recv;
```

然后根据 CS8900A 手册上的时许构造命令对网卡进行控制。

补充：SPL

TBD

U-Boot 加载 Linux

为了分析 Linux 内核的启动，首先应该直到 U-Boot 加载 Kernel Image 的格式，常见的 Linux Image 大概有几种：

vmlinux: 最原始的 linux elf 文件合集, 这些文件包含了符号表和调试等信息

Kernel Image: 经过 objcopy 后的 vmlinux

zImage: 经过压缩的 Kernel Image

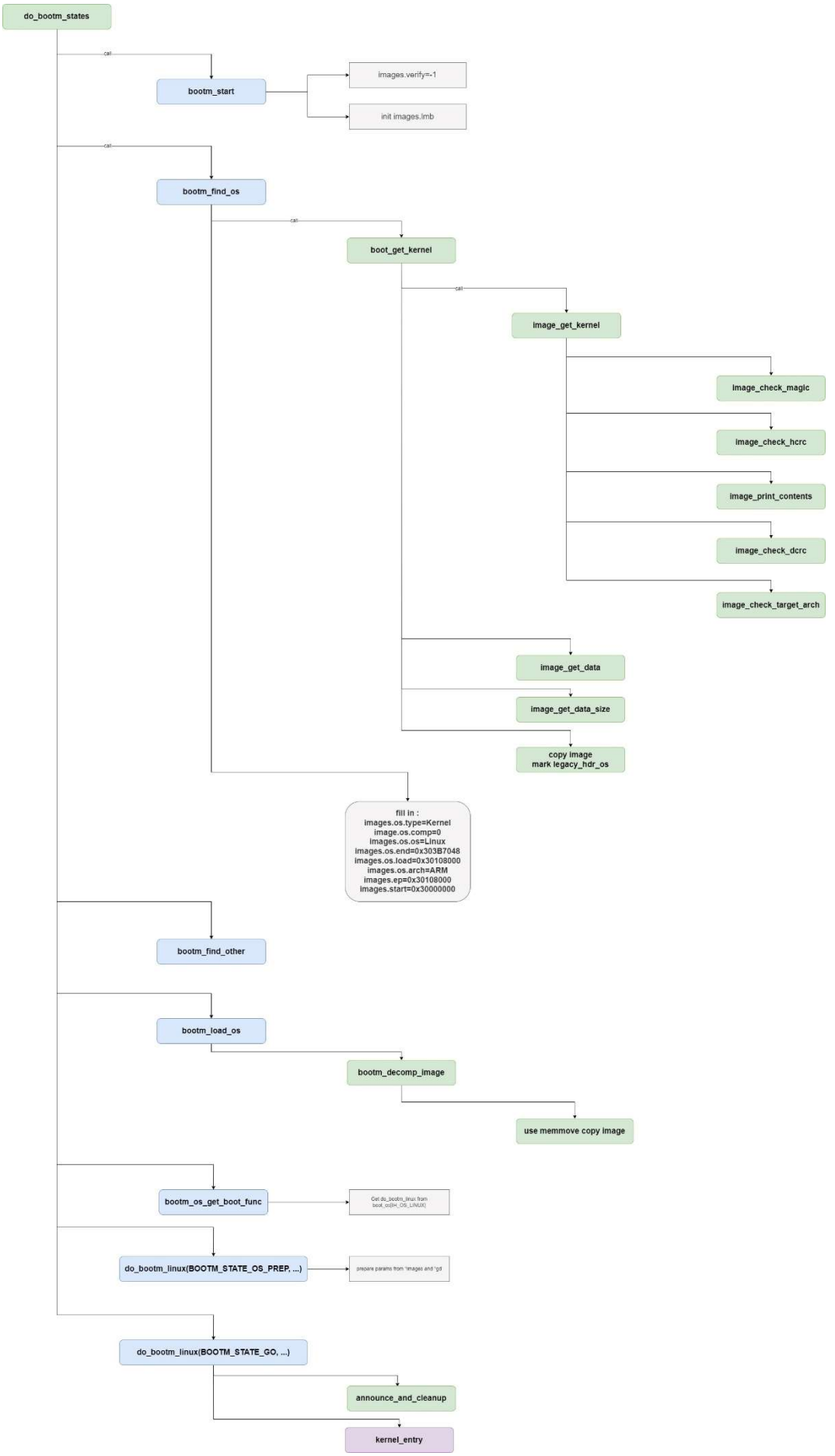
uImage: 用于 U-Boot 启动的 Kernel Image, 在 zImage 前加上了 64 字节的头,包含了 U-Boot 加载 Linux 需要的一些基本信息,比如加载位置, 大小等的。

在 SMDK2410 中，内核的编译最终会生成 uImage。

根据前面的分析，U-Boot 在主函数中最终会通获取环境变量 bootcmd 中的命令，并最终执行"bootm 0x30000000"，跳转到 do_bootm 函数中开始最后的内核引导过程：

```c
    return do_bootm_states(cmdtp, flag, argc, argv, BOOTM_STATE_START |
        BOOTM_STATE_FINDOS | BOOTM_STATE_FINDOTHER |
        BOOTM_STATE_LOADOS |
#if defined(CONFIG_PPC) || defined(CONFIG_MIPS)
        BOOTM_STATE_OS_CMDLINE |
#endif
        BOOTM_STATE_OS_PREP | BOOTM_STATE_OS_FAKE_GO |
        BOOTM_STATE_OS_GO, &images, 1);
```

do_bootm_state()的宏观流程如下：



宏观的流程，在 bootm_start 阶段初始化一些全局的信号，然后通过 bootm_find_os 用来判断 0x3000_0000 位置的 image 的内容，smdk2410 中 Linux Kernel 使用的 uImage, 起 64 字节的头的格式为：

```
272 typedef struct image_header {
273     __be32      ih_magic;   /* Image Header Magic Number    */
274     __be32      ih_hcrc;    /* Image Header CRC Checksum    */
275     __be32      ih_time;    /* Image Creation Timestamp */
276     __be32      ih_size;    /* Image Data Size      */
277     __be32      ih_load;    /* Data  Load  Address      */
278     __be32      ih_ep;      /* Entry Point Address      */
279     __be32      ih_dcrc;    /* Image Data CRC Checksum  */
280     uint8_t     ih_os;      /* Operating System     */
281     uint8_t     ih_arch;    /* CPU architecture     */
282     uint8_t     ih_type;    /* Image Type           */
283     uint8_t     ih_comp;    /* Compression Type     */
284     uint8_t     ih_name[IH_NMLEN];  /* Image Name       */
285 } image_header_t;
```

实际编译出的 uImage 二进制为：

```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded Text
00000000  27 05 19 56 90 D3 91 25 63 4C 41 85 00 2A F0 48  '  .  .  V  .  .  .  %  c  L  A  .  .  *  .  H
00000010  30 10 80 00 30 10 80 00 B6 7D D6 AB 05 02 02 00  0  .  .  .  0  .  .  .  .  }  .  .  .  .  .  .
00000020  4C 69 6E 75 78 2D 33 2E 31 39 2E 38 00 00 00 00  L  i  n  u  x  -  3  .  1  9  .  8  .  .  .  .
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
00000040  00 00 A0 E1 00 00 A0 E1 00 00 A0 E1 00 00 A0 E1  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
00000050  00 00 A0 E1 00 00 A0 E1 00 00 A0 E1 00 00 A0 E1  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
00000060  03 00 00 EA 18 28 6F 01 00 00 00 00 48 F0 2A 00  .  .  .  .  (  o  .  .  .  .  H  .  *  .
00000070  01 02 03 04 00 90 0F E1 01 70 A0 E1 02 80 A0 E1  .  .  .  .  .  .  .  .  p  .  .  .  .  .
00000080  00 20 0F E1 03 00 12 E3 01 00 00 1A 17 00 A0 E3  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
```

那么就可以翻译为：

Size	内容	数据	解释
4B	Magic	0x27051956	IH_MAGIC
4B	HCRC	0x90D39125	Header CRC
4B	Timestamp	0x634C4185	2022/10/17 01:38:13
4B	Size	0x2AF048	Image size
4B	Load	0x30108000	Load Address
4B	EP	0x30108000	Entry Point
1B	OS	0x05	OS_Linux
1B	Arch	0x02	ARM
1B	Type	0x02	Kernel
1B	Compression	0x0	No Compression
32B	Name	Linux-3.19.0	

在获取到了这些信息后，U-Boot 会检查 magic number 确定为 lagacy_image:

```
774  int genimg_get_format(const void *img_addr)
775  {
776  #if defined(CONFIG_IMAGE_FORMAT_LEGACY)
777      const image_header_t *hdr;
778
779      hdr = (const image_header_t *)img_addr;
780      if (image_check_magic(hdr))
781          return IMAGE_FORMAT_LEGACY;
782  #endif
```

CRC 检查：

```
728  static image_header_t *image_get_kernel(ulong img_addr, int verify)
729  {
730      image_header_t *hdr = (image_header_t *)img_addr;
731
732      if (!image_check_magic(hdr)) {
733          puts("Bad Magic Number\n");
734          bootstage_error(BOOTSTAGE_ID_CHECK_MAGIC);
735          return NULL;
736      }
737      bootstage_mark(BOOTSTAGE_ID_CHECK_HEADER);
738
739      if (!image_check_hcrc(hdr)) {
740          puts("Bad Header Checksum\n");
741          bootstage_error(BOOTSTAGE_ID_CHECK_HEADER);
742          return NULL;
743      }
744
745      bootstage_mark(BOOTSTAGE_ID_CHECK_CHECKSUM);
746      image_print_contents(hdr);
```

并通过 image_print_contents 在终端打印启动信息：

```
Image Name:   Linux-3.19.8
Created:      2022-10-16  17:38:13 UTC
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    2814024 Bytes = 2.7 MiB
Load Address: 30108000
Entry Point:  30108000
Verifying Checksum ... OK
Loading Kernel Image ... OK
```

最后获取到了关键的 load_address 和 entry_point，U-Boot 会使用 load_address 将当前 0x3000_0000 + 0x40(header offset)的地址加载到 load_adress 位置上：

```
615     /* Load the OS */
616     if (!ret && (states & BOOTM_STATE_LOADOS)) {
617         ulong load_end;
618
619         iflag = bootm_disable_interrupts();
620         ret = bootm_load_os(images, &load_end, 0);
621         if (ret == 0)
622             lmb_reserve(&images->lmb, images->os.load,
623                         (load_end - images->os.load));
624         else if (ret && ret != BOOTM_ERR_OVERLAP)
```

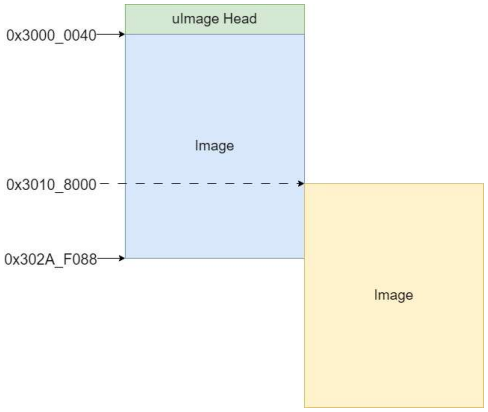
```
429     load_buf = map_sysmem(load, 0);
430     image_buf = map_sysmem(os.image_start, image_len);
431     err = bootm_decomp_image(os.comp, load, os.image_start, os.type,
432                 load_buf, image_buf, image_len,
433                 CONFIG_SYS_BOOTM_LEN, load_end);
```

```
342     switch (comp) {
343     case IH_COMP_NONE:
344         if (load == image_start)
345             break;
346         if (image_len <= unc_len)
347             memmove_wd(load_buf, image_buf, image_len, CHUNKSZ);
348         else
349             ret = 1;
```

这里是通过 GDB 调试获取的 images 结构体的内容:

```
(gdb) print/x *images
$4 = {legacy_hdr_os = 0x30000000, legacy_hdr_os_copy = {ih_magic = 0x56190527,
    ih_hcrc = 0x2591d390, ih_time = 0x85414c63, ih_size = 0x48f02a00,
    ih_load = 0x801030, ih_ep = 0x801030, ih_dcrc = 0xabd67db6, ih_os = 0x5,
    ih_arch = 0x2, ih_type = 0x2, ih_comp = 0x0, ih_name = {0x4c, 0x69, 0x6e,
      0x75, 0x78, 0x2d, 0x33, 0x2e, 0x31, 0x39, 0x2e, 0x38,
      0x0 <repeats 20 times>}}, legacy_hdr_valid = 0x1, os = {
    start = 0x30000000, end = 0x302af088, image_start = 0x30000040,
    image_len = 0x2af048, load = 0x30108000, comp = 0x0, type = 0x2, os = 0x5,
    arch = 0x2}, ep = 0x30108000, rd_start = 0x0, rd_end = 0x0, ft_addr = 0x0,
  ft_len = 0x0, initrd_start = 0x0, initrd_end = 0x0, cmdline_start = 0x0,
  cmdline_end = 0x0, kbd = 0x0, verify = 0xffffffff, state = 0x1, lmb = {
    memory = {cnt = 0x1, size = 0x0, region = {{base = 0x30000000,
          size = 0x4000000}, {base = 0x0, size = 0x0}, {base = 0x0,
          size = 0x0}, {base = 0x0, size = 0x0}, {base = 0x0, size = 0x0}, {
          base = 0x0, size = 0x0}, {base = 0x0, size = 0x0}, {base = 0x0,
          size = 0x0}, {base = 0x0, size = 0x0}}}, reserved = {cnt = 0x1,
      size = 0x0, region = {{base = 0x33b26d28, size = 0x4d92d8}, {base = 0x0,
          size = 0x0}, {base = 0x0, size = 0x0}, {base = 0x0, size = 0x0}, {
          base = 0x0, size = 0x0}, {base = 0x0, size = 0x0}, {base = 0x0,
          size = 0x0}, {base = 0x0, size = 0x0}, {base = 0x0, size = 0x0}}}}}
```

有一点值得留意，image_start 的位置是 0x3000_0040, load 的地址是 0x3010_8000, 而 image size 则是 0x2AF048, 也就是说 Kernel Image 的位置从 start 搬移到 load 的位置二者有重合:



这样启动是不会有问题的，拷贝到 load 地址中并不会破坏原有 Image 内容，因为使用的 memmove()是倒序拷贝的：

```
532 void * memmove(void * dest,const void *src,size_t count)
533 {
534     char *tmp, *s;
535
536     if (src == dest)
537         return dest;
538
539     if (dest <= src) {
540         tmp = (char *) dest;
541         s = (char *) src;
542         while (count--)
543             *tmp++ = *s++;
544         }
545     else {
546         tmp = (char *) dest + count;
547         s = (char *) src + count;
548         while (count--)
549             *--tmp = *--s;
550         }
551
552     return dest;
553 }
```

拷贝完成后，就会通过 bootm_os_get_boot_func 和 os_type 获取对应的启动函数：

```
492
493 boot_os_fn *bootm_os_get_boot_func(int os)
494 {
495     return boot_os[os];
496 }
```

```
435 static boot_os_fn *boot_os[] = {
436     [IH_OS_U_BOOT] = do_bootm_standalone,
437 #ifdef CONFIG_BOOTM_LINUX
438     [IH_OS_LINUX] = do_bootm_linux,
439 #endif
440 #ifdef CONFIG_BOOTM_NETBSD
```

在 do_bootm_linux 中也分成两步，第一步 boot_prep_linux 初始化一些关键的信息到 gd->bd->bi_boot_params 中，这些信息，包括获取到的 machine id, 最终都会当作参数传递给 Linux 内核，第二部执行 boot_jump_linux, 通过前面获取的 entry_point:

```
296     unsigned long r2;
297     int fake = (flag & BOOTM_STATE_OS_FAKE_GO);
298
299     kernel_entry = (void (*)(int, int, uint))images->ep;
300
301     s = getenv("machid");
302     if (s) {
```

这个地址就是 0x3010_8000, 清除 CPU 的 cache 和中断后，直接跳转到这个地址，开始执行 Linux Kernel 的启动过程：

```
315     if (IMAGE_ENABLE_OF_LIBFDT && images->ft_len)
316         r2 = (unsigned long)images->ft_addr;
317     else
318         r2 = gd->bd->bi_boot_params;
319
320     if (!fake) {
321 #ifdef CONFIG_ARMV7_NONSEC
322         if (armv7_boot_nonsec()) {
323             armv7_init_nonsec();
324             secure_ram_addr(_do_nonsec_entry)(kernel_entry,
325                                               0, machid, r2);
326         } else
327 #endif
328             kernel_entry(0, machid, r2);
329     }
330 #endif
331 }
```

至此，U-Boot 完成了 Linux 的加载和引导其启动。

ENV:

```
SMDK2410 # printenv
baudrate=115200
bootargs=console=ttySAC0,115200 root=/dev/mtdblock3
bootcmd=nand read 30000000 kernel 0x400000;bootm 30000000
bootdelay=5
ethact=CS8900-0
ethaddr=1a:2b:3c:4d:5e:6f
fileaddr=30000000
filesize=5a6000
gatewayip=192.168.0.1
ipaddr=192.168.0.166
mtddevname=uboot
mtddevnum=0
mtdids=nand0=s3c2410_nand.0
mtdparts=mtdparts=s3c2410_nand.0:896k(uboot)ro,128k(params)ro,4m(kernel),-(file
ystem)
netmask=255.255.255.0
partition=nand0,0
serverip=192.168.0.116
stderr=serial
stdin=serial
stdout=serial

Environment size: 546/131068 bytes
SMDK2410 #
```

启动参数的传递

前面分析了 U-Boot 最终如何加载并跳转到 Linux 内核的入口 kernel_entry()处开始执行的，但是这个过程中有一处比较重要的就是参数的传递，kernel_entry()处，U-Boot 向 Linux Kernel 传递了三个参数：

```
328                 kernel_entry(0, machid, r2);
```

参数 r0=0, r1=machid:

```
293     unsigned long machid = gd->bd->bi_arch_number;
294     char *s;
295     void (*kernel_entry)(int zero, int arch, uint params);
296     unsigned long r2;
297     int fake = (flag & BOOTM_STATE_OS_FAKE_GO);
298
299     kernel_entry = (void (*)(int, int, uint))images->ep;
300
301     s = getenv("machid");
302     if (s) {
303         if (strict_strtoul(s, 16, &machid) < 0) {
304             debug("strict_strtoul failed!\n");
305             return;
306         }
307         printf("Using machid 0x%lx from environment\n", machid);
308     }
```

machid 如果在 u-boot 环境变量中没有定义"machid"的情况下，就是在前面 mach-types.h 中定义的"#define MACH_TYPE_SMDK2410 193":

```
96                                  #define MACH_TYPE_SMDK2410 193
97  int board_init(void)            扩展到:
98  {                               193
99      /* arch number of SMDK241
00      gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;
01
02      /* adress of boot parameters */
03      gd->bd->bi_boot_params = 0x30000100;
04
05      icache_enable();
06      dcache_enable();
07
08      return 0;
09  }
10
```

而最后一个参数 r2 则是:

```
r2 = gd->bd->bi_boot_params;
```

其中 bi_boot_params 的内容在 smdk2410.c 中被配置为:

```
 97  int board_init(void)
 98  {
 99      /* arch number of SMDK2410-Board */
100      gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;
101
102      /* adress of boot parameters */
103      gd->bd->bi_boot_params = 0x30000100;
104
105      icache_enable();
106      dcache_enable();
107
108      return 0;
109  }
```

着明显是 SDRAM 的开始地址，这个地址有什么具体作用？Linux 的启动需要依赖的参数 bootargs 又和这个地址有何联系？

这个要回到最初的 do_bootm()处，此处向下层传递了 FLAG:

```
131      return do_bootm_states(cmdtp, flag, argc, argv, BOOTM_STATE_START |
132          BOOTM_STATE_FINDOS | BOOTM_STATE_FINDOTHER |
133          BOOTM_STATE_LOADOS |
134  #if defined(CONFIG_PPC) || defined(CONFIG_MIPS)
135          BOOTM_STATE_OS_CMDLINE |
136  #endif
137          BOOTM_STATE_OS_PREP | BOOTM_STATE_OS_FAKE_GO |
138          BOOTM_STATE_OS_GO, &images, 1);
139  }
```

do_bootm_states()中，再分析完 image 的格式和对应的 OS 类型后，U-Boot 加载了针对具体系统类型的启动函数的指针，并执行：

```
676     if (!ret && (states & BOOTM_STATE_OS_PREP))
677         ret = boot_fn(BOOTM_STATE_OS_PREP, argc, argv, images);
```

这里 boot_fn= do_bootm_linux

```
339 int do_bootm_linux(int flag, int argc, char * const argv[],
340                     bootm_headers_t *images)
341 {
342     /* No need for those on ARM */
343     if (flag & BOOTM_STATE_OS_BD_T || flag & BOOTM_STATE_OS_CMDLINE)
344         return -1;
345
346     if (flag & BOOTM_STATE_OS_PREP) {
347         boot_prep_linux(images);
348         return 0;
349     }
350
```

随后从环境变量中读取 bootargs 参数：

```
203 /* Subcommand: PREP */
204 static void boot_prep_linux(bootm_headers_t *images)
205 {
206     char *commandline = getenv("bootargs");
207
```

```
216     } else if (BOOTM_ENABLE_TAGS) {
217         debug("using: ATAGS\n");
218         setup_start_tag(gd->bd);
219         if (BOOTM_ENABLE_SERIAL_TAG)
220             setup_serial_tag(&params);
221         if (BOOTM_ENABLE_CMDLINE_TAG)
222             setup_commandline_tag(gd->bd, commandline);
223         if (BOOTM_ENABLE_REVISION_TAG)
224             setup_revision_tag(&params);
225         if (BOOTM_ENABLE_MEMORY_TAGS)
226             setup_memory_tags(gd->bd);
227         if (BOOTM_ENABLE_INITRD_TAG) {
228             /*
229              * In boot_ramdisk_high(), it may relocate ramdisk to
230              * a specified location. And set images->initrd_start &
231              * images->initrd_end to relocated ramdisk's start/end
232              * addresses. So use them instead of images->rd_start &
233              * images->rd_end when possible.
234              */
235             if (images->initrd_start && images->initrd_end) {
236                 setup_initrd_tag(gd->bd, images->initrd_start,
237                                  images->initrd_end);
238             } else if (images->rd_start && images->rd_end) {
239                 setup_initrd_tag(gd->bd, images->rd_start,
```

再 setup_start_tag 中：

```
90  static void setup_start_tag (bd_t *bd)
91  {
92      params = (struct tag *)bd->bi_boot_params;
93
94      params->hdr.tag = ATAG_CORE;
95      params->hdr.size = tag_size (tag_core);
96
97      params->u.core.flags = 0;
98      params->u.core.pagesize = 0;
99      params->u.core.rootdev = 0;
100
101     params = tag_next (params);
102 }
```

将全局指针 params 指向了 gd->bd->bi_boot_params，也就是初始化了 params=0x30000100，然后就将 u-boot 中准备向 Linux Kernel 传递的参数，都通过这套 atags 机制，放在 0x30000100 为起始地址的 RAM 中，比如 bootargs 就被放在了 params 下面的 cmdline 中

```
21          if (BOOTM_ENABLE_CMDLINE_TAG)
22              setup_commandline_tag(gd->bd, commandline);
23          if (BOOTM_ENABLE_REVISION_TAG)
```

```
119 static void setup_commandline_tag(bd_t *bd, char *commandline)
120 {
121     char *p;
122
123     if (!commandline)
124         return;
125
126     /* eat leading white space */
127     for (p = commandline; *p == ' '; p++);
128
129     /* skip non-existent command lines so the kernel will still
130      * use its default command line.
131      */
132     if (*p == '\0')
133         return;
134
135     params->hdr.tag = ATAG_CMDLINE;
136     params->hdr.size =
137         (sizeof (struct tag_header) + strlen (p) + 1 + 4) >> 2;
138
139     strcpy (params->u.cmdline.cmdline, p);
140
141     params = tag_next (params);
142 }
```

然后计算下一个 params 的偏移。参数传递过程基本如此，将参数存储在这个位于 bi_boot_params 中定义的地址的 tag struct 列表中：

```
208 struct tag {
209     struct tag_header hdr;
210     union {
211         struct tag_core     core;
212         struct tag_mem32    mem;
213         struct tag_videotext    videotext;
214         struct tag_ramdisk  ramdisk;
215         struct tag_initrd   initrd;
216         struct tag_serialnr serialnr;
217         struct tag_revision revision;
218         struct tag_videolfb videolfb;
219         struct tag_cmdline  cmdline;
220 
221         /*
222          * Acorn specific
223          */
224         struct tag_acorn    acorn;
225 
226         /*
227          * DC21285 specific
228          */
229         struct tag_memclk   memclk;
230     } u;
231 };
```

Linux Kernel 分析

Kernel 编译

分析 Linux 的编译，还是从顶层 Makefile 的两个层面入手：“make xxx_defconfig” 和 “make”

首先是 make s3c2410_defconfig, 这个过程和前面分析的 U-Boot 非常相似：

对于目标 xxx_defconfig, 精简后的 Makefile:

```
...

$(srctree)/scripts/Kbuild.include: ;
include $(srctree)/scripts/Kbuild.include

...

scripts_basic:
    $(Q)$(MAKE) $(build)=scripts/basic
    $(Q)rm -f .tmp_quiet_recordmcount

scripts/basic/%: scripts_basic ;

...

outputmakefile:

...

include $(srctree)/arch/$(SRCARCH)/Makefile
export KBUILD_DEFCONFIG KBUILD_KCONFIG

config: scripts_basic outputmakefile FORCE
    $(Q)$(MAKE) $(build)=scripts/kconfig $@

%config: scripts_basic outputmakefile FORCE
    $(Q)$(MAKE) $(build)=scripts/kconfig $@

```

%config 的依赖依次为 scripts_basic, outputmakefile 和 FORCE，FORCE 是伪目标，作用是每次 make 的时候都会更新时间戳，从而导致依赖这个 FORCE 的%config 都会被重新执行。

而 scripts_basic 则会执行(Q)(MAKE) $(build)=scripts/basic， build 在前面导入的 Kbuild.include 中定义：

```
###
# Shorthand for $(Q)$(MAKE) -f scripts/Makefile.build obj=
# Usage:
# $(Q)$(MAKE) $(build)=dir
build := -f $(srctree)/scripts/Makefile.build obj
```

所以 scripts_basic 则最终会执行 make -f ./scripts/Makefile.build obj=scripts/basic

Makefile.build 和前面分析 U-Boot 的时候一样，通过 Makefile.build 最终索引到 scripts/basic/Makefile 下的内容，从而添加编译的目标，最终完成 make xxx_defconfig 中所依赖的基本工具。Linux 下 xxx_defconfig 的编译就不再继续分析，其过程可以参考 U-Boot 编译。

当输入“make ARCH=arm CROSS_COMPILE=arm-linux- “时，分析顶层 Makefile:

首先定义默认的目标:

```
# That's our default target when none is given on the command line
PHONY := _all
_all:
```

_all 真正的依赖关系为:

```
# If building an external module we do not care about the all: rule
# but instead _all depend on modules
PHONY += all
ifeq ($(KBUILD_EXTMOD),)
_all: all
else
_all: modules
endif
```

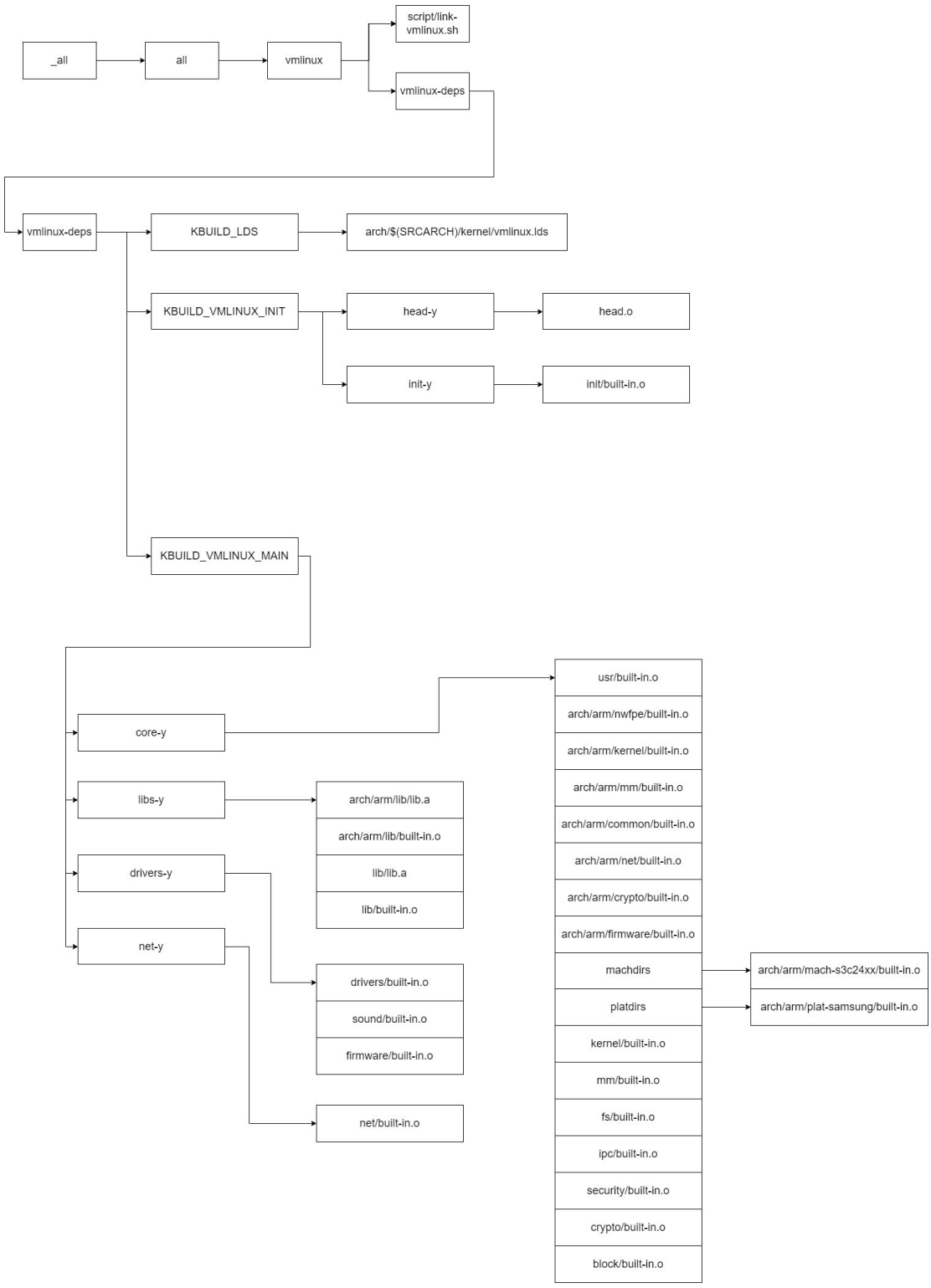
```
# The all: target is the default when no target is given on the
# command line.
# This allow a user to issue only 'make' to build a kernel including modules
# Defaults to vmlinux, but the arch makefile usually adds further targets
all: vmlinux
```

而 vmlinux 的依赖:

```
913  # Include targets which we want to
914  # execute if the rest of the kernel build went well.
915  vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
916  ifdef CONFIG_HEADERS_CHECK
917      $(Q)$(MAKE) -f $(srctree)/Makefile headers_check
918  endif
919  ifdef CONFIG_SAMPLES
920      $(Q)$(MAKE) $(build)=samples
921  endif
922  ifdef CONFIG_BUILD_DOCSRC
923      $(Q)$(MAKE) $(build)=Documentation
924  endif
925      +$(call if_changed,link-vmlinux)
```

```
891  init-y      := $(patsubst %/, %/built-in.o, $(init-y))
892  core-y      := $(patsubst %/, %/built-in.o, $(core-y))
893  drivers-y   := $(patsubst %/, %/built-in.o, $(drivers-y))
894  net-y       := $(patsubst %/, %/built-in.o, $(net-y))
895  libs-y1     := $(patsubst %/, %/lib.a, $(libs-y))
896  libs-y2     := $(patsubst %/, %/built-in.o, $(libs-y))
897  libs-y      := $(libs-y1) $(libs-y2)
898
899  # Externally visible symbols (used by link-vmlinux.sh)
900  export KBUILD_VMLINUX_INIT := $(head-y) $(init-y)
901  export KBUILD_VMLINUX_MAIN := $(core-y) $(libs-y) $(drivers-y) $(net-y)
902  export KBUILD_LDS          := arch/$(SRCARCH)/kernel/vmlinux.lds
903  export LDFLAGS_vmlinux
904  # used by scripts/pacmage/Makefile
905  export KBUILD_ALLDIRS := $(sort $(filter-out arch/%,$(vmlinux-alldirs)) arch Documenta
906
907  vmlinux-deps := $(KBUILD_LDS) $(KBUILD_VMLINUX_INIT) $(KBUILD_VMLINUX_MAIN)
```

分析整理顶层的依赖关系可以得到:



具有了完整的依赖关系，那么 Linux 内核中每个元素，是如何被索引并用何种规则进行编译的呢?

在组成 vmlinux 的依赖中，主要有 init-y, core-y, driver-y 等他们共同组成了 vmlinux-dirs:

```
883
884  vmlinux-dirs    := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
885               $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
886               $(net-y) $(net-m) $(libs-y) $(libs-m)))
887
```

对于 vmlinux-dir 其规则为：

```
937  PHONY += $(vmlinux-dirs)
938  $(vmlinux-dirs): prepare scripts
939      $(Q)$(MAKE) $(build)=$@
940
```

prepare scripts 等依赖，和 U-Boot 中的类似，都是一些早期脚本，头文件依赖等关系，不再赘述。

而(Q)(MAKE) $(build)=$(@) 这个命令则已经出现很多次了，实际就是执行 make -f Makefile.build obj=xxx 这样命令，比如 vmlinux-dirs 中包含了 init usr arch/arm/kernel 等，Makefile 会自动展开对于每个目标执行规则 make -f Makefile.build obj=xxx, 以 init 为例，就是执行 make -f Makefile.build obj=init

在 Makefile.build 中，ojb 赋值给 src 并定义默认目标为__build

```
4
5  src := $(obj)
6
7  PHONY := __build
8  __build:
9
```

然后引入 init 目录下的 Makefile:

```
41  # The filename Kbuild has precedence over Makefile
42  kbuild-dir := $(if $(filter /%,$(src)),$(src),$(srctree)/$(src))
43  kbuild-file := $(if $(wildcard $(kbuild-dir)/Kbuild),$(kbuild-dir)/Kbuild,$(kbuild-dir)/Makefile)
44  include $(kbuild-file)
45
```

在这个子目录下的 Makefile 一般就会定义目标；

```
5   obj-y                          := main.o version.o mounts.o
6   ifneq ($(CONFIG_BLK_DEV_INITRD),y)
7   obj-y                          += noinitramfs.o
8   else
9   obj-$(CONFIG_BLK_DEV_INITRD)   += initramfs.o
10  endif
11  obj-$(CONFIG_GENERIC_CALIBRATE_DELAY) += calibrate.o
12
13  ifneq ($(CONFIG_ARCH_INIT_TASK),y)
14  obj-y                          += init_task.o
15  endif
16
17  mounts-y            := do_mounts.o
18  mounts-$(CONFIG_BLK_DEV_RAM)    += do_mounts_rd.o
19  mounts-$(CONFIG_BLK_DEV_INITRD) += do_mounts_initrd.o
20  mounts-$(CONFIG_BLK_DEV_MD) += do_mounts_md.o
21
```

获取到的 obj-y 列表，如果不空，则组成了目标 builtin-target:

```
ifneq ($(strip $(obj-y) $(obj-m) $(obj-) $(subdir-m) $(lib-target)),)
builtin-target := $(obj)/built-in.o
endif
```

builtin-target 的规则则是:

```
ifdef builtin-target
quiet_cmd_link_o_target = LD      $@
# If the list of objects to link is empty, just create an empty built-in.o
cmd_link_o_target = $(if $(strip $(obj-y)),\
                $(LD) $(ld_flags) -r -o $@ $(filter $(obj-y), $^) \
                $(cmd_secanalysis),\
                rm -f $@; $(AR) rcs$(KBUILD_ARFLAGS) $@)

$(builtin-target): $(obj-y) FORCE
    $(call if_changed,link_o_target)

targets += $(builtin-target)
endif # builtin-target
```

这里 builtin-target 则依赖 obj-y, 也就是前面 init/Makefile 中定义的目标，比如 main.o, version.o 等。

在 Makefile.build 中，又定义了.c 编译为.o 的规则:

```
define rule_cc_o_c
    $(call echo-cmd,checksrc) $(cmd_checksrc)             \
    $(call echo-cmd,cc_o_c) $(cmd_cc_o_c);                \
    $(cmd_modversions)                          \
    $(call echo-cmd,record_mcount)                    \
    $(cmd_record_mcount)                            \
    scripts/basic/fixdep $(depfile) $@ '$(call make-cmd,cc_o_c)' >    \
                                                $(dot-target).tmp;  \
    rm -f $(depfile);                           \
    mv -f $(dot-target).tmp $(dot-target).cmd
endef

# Built-in and composite module parts
$(obj)/%.o: $(src)/%.c $(recordmcount_source) FORCE
    $(call cmd,force_checksrc)
    $(call if_changed_rule,cc_o_c)
```

这里核心的部分就是执行了 if_changed_rule 函数，其定义在 Kbuild.include 下:

```
# Usage: $(call if_changed_rule,foo)
# Will check if $(cmd_foo) or any of the prerequisites changed,
# and if so will execute $(rule_foo).
if_changed_rule = $(if $(strip $(any-prereq) $(arg-check) ),                  \
    @set -e;                                                    \
    $(rule_$(1)))
```

其核心功能是判断 prerequisites 是否又变化或者说更新，这里拆开看:

if 函数:

`$(if condition,then-part[,else-part])`

The `if` function provides support for conditional expansion in a functional context (as opposed to the GNU `make` makefile conditionals such as `ifeq` (see Syntax of Conditionals)).

The first argument, *condition*, first has all preceding and trailing whitespace stripped, then is expanded. If it expands to any non-empty string, then the condition is considered to be true. If it expands to an empty string, the condition is considered to be false.

If the condition is true then the second argument, *then-part*, is evaluated and this is used as the result of the evaluation of the entire `if` function.

也就是判断$(strip $(any-prereq) $(arg-check))，去掉首尾空格后是否非空，如果非空则执行命令$(rule_$(1))。

而 any-prereq:

```
244  # Find any prerequisites that is newer than target or that does not exist.
245  # PHONY targets skipped in both cases.
246  any-prereq = $(filter-out $(PHONY),$?) $(filter-out $(PHONY) $(wildcard $^),$^)
247
```

Makefile 中 filter-out 函数:

`$(filter-out pattern…,text)`

Returns all whitespace-separated words in *text* that *do not* match any of the *pattern* words, removing the words that *do* match one or more. This is the exact opposite of the `filter` function.

For example, given:

```
objects=main1.o foo.o main2.o bar.o
mains=main1.o main2.o
```

$?为自动变量，表示在所有依赖中，所有有更新的依赖

$^也是自动变量，表示所有的依赖

所以这两个 filter-out 函数，最终产生的结果就是，所有不包含 PHONY 中的目标且更新过的依赖，和新的没有被执行过的目标

而对于$(arg-check):

```
227  ifneq ($(KBUILD_NOCMDDEP),1)
228  # Check if both arguments has same arguments. Result is empty string if equal.
229  # User may override this check using make KBUILD_NOCMDDEP=1
230  arg-check = $(strip $(filter-out $(cmd_$(1)), $(cmd_$@)) \
231                      $(filter-out $(cmd_$@),   $(cmd_$(1))) )
232  else
233  arg-check = $(if $(strip $(cmd_$@)),,1)
234  endif
```

由于没有实际定义 KBUILD_NOCMDDEP, 所以执行的是前面的 arg-check, 用来互相判断 cmd_$(1) 和 cmd_$@中内容是否一致,

这里 cmd_$(1)是谁？回到 Makefile.build 中：cmd_$(1)就是 cmd_cc_o_cc

```
255  # Built-in and composite module parts
256  $(obj)/%.o: $(src)/%.c $(recordmcount_source) FORCE
257      $(call cmd,force_checksrc)
258      $(call if_changed_rule,cc_o_c)
259
```

```
184  cmd_cc_o_c = $(CC) $(c_flags) -c -o $@ $<
```

那么$@则应该对应目标，例如 init 下的 main.o, 则就是 cmd_init/main.o, 通过搜索找到 init/.main.o.cmd 下定义：

```
init > .main.o.cmd
1  cmd_init/main.o := arm-linux-gcc -Wp,-MD,init/.main.o.d  -nostdinc -isystem /usr/arm-linux-toolchai
2
```

这个文件是编译时自动生成的，具体过程后面再分析。那么此处，也就是 arg-check 是 cmd_cc_o_c 中定义的命令和 cmd_init/main.o 下定义的比较，如果不同则重新执行 $(rule_$(1)), 也就是$(rule_cc_o_c):

```
243  define rule_cc_o_c
244      $(call echo-cmd,checksrc) $(cmd_checksrc)              \
245      $(call echo-cmd,cc_o_c) $(cmd_cc_o_c);                 \
246      $(cmd_modversions)                          \
247      $(call echo-cmd,record_mcount)                  \
248      $(cmd_record_mcount)                            \
249      scripts/basic/fixdep $(depfile) $@ '$(call make-cmd,cc_o_c)' >    \
250                                                  $(dot-target).tmp;  \
251      rm -f $(depfile);                           \
252      mv -f $(dot-target).tmp $(dot-target).cmd
253  endef
254
255  # Built-in and composite module parts
256  $(obj)/%.o: $(src)/%.c $(recordmcount_source) FORCE
257      $(call cmd,force_checksrc)
258      $(call if_changed_rule,cc_o_c)
259
```

虽然命令很多，但是关键的还是再 cc_o_c 上，echo-cmd 在 Kbuild.include 中定义：

```
208  # echo command.
209  # Short version is used, if $(quiet) equals `quiet_', otherwise full one.
210  echo-cmd = $(if $($(quiet)cmd_$(1)),\
211      echo '  $(call escsq,$($(quiet)cmd_$(1)))$(echo-why)';)
```

也就是在 make 过程中对应的 V=1, V=2 等打印信息，例如默认情况下执行 quite_cmd_xxx, 对应这个例子就是 quite_cmd_cc_o_c:

```
180
181   quiet_cmd_cc_o_c = CC $(quiet_modtag)  $@
182
183   ifndef CONFIG_MODVERSIONS
184   cmd_cc_o_c = $(CC) $(c_flags) -c -o $@ $<
185
```

这也是编译中命令行中“CC xxx“的输出来源:

```
CC      init/main.o
CHK     include/generated/compile.h
CC      init/version.o
CC      init/do_mounts.o
CC      init/do_mounts_rd.o
```

其他的输出信息方法类似，都是定义了 cmd_xxx 和 quite_cmd_xxx 这样的方法。

简化后的 Makefile.build:

```
quiet_cmd_cc_o_c = CC $(quiet_modtag)  $@

cmd_cc_o_c = $(CC) $(c_flags) -c -o $@ $<

define rule_cc_o_c
    $(call echo-cmd,checksrc) $(cmd_checksrc)
    $(call echo-cmd,cc_o_c) $(cmd_cc_o_c);
    $(cmd_modversions)
    $(call echo-cmd,record_mcount)
    $(cmd_record_mcount)
    scripts/basic/fixdep $(depfile) $@ '$(call make-cmd,cc_o_c)' >
                                                $(dot-target).tmp;
    rm -f $(depfile);
    mv -f $(dot-target).tmp $(dot-target).cmd
endef

# Built-in and composite module parts
$(obj)/%.o: $(src)/%.c $(recordmcount_source) FORCE
    $(call cmd,force_checksrc)
    $(call if_changed_rule,cc_o_c)
```

最终在第三步，$(cmd_cc_o_c)中调用了 GCC 将.c 编译成对应的.o, 在第四步中将这个执行的命令存储到文件， $(dot-target)在 Kbuid.include 中的定义:

```
###
# Name of target with a '.' as filename prefix. foo/bar.o => foo/.bar.o
dot-target = $(dir $@).$(notdir $@)
```

对应着 init/main.o 这个例子就是 init/.main.o.cmd, 将命令记录下并在下次执行目标编译前，在 arg-check 中对比所执行的命令和上次执行的是否有变化，例如新加入了编译器参数等，至此，对于单个目标文件的编译流程就完成了。

但是最初的编译目标是 init/built-in.o，这些分散的.o 文件只是它的依赖而已，依赖都准备好了，就可以执行这个目标了，对应的就是执行$(call if_changed,link_o_target):

```
327  ifdef builtin-target
328  quiet_cmd_link_o_target = LD      $@
329  # If the list of objects to link is empty, just create an empty built-in.o
330  cmd_link_o_target = $(if $(strip $(obj-y)),\
331                 $(LD) $(ld_flags) -r -o $@ $(filter $(obj-y), $^) \
332                 $(cmd_secanalysis),\
333                 rm -f $@; $(AR) rcs$(KBUILD_ARFLAGS) $@)
334
335  $(builtin-target): $(obj-y) FORCE
336      $(call if_changed,link_o_target)
337
338  targets += $(builtin-target)
339  endif # builtin-target
340
```

有了前面的对于 if_changed_rule 的分析，这部分就容易多了：

```
248  # Execute command if command has changed or prerequisite(s) are updated.
249  #
250  if_changed = $(if $(strip $(any-prereq) $(arg-check)),                    \
251      @set -e;                                                              \
252      $(echo-cmd) $(cmd_$(1));                                              \
253      printf '%s\n' 'cmd_$@ := $(make-cmd)' > $(dot-target).cmd)
```

二者几乎没有太大差别，都是比较依赖是否有变化，然后执行 echo-cmd, 对应的就是 quite_cmd_link_o_target 打印 “LD init/built-in.o”，然后执行$(cmd_link_o_target)：

```
327  ifdef builtin-target
328  quiet_cmd_link_o_target = LD      $@
329  # If the list of objects to link is empty, just create an empty built-in.o
330  cmd_link_o_target = $(if $(strip $(obj-y)),\
331                 $(LD) $(ld_flags) -r -o $@ $(filter $(obj-y), $^) \
332                 $(cmd_secanalysis),\
333                 rm -f $@; $(AR) rcs$(KBUILD_ARFLAGS) $@)
334
335  $(builtin-target): $(obj-y) FORCE
336      $(call if_changed,link_o_target)
337
338  targets += $(builtin-target)
339  endif # builtin-target
```

也就是调用 arm-linux-ld 执行链接，将 obj-y 下定义的目标.o 链接成一个整体的 init/built-in.o

最终，完成了包括 init/ kernel/ driver/等前面分析的各个字模块的 built-in.o, 回到顶层 Makefile 下，完成最后的链接并生成 vmlinux:

```
913  # Include targets which we want to
914  # execute if the rest of the kernel build went well.
915  vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
916  ifdef CONFIG_HEADERS_CHECK
917      $(Q)$(MAKE) -f $(srctree)/Makefile headers_check
918  endif
919  ifdef CONFIG_SAMPLES
920      $(Q)$(MAKE) $(build)=samples
921  endif
922  ifdef CONFIG_BUILD_DOCSRC
923      $(Q)$(MAKE) $(build)=Documentation
924  endif
925      +$(call if_changed,link-vmlinux)
926
```

这里 call 函数前的+号**应该**是让 make 在调试或者 dryrun 的情况下，也会执行这个目标。

稍微不一样的地方是最终的链接是执行一个脚本来完成的：

```
909  # Final link of vmlinux
910        cmd_link-vmlinux = $(CONFIG_SHELL) $< $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux)
911  quiet_cmd_link-vmlinux = LINK    $@
912
913  # Include targets which we want to
914  # execute if the rest of the kernel build went well.
915  vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
916  ifdef CONFIG_HEADERS_CHECK
917      $(Q)$(MAKE) -f $(srctree)/Makefile headers_check
```

不过宏观流程仍然不会改变，就是把所有的 built-in.o 或者.a，根据 vmlinux.ld 最终链接为 vmlinux：

```
48   # Link of vmlinux
49   # ${1} - optional extra .o files
50   # ${2} - output file
51   vmlinux_link()
52   {
53       local lds="${objtree}/${KBUILD_LDS}"
54
55       if [ "${SRCARCH}" != "um" ]; then
56           ${LD} ${LDFLAGS} ${LDFLAGS_vmlinux} -o ${2}                \
57               -T ${lds} ${KBUILD_VMLINUX_INIT}                       \
58               --start-group ${KBUILD_VMLINUX_MAIN} --end-group ${1}
59       else
60           ${CC} ${CFLAGS_vmlinux} -o ${2}                            \
61               -Wl,-T,${lds} ${KBUILD_VMLINUX_INIT}                   \
62               -Wl,--start-group                                      \
63                    ${KBUILD_VMLINUX_MAIN}                            \
64               -Wl,--end-group                                        \
65               -lutil ${1}
66           rm -f linux
67       fi
68   }
```

uImage 的产生

在内核的启动流程中，有一个隐藏的细节需要关注就是内核的解压缩。在前面分析 U-Boot 加载 Linux 的过程中，提到了 U-Boot 实际上是分析了 uImage 并把 Image 偏移 0x40 位置的镜像加载到目标地址，但这个镜像到底是什么？了解这一点首先就要搞清楚 uImage 的组成，也就是如题如何产生的。

在编译内核阶段，输入的指令是"make ARCH=arm CROSS_COMPIEL=arm-linux- uImage"，那么编译的目标就是 uImage, 在 arch/arm/boot/Makefile 中可以找到：

```
78 $(obj)/uImage:  $(obj)/zImage FORCE
79     @$(check_for_multiple_loadaddr)
80     $(call if_changed,uimage)
81     @$(kecho) '  Image $@ is ready'
```

```
47 $(obj)/Image: vmlinux FORCE
48     $(call if_changed,objcopy)
49     @$(kecho) '  Kernel: $@ is ready'
50
51 $(obj)/compressed/vmlinux: $(obj)/Image FORCE
52     $(Q)$(MAKE) $(build)=$(obj)/compressed $@
53
54 $(obj)/zImage:  $(obj)/compressed/vmlinux FORCE
55     $(call if_changed,objcopy)
56     @$(kecho) '  Kernel: $@ is ready'
```

从这样的依赖关系中就能得到依赖关系

uImage→zImage→compressed/vmlinux→Image→vmlinux

这里又出现了常用的函数 if_changed, 所以直接搜索相关命令 cmd_uimage 和 cmd_objcopy

```
243 quiet_cmd_objcopy = OBJCOPY $@
244 cmd_objcopy = $(OBJCOPY) $(OBJCOPYFLAGS) $(OBJCOPYFLAGS_$(@F)) $< $@
245
```

```
# SRCARCH just happens to match slightly more than ARCH (on sparc), so reduces
# the number of overrides in arch makefiles
UIMAGE_ARCH ?= $(SRCARCH)
UIMAGE_COMPRESSION ?= $(if $(2),$(2),none)
UIMAGE_OPTS-y ?=
UIMAGE_TYPE ?= kernel
UIMAGE_LOADADDR ?= arch_must_set_this
UIMAGE_ENTRYADDR ?= $(UIMAGE_LOADADDR)
UIMAGE_NAME ?= 'Linux-$(KERNELRELEASE)'
UIMAGE_IN ?= $<
UIMAGE_OUT ?= $@

quiet_cmd_uimage = UIMAGE  $(UIMAGE_OUT)
      cmd_uimage = $(CONFIG_SHELL) $(MKIMAGE) -A $(UIMAGE_ARCH) -O linux \
                   -C $(UIMAGE_COMPRESSION) $(UIMAGE_OPTS-y) \
                   -T $(UIMAGE_TYPE) \
                   -a $(UIMAGE_LOADADDR) -e $(UIMAGE_ENTRYADDR) \
                   -n $(UIMAGE_NAME) -d $(UIMAGE_IN) $(UIMAGE_OUT)

```

就可以得到这样的生成关系，首先编译 Linux 内核产生 vmlinux, 这个文件本质上就是一个 ELF 文件，包含了符号表和调试信息，再由 objcopy 产生了 Image, 就是一个.bin 或者 binary 文件。而 boot/compressed/vmlinux 则是包含了真正 Kernel Image 的一个独立的 ELF，从 boot/compressed 编译而来，也包含了 boot/compressed/head.S 等文件，作用就是将压缩的内核解压缩到目标地址，而 zImage 就是 compressed/vmlinux 的 objcopy 出来的 binary 文件。最后使用 mkimage 工具，将 zImage 转换为 uImage。

U-Boot 加载 Linux 内核的过程中，uImage+0x40 地址加载到目标 0x30108000 的，其实就是 zImage, zImage 的入口和 Linux Kernel 的入口一样，然后将真正的 Kernel 再次解压到目标地址，然后再次跳转到 Kernel 入口， 这里的地址是可以重叠的，比如 U-Boot 的加载地址为 0x30108000, zImage 的解压，跳转地址也可以是 0x30108000。

所以如果使用 JTAG 调试，在 0x30108000 处打断点，从 U-Boot 执行完成后，可以发现两次进入断点，第一次就是 U-Boot 的跳转，第二次是 zImage 完成解压后的跳转。zImage 的解压的具体代码在 arch/arm/boot/compressed 下， 这里不再详细的追踪和分析。

分析 Kernel 启动流程的准备工作

和 U-Boot 启动分析类似，在分析 Linux Kernel 启动前先做两个准备工作，首先对已经编译好的 vmlinux（本质上就是一个 elf 文件）做反汇编，这样有利于早期汇编阶段的分析：

执行： arm-linux-objdump -d -g vmlinux > vmlinux.dis

第二参考链接脚本，在 Makefile 中，定义了链接时用到的脚本：

```
899  # Externally visible symbols (used by link-vmlinux.sh)
900  export KBUILD_VMLINUX_INIT := $(head-y) $(init-y)
901  export KBUILD_VMLINUX_MAIN := $(core-y) $(libs-y) $(drivers-y) $(net-y)
902  export KBUILD_LDS          := arch/$(SRCARCH)/kernel/vmlinux.lds
903  export LDFLAGS_vmlinux
904  # used by scripts/pacmage/Makefile
905  export KBUILD_ALLDIRS := $(sort $(filter-out arch/%,$(vmlinux-alldirs)) arch Documentation
906
907  vmlinux-deps := $(KBUILD_LDS) $(KBUILD_VMLINUX_INIT) $(KBUILD_VMLINUX_MAIN)
908
909  # Final link of vmlinux
910      cmd_link-vmlinux = $(CONFIG_SHELL) $< $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux)
911  quiet_cmd_link-vmlinux = LINK    $@
912
913  # Include targets which we want to
914  # execute if the rest of the kernel build went well.
915  vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
916  ifdef CONFIG_HEADERS_CHECK
917      $(Q)$(MAKE) -f $(srctree)/Makefile headers_check
918  endif
919  ifdef CONFIG_SAMPLES
920      $(Q)$(MAKE) $(build)=samples
921  endif
922  ifdef CONFIG_BUILD_DOCSRC
923      $(Q)$(MAKE) $(build)=Documentation
924  endif
925      +$(call if_changed,link-vmlinux)
926
```

arm-linux-ld -T 参数后面就是链接脚本的位置：

```
48  # Link of vmlinux
49  # ${1} - optional extra .o files
50  # ${2} - output file
51  vmlinux_link()
52  {
53      local lds="${objtree}/${KBUILD_LDS}"
54
55      if [ "${SRCARCH}" != "um" ]; then
56          ${LD} ${LDFLAGS} ${LDFLAGS_vmlinux} -o ${2}                  \
57              -T ${lds} ${KBUILD_VMLINUX_INIT}                         \
58              --start-group ${KBUILD_VMLINUX_MAIN} --end-group ${1}
59      else
60          ${CC} ${CFLAGS_vmlinux} -o ${2}                              \
61              -Wl,-T,${lds} ${KBUILD_VMLINUX_INIT}                     \
62              -Wl,--start-group                                        \
63                   ${KBUILD_VMLINUX_MAIN}                              \
64              -Wl,--end-group                                          \
65              -lutil ${1}
66          rm -f linux
67      fi
68  }
69
```

arch/arm/kernel/下其实并没有 vmlinux.lds 文件，实际上是有 vmlinux.lds.S，vmlinux.lds 文件其实就是从这个文件生成出来的：

```
298  # Linker scripts preprocessor (.lds.S -> .lds)
299  # ---------------------------------------------------------------------------
300  quiet_cmd_cpp_lds_S = LDS     $@
301        cmd_cpp_lds_S = $(CPP) $(cpp_flags) -P -C -U$(ARCH) \
302                              -D__ASSEMBLY__ -DLINKER_SCRIPT -o $@ $<
303
304  $(obj)/%.lds: $(src)/%.lds.S FORCE
305      $(call if_changed_dep,cpp_lds_S)
306
```

这样做的好处就是在.lds.s 中可以使用宏，这些宏可以用来控制一些开关和地址偏移的计算，根据不同的平台或者.config 配置来生成具有不同功能和地址大小、地址偏移的链接脚本。

类似的做法还有使用.lds.h 头文件的方式，通过编译器参数-E 使用仅预编译的方式动态转换链接脚本。

所以分析 arch/arm/kernel/vmlinux.lds.S 可以获知入口函数为 stext:

```
47
48   OUTPUT_ARCH(arm)
49   ENTRY(stext)
50
51   #ifndef   ARMEB
```

.head 的指令放在最前面：

```
89   #else
90       . = PAGE_OFFSET + TEXT_OFFSET;
91   #endif
92       .head.text : {
93           _text = .;
94           HEAD_TEXT
95       }
96
```

根据前面的分析，参与编译的 head-y 是 arch/arm/hernel/head.S

Processor Type 匹配

```
80  ENTRY(stext)
81    ARM_BE8(setend be )           @ ensure we are in BE8 mode
82
83    THUMB( adr r9, BSYM(1f)    )   @ Kernel is always entered in ARM.
84    THUMB( bx  r9      )   @ If this is a Thumb-2 kernel,
85    THUMB( .thumb         )   @ switch to Thumb now.
86    THUMB(1:           )
```

ARM_BE8(), THUMB()全为空，可以忽略，第一条实际指令：

```
91      @ ensure svc mode and all interrupts masked
92      safe_svcmode_maskall r9
93
```

```
316 .macro safe_svcmode_maskall reg:req
317 #if __LINUX_ARM_ARCH__ >= 6 && !defined(CONFIG_CPU_V7M)
318     mrs \reg , cpsr
319     eor \reg, \reg, #HYP_MODE
320     tst \reg, #MODE_MASK
321     bic \reg , \reg , #MODE_MASK
322     orr \reg , \reg , #PSR_I_BIT | PSR_F_BIT | SVC_MODE
323 THUMB(  orr \reg , \reg , #PSR_T_BIT    )
324     bne 1f
325     orr \reg, \reg, #PSR_A_BIT
326     adr lr, BSYM(2f)
327     msr spsr_cxsf, \reg
328     __MSR_ELR_HYP(14)
329     __ERET
330 1:  msr cpsr_c, \reg
331 2:
332 #else
333 /*
334  * workaround for possibly broken pre-v6 hardware
335  * (akita, Sharp Zaurus C-1000, PXA270-based)
336  */
337     setmode PSR_F_BIT | PSR_I_BIT | SVC_MODE, \reg
338 #endif
339 .endm
340
```

实际上这个宏会被展开为 setmode PSR_F_BIT | PSR_I_BIT | SVC_MODE, r9

而 setmode:

```
290
291 #if defined(CONFIG_CPU_V7M)
292     /*
293      * setmode is used to assert to be in svc mode during boot. For v7-M
294      * this is done in __v7m_setup, so setmode can be empty here.
295      */
296     .macro  setmode, mode, reg
297     .endm
298 #elif defined(CONFIG_THUMB2_KERNEL)
299     .macro  setmode, mode, reg
300     mov \reg, #\mode
301     msr cpsr_c, \reg
302     .endm
303 #else
304     .macro  setmode, mode, reg
305     msr cpsr_c, #\mode
306     .endm
307 #endif
```

所以第一条指令会实际执行的是"msr cpsr_c, 0xd3"

CPSR 寄存器的定义, 指令中的 CPSR_c 代表的就是 CPSR 寄存器第八位的控制位:

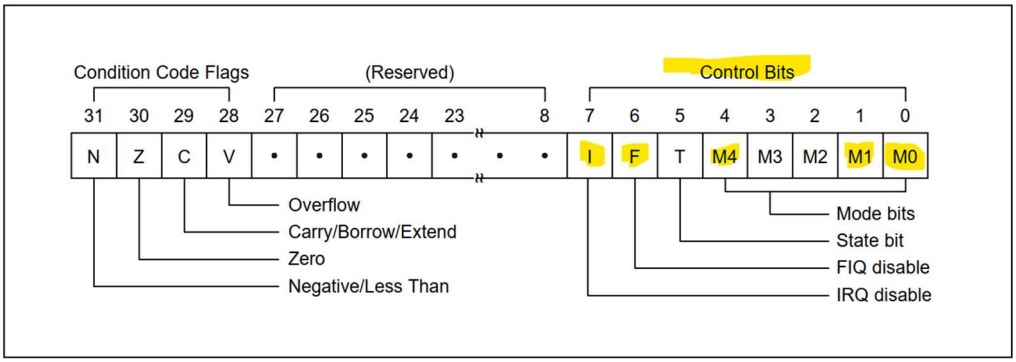


Figure 2-6. Program Status Register Format

而 Mode 的定义：

Table 2-1. PSR Mode Bit Values

M[4:0]	Mode	Visible THUMB state registers	Visible ARM state registers
10000	User	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR
10001	FIQ	R7..R0, LR_fiq, SP_fiq PC, CPSR, SPSR_fiq	R7..R0, R14_fiq..R8_fiq, PC, CPSR, SPSR_fiq
10010	IRQ	R7..R0, LR_irq, SP_irq PC, CPSR, SPSR_irq	R12..R0, R14_irq, R13_irq, PC, CPSR, SPSR_irq
10011	Supervisor	R7..R0, LR_svc, SP_svc, PC, CPSR, SPSR_svc	R12..R0, R14_svc, R13_svc, PC, CPSR, SPSR_svc
10111	Abort	R7..R0, LR_abt, SP_abt, PC, CPSR, SPSR_abt	R12..R0, R14_abt, R13_abt, PC, CPSR, SPSR_abt
11011	Undefined	R7..R0 LR_und, SP_und, PC, CPSR, SPSR_und	R12..R0, R14_und, R13_und, PC, CPSR
11111	System	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR

第一个指令的作用就是关闭 IRQ，FIQ 中断，让 CPU 工作在特权模式下，和反汇编的 vmlinux.dis 对比一下：

```
Disassembly of section .head.text:

c0108000 <stext>:
c0108000:   e321f0d3    msr CPSR_c, #211    ; 0xd3
c0108004:   ee109f10    mrc 15, 0, r9, cr0, cr0, {0}
c0108008:   eb00023a    bl  c01088f8 <__lookup_processor_type>
c010800c:   e1b0a005    movs    sl, r5
c0108010:   0a00025d    beq c010898c <__error_p>
c0108014:   e28f302c    add r3, pc, #44 ; 0x2c
```

指令内容对上了。

第二条指令" mrc p15, 0, r9, c0, c0"为读取 CP15 协处理器 0 寄存器中的 ID,

REGISTER 0: ID CODE REGISTER

This is a read-only register which returns a 32-bit device ID code.

The ID code register is accessed by reading CP15 register 0 with the opcode_2 field set to any value other than 1 (the CRm field should be zero when reading). For example:

MRC p15,0,Rd,c0,c0,0; returns ID register

The contents of the ID code are shown in Table 2-5.

Table 2-5. Register 0: ID Code

Register bits	Function	Value
31:24	Implementor	0x41
23:20	Specification revision	0x1
19:16	Architecture version (4T)	0x2
15:4	Part number	0x920
3:0	Layout revision	0x0

跳转到__lookup_processor_type 函数中执行:

```
140   * Read processor ID register (CP#15, CR0), and look up in the linker-built
141   * supported processor list.  Note that we can't use the absolute addresses
142   * for the __proc_info lists since we aren't running with the MMU on
143   * (and therefore, we are not in the correct address space).  We have to
144   * calculate the offset.
145   *
146   *  r9 = cpuid
147   * Returns:
148   *  r3, r4, r6 corrupted
149   *  r5 = proc_info pointer in physical address space
150   *  r9 = cpuid (preserved)
151   */
152  __lookup_processor_type:
153      adr r3, __lookup_processor_type_data
154      ldmia   r3, {r4 - r6}
155      sub r3, r3, r4          @ get offset between virt&phys
156      add r5, r5, r3          @ convert virt addresses to
157      add r6, r6, r3          @ physical address space
158  1:  ldmia   r5, {r3, r4}            @ value, mask
159      and r4, r4, r9          @ mask wanted bits
160      teq r3, r4
161      beq 2f
162      add r5, r5, #PROC_INFO_SZ       @ sizeof(proc_info_list)
163      cmp r5, r6
164      blo 1b
165      mov r5, #0              @ unknown processor
166  2:  ret lr
167  ENDPROC(__lookup_processor_type)
```

首先获取__lookup_processor_type_data:

```
172     .align  2
173     .type   __lookup_processor_type_data, %object
174 __lookup_processor_type_data:
175     .long   .
176     .long   __proc_info_begin
177     .long   __proc_info_end
178     .size   __lookup_processor_type_data, . - __lookup_processor_type_data
179
```

对应的 vmlinux.dis:

```
505   c0108930 <__lookup_processor_type_data>:
506   c0108930:   c0108930 c06247b8 c0624820              0....Gb. Hb.
507
```

再 vmlinux.lds.S 中找到定义:

```
14
15 #define PROC_INFO                           \
16     . = ALIGN(4);                           \
17     VMLINUX_SYMBOL(__proc_info_begin) = .;
18     *(.proc.info.init)                      \
19     VMLINUX_SYMBOL(__proc_info_end) = .;
20
```

这部分会被链接脚本 keep:

```
116             *(.glue_7)
117             *(.glue_7t)
118         . = ALIGN(4);
119         *(.got)         /* Global offset table      */
120             ARM_CPU_KEEP(PROC_INFO)
121     }
122
123 #ifdef CONFIG_DEBUG_RODATA
124     . = ALIGN(1<<SECTION_SHIFT);
125 #endif
126     RO_DATA(PAGE_SIZE)
```

而真正的数据会被放在.proc.info.init 段内，搜索可以找到对于每一个不同的 CPU，都有一个 proc-xxxx.S 的汇编代码包含了这个.proc.info.init, 参考 proc-arm920.S:

```
450
451     .section ".proc.info.init", #alloc, #execinstr
452
453     .type   __arm920_proc_info,#object
454 __arm920_proc_info:
455     .long   0x41009200
456     .long   0xff00fff0
457     .long   PMD_TYPE_SECT | \
458         PMD_SECT_BUFFERABLE | \
459         PMD_SECT_CACHEABLE | \
460         PMD_BIT4 | \
461         PMD_SECT_AP_WRITE | \
462         PMD_SECT_AP_READ
463     .long   PMD_TYPE_SECT | \
464         PMD_BIT4 | \
465         PMD_SECT_AP_WRITE | \
466         PMD_SECT_AP_READ
467     b   __arm920_setup
```

其对应的结构体位于 arch/arm/include/asm/procinfo.h:

```
28  */
29 struct proc_info_list {
30     unsigned int        cpu_val;
31     unsigned int        cpu_mask;
32     unsigned long       __cpu_mm_mmu_flags; /* used by head.S */
33     unsigned long       __cpu_io_mmu_flags; /* used by head.S */
34     unsigned long       __cpu_flush;        /* used by head.S */
35     const char      *arch_name;
36     const char      *elf_name;
37     unsigned int        elf_hwcap;
38     const char      *cpu_name;
39     struct processor    *proc;
40     struct cpu_tlb_fns  *tlb;
41     struct cpu_user_fns *user;
42     struct cpu_cache_fns    *cache;
43 };
44
```

这里存储了 CPU 的 ID 和对应的 MASK，同时也包括了 MMU 的控制标志。

编译后对应的 vmlinux.dis:

```
1053112   c06247b8 <__proc_info_begin>:
1053113   c06247b8:   41009200    .word   0x41009200
1053114   c06247bc:   ff00fff0    .word   0xff00fff0
1053115   c06247c0:   00000c1e    .word   0x00000c1e
1053116   c06247c4:   00000c12    .word   0x00000c12
1053117   c06247c8:   eaebe9d3    b   c011ef1c <__arm920_setup>
1053118   c06247cc:   c04c8798    .word   0xc04c8798
1053119   c06247d0:   c04c879f    .word   0xc04c879f
1053120   c06247d4:   00000007    .word   0x00000007
1053121   c06247d8:   c04c87a2    .word   0xc04c87a2
1053122   c06247dc:   c0626bdc    .word   0xc0626bdc
1053123   c06247e0:   c0626bd0    .word   0xc0626bd0
1053124   c06247e4:   c0626bc8    .word   0xc0626bc8
1053125   c06247e8:   c011ee20    .word   0xc011ee20
```

那么再看__lookup_processor_type 这段代码:

```
__lookup_processor_type:
    adr r3, __lookup_processor_type_data    r3=0xc0108930
    ldmia   r3, {r4 - r6}   r4,r5,r5= c0108930 c06247b8 c0624820
    sub r3, r3, r4  获取虚拟-物理地址偏移   @ get offset between virt&phys
    add r5, r5, r3          @ convert virt addresses to
    add r6, r6, r3  偏移r5,r6地址为物理地址  @ physical address space
1:  ldmia   r5, {r3, r4}    r3=0x41009200, r4=0xff00fff0    @ value, mask
    and r4, r4, r9  r9=0x41129200, r4=0x41009200    @ mask wanted bits
    teq r3, r4  if(r3==r4) return
    beq 2f
    add r5, r5, #PROC_INFO_SZ       @ sizeof(proc_info_list)
    cmp r5, r6      else continue
    blo 1b
    mov r5, #0  循环结束未匹配r3==r4, 返回r5=0      @ unknown processor
2:  ret lr
ENDPROC(__lookup_processor_type)
```

这个函数就是用来监测从硬件 CP15 寄存器 0 中读取的 CPUID，在忽略掉 CPU varent 信息后，和软件中存储的 CPUID 是否匹配，并返回值 r5, 如果匹配到 CPU 信息，返回 __proc_info_begin 的物理地址，否则返回 0.

返回后执行

```
    bl  __lookup_processor_type     @ r5=procinfo r9=cpuid
    movs    r10, r5             @ invalid processor (r5=0)?
THUMB( it  eq )         @ force fixup-able long branch encoding
    beq __error_p           @ yes, error 'p'
```

这里利用了 MOVS 执行后，会更新 CPU 标志位，如果 R5=0， 那么 Z 标志置位，后边的 BEQ 会检查 Z 标志，Z 标志置位则执行跳转__error_p 做错误处理， 打印错误信息并陷入死循环:

```
#ifdef CONFIG_DEBUG_LL
    adr r0, str_p1
    bl  printascii
    mov r0, r9
    bl  printhex8
    adr r0, str_p2
    bl  printascii
    b   __error
str_p1: .asciz  "\nError: unrecognized/unsupported processor variant (0x"
str_p2: .asciz  ").\n"
    .align
#endif
ENDPROC(__error_p)

__error:
#ifdef CONFIG_ARCH_RPC
/*
 * Turn the screen red on a error - RiscPC only.
 */
    mov r0, #0x02000000
    mov r3, #0x11
    orr r3, r3, r3, lsl #8
    orr r3, r3, r3, lsl #16
    str r3, [r0], #4
    str r3, [r0], #4
    str r3, [r0], #4
    str r3, [r0], #4
#endif
1:  mov r0, r0
    b   1b
```

虚实地址转换

后面的很多代码需要利用到虚拟地址和物理地址的转换方法，Linux 内核一般使用 0xC000_000-0xFFFF_FFFF 为起始的虚拟地址空间为内核使用，也就是保留了 1GB 的空间，在 S3C2410 里面对应的 SDRAM 的起始物理地址为 0x3000_0000, 所以就需要将这个虚拟地址和物理地址做映射，而为了保证执行效率，内核的这个空间是常驻的，并不会随着系统调度，页表内容发生改变，这样也就能保证 syscall 总是能在这个虚拟地址上避免缺页异常产生的不惜要的开销。当然这个空间 0xC000_0000-0xFFFF_FFFF 是可以改变的，对于绝大多数应用，1GB 的内核空间已经足够了，但是特殊应用需要内核拥有足够大的空间时，可以通过 KCONFIG 修改：

arch > arm > Kconfig

```
1455
1456    config PAGE_OFFSET
1457        hex
1458        default PHYS_OFFSET if !MMU
1459        default 0x40000000 if VMSPLIT_1G
1460        default 0x80000000 if VMSPLIT_2G
1461        default 0xC0000000
1462
```

回到映射的话题上，既然内核的映射总是 0x3000_0000 -> 0xC000_0000, 而且应为常驻的特性，所以这个映射是线性的，那么就需要计算一个偏移量，后面的许多物理-虚拟地址转换都需要使用到这个偏移。而由于 Linux 内核可以灵活的加载在各种平台，所以不记录编译时指定的物理地址，而是通过计算获得则是最灵活的方案，可以让 Linux 自由的加载在不同的环境或者地址上，所以编译后的镜像，可以运行在物理地址 0x3000_0000 的内存中，也可以运行在物理地址 0x8000_0000 的内存中，只要 Bootloader 能在运行时准确的告诉 Kernel 其内存的“描述”。

对于地址偏移的计算，考虑这样的公式：$P_{addr} = V_{addr} - V_{base} + P_{base} = V_{addr} + (P_{base} - V_{base})$

也就是说每次通过虚拟地址计算物理地址时，只需要**加上**$(P_{base} - V_{base})$[以后这个地址统称为 Base-Delta]即可。而在 Linux 代码中，计算这个偏移，则是通过计算$(P_{addr} - V_{addr})$来获取：

```
108 #ifndef CONFIG_XIP_KERNEL
109     adr r3, 2f
110     ldmia   r3, {r4, r8}
111     sub r4, r3, r4          @ (PHYS_OFFSET - PAGE_OFFSET)
112     add r8, r8, r4          @ PHYS_OFFSET
113 #else
```

```
148 2:  .long   .
149       .long   PAGE_OFFSET
```

上面的”adr r3, 2f”是一条伪指令，目的是为了获取相对于当前 PC 的标签 2（f 表示前向，b 表示向后获取）处的地址，标签 2 中，第一行”.long .”告诉链接器此处内容为当前的地址，“.long PAGE_OFFSET”则是把 CONFIG_PAGE_OFFSET（0xC000_0000）内容放在此处，实际反汇编：

```
c0108010:   0a00025d    beq c010898c <__error_p>
c0108014:   e28f302c    add r3, pc, #44 ; 0x2c
c0108018:   e8930110    ldm r3, {r4, r8}
c010801c:   e0434004    sub r4, r3, r4
c0108020:   e0888004    add r8, r8, r4
c0108024:   eb000062    bl  c01081b4 <__vet_atags>
c0108028:   eb00004c    bl  c0108160 <__fixup_pv_table>
c010802c:   eb000007    bl  c0108050 <__create_page_tables>
c0108030:   e59fd00c    ldr sp, [pc, #12]   ; c0108044 <stext+0x44>
c0108034:   e28fe004    add lr, pc, #4
c0108038:   e1a08004    mov r8, r4
c010803c:   e28af010    add pc, sl, #16
c0108040:   ea00003a    b   c0108130 <__enable_mmu>
c0108044:   c06032e0    .word   0xc06032e0
c0108048:   c0108048    .word   0xc0108048
c010804c:   c0000000    .word   0xc0000000
```

也就是 r3=0x30108048 执行”ldm r3, {r4, r8}”，则顺序读取 r3 地址所指向的两个 DWORD 内容到 r4,r8 中，也就是 r4=0xc0108048, r8=0xc0000000。

首先计算"sub r4, r3, r4", 也就是 r4=0x30108048 - 0xc0108048 = 0x70000000，那么执行 add r8, r8, r4"这会得到 **r8=0xc0000000 + 0x70000000 = 0x30000000（要记住这个值，后面会用到，而且会被注释位 phys_offset：物理偏移，应该叫 phys_base 可能更合适）**，也正是 S3C2410 中 SDRAM 的物理地址首地址。

继续执行（精简代码后）：

```
117     /*
118      * r1 = machine no, r2 = atags or dtb,
119      * r8 = phys_offset, r9 = cpuid, r10 = procinfo
120      */
121     bl  __vet_atags
122     bl  __fixup_pv_table
123     bl  __create_page_tables
```

这里注释提示到，r1 是 machine number, r2 为 atags 或者 dtb, r8 是前面刚刚计算出来的 PAGE_OFFSET 的物理偏移，r9 是读取的 cpuid, 而从前面分析的 U-Boot 加载 Linux Kernel 的分析中可知，最后 Linux 的入口：

```
319     r2 = gd->bd->bi_boot_params;
320
321     if (!fake) {
322             kernel_entry(0, machid, r2);
323     }
```

根据 AAPCS 的规定，kernel_entry 提供的就是 r0=0, r1=machid, r2=gd->bd->bi_boot_params, 由于这个版本是 Linux-3.19.8，此时针对这个处理器的配置还没有使用设备树，所以 head.S 中提到的 R2 就是 atags, 也就是 U-Boot 中的 gd->bd->bi_boot_params, 其位置为：

```
97  int board_init(void)
98  {
99      /* arch number of SMDK2410-Board */
100     gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;
101
102     /* adress of boot parameters */
103     gd->bd->bi_boot_params = 0x30000100;
104
105     icache_enable();
106     dcache_enable();
107
108     return 0;
109 }
```

这里面的存储的参数（ATAGS）都是再上一个阶段 U-Boot 准备好的，详细信息参考启动参数的传递。

__fixup_pv_table

在分析这部分前，需要一些相关的背景才能更容易理解这部分。前面提到，内核的虚拟地址是 0xC000_0000，而对应的物理地址映射则是线性的，那么一些地方就会用到虚拟-物理地址的转换。

在 arch/arm/include/asm/memory.h 中，定义了：

```
223  static inline phys_addr_t __virt_to_phys(unsigned long x)
224  {
225      phys_addr_t t;
226
227      if (sizeof(phys_addr_t) == 4) {
228          __pv_stub(x, t, "add", __PV_BITS_31_24);
229      } else {
230          __pv_stub_mov_hi(t);
231          __pv_add_carry_stub(x, t);
232      }
233      return t;
234  }
```

这个函数负责把虚拟地址转换为物理地址，而__pv_stub()的定义则是：

```
194  #define __pv_stub(from,to,instr,type)           \
195      __asm__("@ __pv_stub\n"                     \
196      "1: " instr "    %0, %1, %2\n"              \
197      "    .pushsection .pv_table,\"a\"\n"        \
198      "    .long    1b\n"                         \
199      "    .popsection\n"                         \
200      : "=r" (to)                                 \
201      : "r" (from), "I" (type))
```

这里".pushsection .pv_table … .popsection"是告诉链接器，把中间的内容放在.pv_table 这个 section 内，内容就是前面标签 1 的地址：

vmlinux.lds.S:

```
208      .init.pv_table : {
209          __pv_table_begin = .;
210          *(.pv_table)
211          __pv_table_end = .;
212      }
```

而这些地址所指向的内容，正是"instr" to, from, type, 对应就是 add to, from, off。例如，当一段代码执行了__virt_to_phys(x), 那么就会产生一个对应的 add 指令，而同时这个指令所在的地址也会存储在.pv_table 这个 section 中。

了解了这些后，那么后面的工作就不难猜测了，在系统计算出 Phy-Offset 后，需要一个 patch 函数，通过.pv_table 来寻找这些分散在代码各处的计算物理地址的指令，把 hard code 的 offset，替换为 Base-Delta。

那么看下面的代码就不难理解了：

```
580  __fixup_pv_table:
581      adr r0, 1f
582      ldmia   r0, {r3-r7}
583      mvn ip, #0
584      subs    r3, r0, r3  @ PHYS_OFFSET - PAGE_OFFSET
585      add r4, r4, r3  @ adjust table start address
586      add r5, r5, r3  @ adjust table end address
587      add r6, r6, r3  @ adjust __pv_phys_pfn_offset address
588      add r7, r7, r3  @ adjust __pv_offset address
589      mov r0, r8, lsr #12 @ convert to PFN
590      str r0, [r6]    @ save computed PHYS_OFFSET to __pv_phys_pfn_offset
591      strcc   ip, [r7, #HIGH_OFFSET]  @ save to __pv_offset high bits
592      mov r6, r3, lsr #24 @ constant for add/sub instructions
593      teq r3, r6, lsl #24 @ must be 16MiB aligned
594  THUMB(  it  ne      @ cross section branch )
595      bne __error
596      str r3, [r7, #LOW_OFFSET]   @ save to __pv_offset low bits
597      b   __fixup_a_pv_table
598  ENDPROC(__fixup_pv_table)
599
600      .align
601  1:  .long   .
602      .long   __pv_table_begin
603      .long   __pv_table_end
604  2:  .long   __pv_phys_pfn_offset
605      .long   __pv_offset
```

通过寻找__pv_table_begin，找到了所有调用__virt_to_phys()和__phys_to_virt()所产生的 add/sub 指令所在的位置，然后修改这些指令，让 add/sub P_{addr}, V_{addr}, **Offset** 中的偏移为动态计算出来的偏移，以满足内核的地址无关加载的特性。

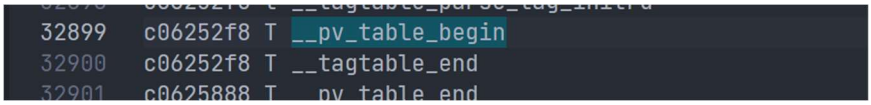
但是为什么不用类似 U-Boot 中使用的地址无关代码的方法，利用-pie 选项和.rel_dyn 段的方法，把偏移的地址存储在对应程序段的尾部，然后再早期通过.rel_dyn 段寻找到这个.rodata，并修改对应的偏移地址？这里猜测可能时为了提高执行效率和减少冗余的内容消耗，所以采取了这种修改指令码的 patch 方式来完成地址无关特性。

验证：

查看反汇编：

```
334925:Contents of section .init.pv_table:
334926- c06252f8 584160c0 d86c11c0 e46a60c0 306b60c0  XA`..l...j`.0k`.
334927- c0625308 3c6b60c0 486b60c0 e07911c0 187160c0  <k`.Hk`..y...q`.
334928- c0625318 509c11c0 649c11c0 789c11c0 109d11c0  P...d...x.......
334929- c0625328 989d11c0 a8b411c0 40b811c0 1cbd11c0  ........@.......
334930- c0625338 207560c0 247560c0 647560c0 10c311c0   u`.$u`.du`.....
```

第一列绿色字时行号，第二列时反汇编文件中的地址，也就是对应着.init.pv_table 的地址：



后面的内容时数据，这里为小端格式所以字节顺序为反的，那么对应的地址为 0xc0604158, 0xc0116cd8, 0xc0606ae4, …

```
5213874:c0604158:       e2466481        sub     r6, r6, #-2130706432    ; 0x81000000

4210814:c0116cd8:       e2833481        add     r3, r3, #-2130706432    ; 0x81000000

5216624:c0606ae4:       e2855481        add     r5, r5, #-2130706432    ; 0x81000000
```

初始化页表

在分析页表的初始化前，首先需要了解一下 Linux 的页表机制和如何关联 S3C2410 的 MMU 硬件。

Linux 内核的页表按照层级顺序为 PGD->P4D->PUD->PMD->PTE。

PGD：Page Global Directory, 是顶层的页表, 一般有 2048 个 entry

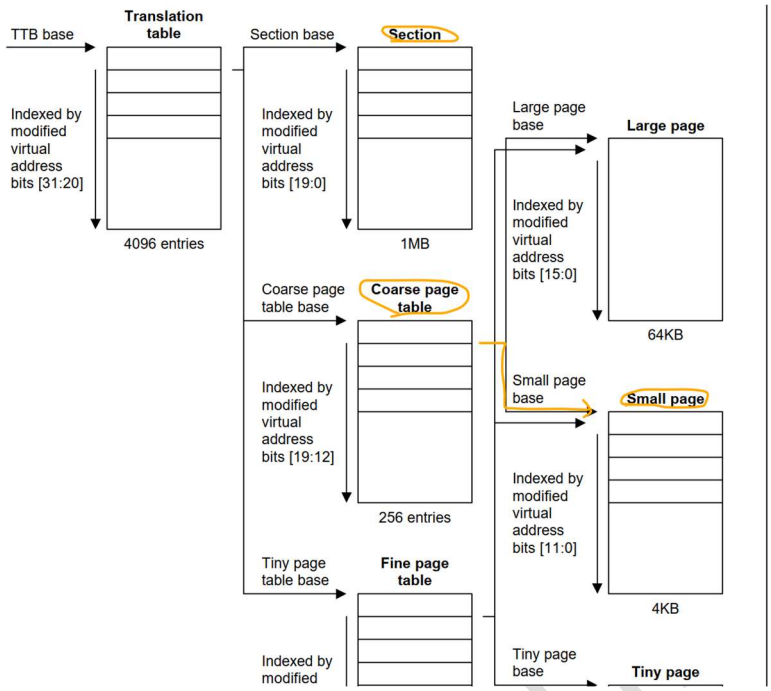
P4D：Page Level 4 Directory，ARM 中没有用到

PUD：Page Upper Direcotry，ARM 中没有用到

PMD: Page Middle Directory

PTE：Page Table Entry, 最后一级指针数组，一般有 512 个 entry，一般配置每一个 Entry 指向一个 4KB 的 Page.

对于 MMU 的硬件，前面分析 U-Boot 启动时分析过，其 MMU 的硬件页表结构：



S3C2410 的 TTB 指向一个具有 4096 个条目的转换表，这个转换可以分为一级转换和二级转换，如果是一级转换，则转换表中每一个条目指向一个 1MB 大小的 section, 如果是二级转换，则转换表中每一个条目指向一个具有 256 条目的 Coarse Page Table, 这个 Coarse Page Table 的每一个条目则指向 4KB 大小的 Page, 其他 Large Page 或者 Fine Page Table 则 Linux 都没有用到，不再详述。

这样这个 MMU 覆盖的空间就是 4096 * 1MB = 4GB, 或者 4096 * 256 * 4KB = 4GB, 覆盖全部 32 位的地址空间。

区分 Section 和 Coarse Page Table 是一级转换表中的 Bit[1:0]:

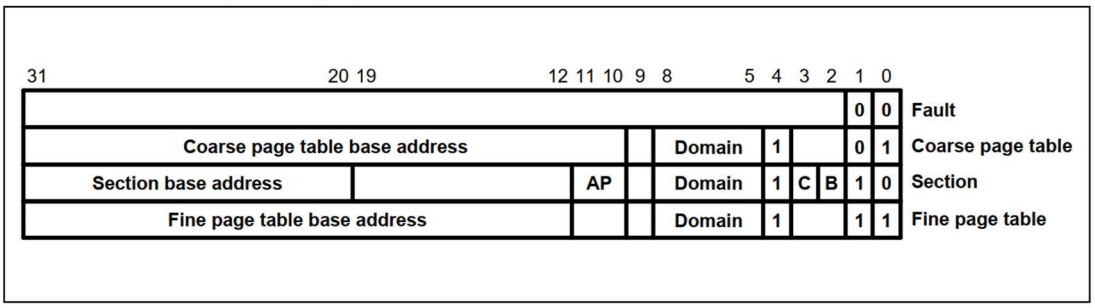
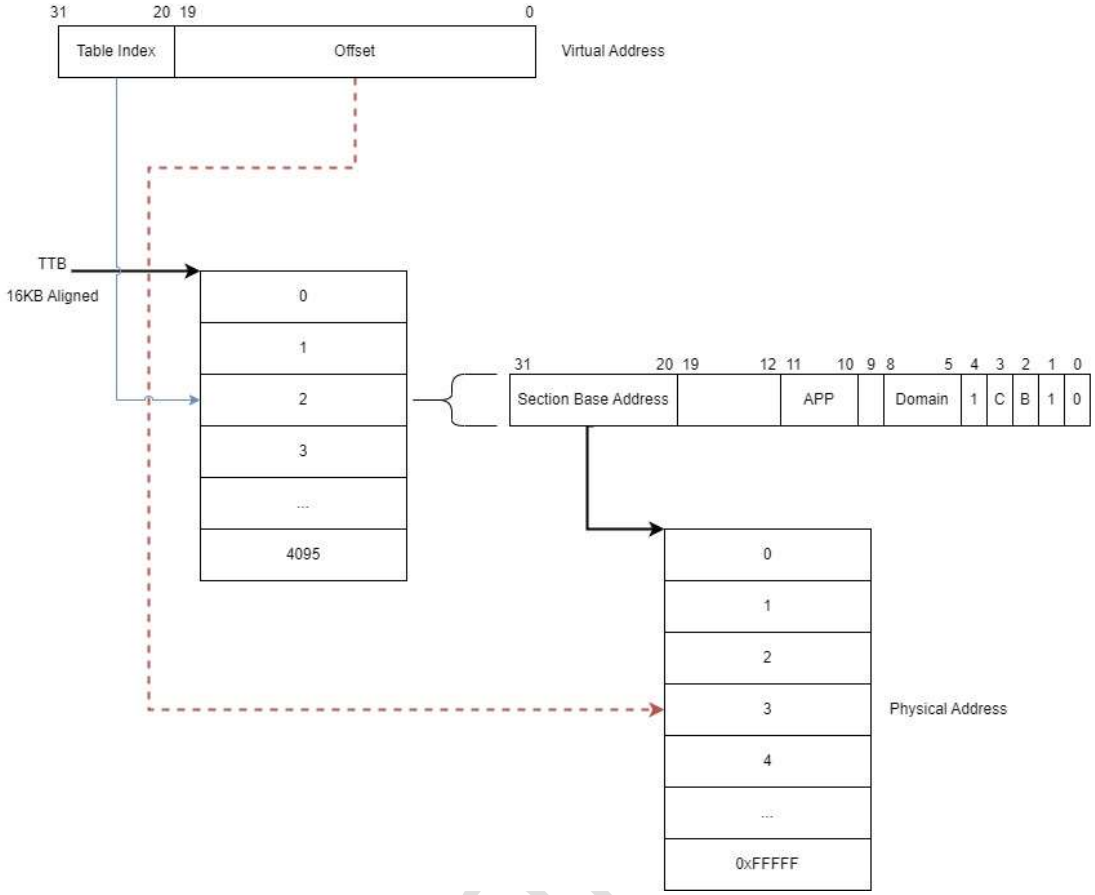
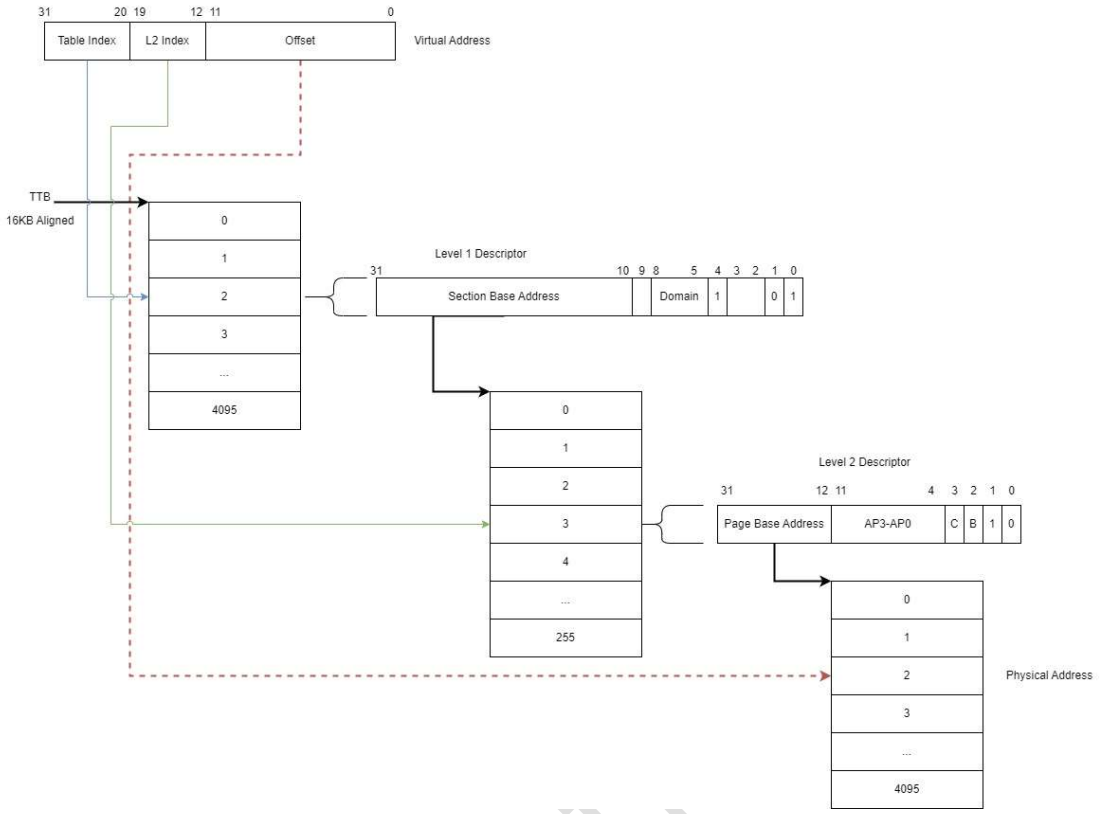


Figure 3-4. Level One Descriptors

如果是一级索引:



二级索引:



了解了以上背景信息后，开始设置初始化页表:

```
152  /*
153   * Setup the initial page tables.  We only setup the barest
154   * amount which are required to get the kernel running, which
155   * generally means mapping in the kernel code.
156   *
157   * r8 = phys_offset, r9 = cpuid, r10 = procinfo
158   *
159   * Returns:
160   *  r0, r3, r5-r7 corrupted
161   *  r4 = page table (see ARCH_PGD_SHIFT in asm/memory.h)
162   */
163  __create_page_tables:
164      pgtbl   r4, r8              @ page table address
165
166      /*
167       * Clear the swapper page table
168       */
169      mov r0, r4
170      mov r3, #0
171      add r6, r0, #PG_DIR_SIZE
172  1:  str r3, [r0], #4
173      str r3, [r0], #4
174      str r3, [r0], #4
175      str r3, [r0], #4
176      teq r0, r6
177      bne 1b
178
179      ldr r7, [r10, #PROCINFO_MM_MMUFLAGS] @ mm_mmuflags
180
```

第一个宏”pgtbl r4,r8”会被编译成：

```
54      .macro  pgtbl, rd, phys
55      add \rd, \phys, #TEXT_OFFSET
56      sub \rd, \rd, #PG_DIR_SIZE
57      .endm
58
```

对应的汇编就是”add r4, r8, 0x108000” 和 “sub r4, r4, 0x4000”, 通过前面的分析 r8=0x3000_0000, 那么 r4=0x3010_4000。

前面分析 U-Boot 配置 S3C2410 MMU 的时候提到过，S3C2410 处理器中 MMU 的基地址寄存器是 CP15 寄存器 2：

REGISTER 2: TRANSLATION TABLE BASE (TTB) REGISTER

This is the translation table base register, for the currently active first level translation table. The contents of register 2 are shown in Table 2-12.

Table 2-12. Register 2: Translation Table Base

Register Bits	Function
31:14	Pointer to first level translation table base. Read/write
13:0	Reserved Read = Unpredictable Write = Should be zero

其低 14 位无效，所以对应的在 Linux 中，页表的起始地址也必须是 16KB 对其的，所以最初始的页表的地址就是：0x3010_4000 = 0x3000_0000 + 0x108000(TEXT_OFFSET) - 0x4000(16KB),

上面的代码就是加载 Page Table 的首地址和根据 PG_DIR_SIZE(0x4000)计算出 PGD 的空间范围，然后全部初始化为 0。

然后获取要配置 MMU 的标志位，这些信息存储在 proc-arm920.s 中，根据 proc-arm920.s 中存储的 proc_info_list 中存储的 MMU 标志：

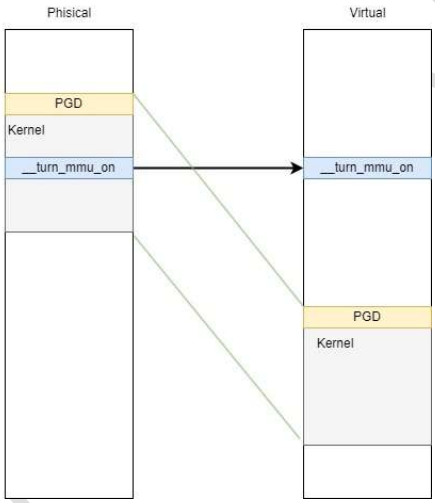
```
457     .long   PMD_TYPE_SECT | \
458         PMD_SECT_BUFFERABLE | \
459         PMD_SECT_CACHEABLE | \
460         PMD_BIT4 | \
461         PMD_SECT_AP_WRITE | \
462         PMD_SECT_AP_READ
463     .long   PMD_TYPE_SECT | \
464         PMD_BIT4 | \
465         PMD_SECT_AP_WRITE | \
466         PMD_SECT_AP_READ
467     b   __arm920_setup
```

这里存储了两种 MMU FLAG, 一种是 RAM 用的，采用的策略就是 Cachable & Bufferable, 第二个则是 IO 段使用的，则关闭了 Cacheable 和 Bufferable。

然后初始化一个特殊的映射：

```
209     /*
210      * Create identity mapping to cater for __enable_mmu.
211      * This identity mapping will be removed by paging_init().
212      */
213     adr r0, __turn_mmu_on_loc
214     ldmia   r0, {r3, r5, r6}
215     sub r0, r0, r3          @ virt->phys offset
216     add r5, r5, r0          @ phys __turn_mmu_on
217     add r6, r6, r0          @ phys __turn_mmu_on_end
218     mov r5, r5, lsr #SECTION_SHIFT
219     mov r6, r6, lsr #SECTION_SHIFT
```

把__turn_mmu_on 这个函数做一个特殊的映射，把起物理地址和虚拟地址做 1：1 映射，而不是整体从 0x3000_0000 偏移映射到 0xC000_0000:



例如__turn_mmu_on 所在的地址是 0x304c_6b98 ~ 0x304c_6bb0, 那么起对应的虚拟地址也是这个地址，而不是 0xc04c_6b98 ~ 0xc04c_6bb0。

这样做的目的就是在打开 MMU 的瞬间，MMU 的物理硬件能让然保持 PC 在正确的区域，如果没有虚拟地址对应的 0x304c_6b98 对应的页表，则 MMU 则无法正确的完成虚拟地址到物理地址的转换，程序也将无法继续正确执行。

```
    ldr r7, [r10, #PROCINFO_MM_MMUFLAGS] @ mm_mmuflags
                                         @ L1Descriptor 获取标志位

    adr r0, __turn_mmu_on_loc    @ 获取标签__turn_mmu_on_loc地址
    ldmia   r0, {r3, r5, r6} @ 去读标签处DWORD *3 内容
    sub r0, r0, r3          @ 计算Base-Delta
    add r5, r5, r0          @ 计算 __turn_mmu_on 物理地址
    add r6, r6, r0          @ 计算 __turn_mmu_on_end 物理地址
    mov r5, r5, lsr #SECTION_SHIFT @ 逻辑右移20 计算页表Index, 结果为r5=772
    mov r6, r6, lsr #SECTION_SHIFT @ 逻辑右移20 计算页表Index, 结果为r6=772

1:  orr r3, r7, r5, lsl #SECTION_SHIFT  @ 把Index << 20 | flags 构建L1 Descriptor
    str r3, [r4, r5, lsl #PMD_ORDER]    @ r4是前面计算的PGD的首地址, 把r3中的L1 Descriptor
                                        @ 存储在r4 + Index * 4的地址上
    cmp r5, r6
    addlo   r5, r5, #1
    blo 1b                              @ 循环以确保所有index都被1：1映射

__turn_mmu_on_loc:
    .long   .
    .long   __turn_mmu_on
    .long   __turn_mmu_on_end

```

后面的代码功能整体类似，把整个 kernel 用到的空间都根据其所在的 section， 构建 L1 Descriptor, 然后映射到对应的虚拟地址。如果使用了 DEBUG_LL 的功能，则同样会把 UART 的空间也做映射，区别是 L1 Descriptor 的 CB 策略不同。

再配置完页表后，就要开启 MMU，将以后的内核执行地址从物理地址 0x3xxx_xxxx 切换到 0xCxxx_xxxx 上：

```
    /*
     * The following calls CPU specific code in a position independent
     * manner.  See arch/arm/mm/proc-*.S for details.  r10 = base of
     * xxx_proc_info structure selected by __lookup_processor_type
     * above.  On return, the CPU will be ready for the MMU to be
     * turned on, and r0 will hold the CPU control register value.
     */
    ldr r13, =__mmap_switched       @ address to jump to after
                                    @ mmu has been enabled
    adr lr, BSYM(1f)                @ return (PIC) address
    mov r8, r4                      @ set TTBR1 to swapper_pg_dir
ARM(    add pc, r10, #PROCINFO_INITFUNC )       @跳转至__arm920_setup
THUMB(  add r12, r10, #PROCINFO_INITFUNC    )
THUMB(  mov pc, r12             )
1:  b   __enable_mmu
ENDPROC(stext)
```

最后的跳转有一些绕，首先把__map_switched 的地址存储再 r13(sp 寄存器)中，随后赋值 lr 地址指向”b __enable_mmu”指令处，然后直接修改 PC 值为 proc_info_list→__cpu_flush，再 ARM920 中就对应着__arm920_setup:

```
414 __arm920_setup:
415     mov r0, #0
416     mcr p15, 0, r0, c7, c7      @ invalidate I,D caches on v4
417     mcr p15, 0, r0, c7, c10, 4      @ drain write buffer on v4
418 #ifdef CONFIG_MMU
419     mcr p15, 0, r0, c8, c7      @ invalidate I,D TLBs on v4
420 #endif
421     adr r5, arm920_crval
422     ldmia   r5, {r5, r6}
423     mrc p15, 0, r0, c1, c0      @ get control register v4
424     bic r0, r0, r5
425     orr r0, r0, r6
426     ret lr
427     .size   __arm920_setup, . - __arm920_setup
```

这个函数清除了 I/D chche 和 TLB，然后返回，也就是回到指令"b __enable_mmu"处，然后跳转执行，__enable_mmu 中更新了 CP15-寄存器 3 更新 Domain Access 和寄存器 2 TTB 基地址后，又再次跳转到__turn_mmu_on:

```
436 #ifndef CONFIG_ARM_LPAE
437     mov r5, #(domain_val(DOMAIN_USER, DOMAIN_MANAGER) | \
438               domain_val(DOMAIN_KERNEL, DOMAIN_MANAGER) | \
439               domain_val(DOMAIN_TABLE, DOMAIN_MANAGER) | \
440               domain_val(DOMAIN_IO, DOMAIN_CLIENT))
441     mcr p15, 0, r5, c3, c0, 0       @ load domain access register
442     mcr p15, 0, r4, c2, c0, 0       @ load page table pointer
443 #endif
444     b   __turn_mmu_on
445 ENDPROC(__enable_mmu)
446
```

```
462     .pushsection    .idmap.text, "ax"
463 ENTRY(__turn_mmu_on)
464     mov r0, r0
465     instr_sync
466     mcr p15, 0, r0, c1, c0, 0       @ write control reg
467     mrc p15, 0, r3, c0, c0, 0       @ read id reg
468     instr_sync
469     mov r3, r3
470     mov r3, r13
471     ret r3
```

在__turn_mmu_on 中，通过 CP15-寄存器 1 控制打开 MMU，至此 MMU 功能完全打开，虚拟-物理地址映射开始起作用，然后通过前面存储__map_switched 地址的 r13，跳转到这个函数继续执行:

```
__mmap_switched:
    adr r3, __mmap_switched_data

    ldmia   r3!, {r4, r5, r6, r7}
    cmp r4, r5              @ Copy data segment if needed
1:  cmpne   r5, r6
    ldrne   fp, [r4], #4
    strne   fp, [r5], #4
    bne 1b

    mov fp, #0              @ Clear BSS (and zero fp)
1:  cmp r6, r7
    strcc   fp, [r6],#4
    bcc 1b

 ARM(   ldmia   r3, {r4, r5, r6, r7, sp})
 THUMB( ldmia   r3, {r4, r5, r6, r7}    )
 THUMB( ldr sp, [r3, #16]       )
    str r9, [r4]            @ Save processor ID
    str r1, [r5]            @ Save machine type
    str r2, [r6]            @ Save atags pointer
    cmp r7, #0
    strne   r0, [r7]            @ Save control register values
    b   start_kernel
ENDPROC(__mmap_switched)

    .align  2
    .type   __mmap_switched_data, %object
__mmap_switched_data:
    .long   __data_loc          @ r4
    .long   _sdata              @ r5
    .long   __bss_start         @ r6
    .long   _end                @ r7
    .long   processor_id            @ r4
    .long   __machine_arch_type     @ r5
    .long   __atags_pointer         @ r6
#ifdef CONFIG_CPU_CP15
    .long   cr_alignment            @ r7
#else
    .long   0               @ r7
#endif
    .long   init_thread_union + THREAD_START_SP @ sp
    .size   __mmap_switched_data, . - __mmap_switched_data
```

这段函数基本的功能就是存储一些关键变量例如 processor_id(0x41129200), __machine_arch_type 等，然后初始化 C 语言环境，本质上就是初始化堆栈指针 SP。

对应的汇编“ARM(ldmia r3, {r4, r5, r6, r7, sp}) “， 会让 sp= init_thread_union + THREAD_START_SP

先看 THREAD_START_SP:

```
#define THREAD_SIZE_ORDER   1
#define THREAD_SIZE     (PAGE_SIZE << THREAD_SIZE_ORDER)
#define THREAD_START_SP     (THREAD_SIZE - 8)

```

```
/* PAGE_SHIFT determines the page size */
#define PAGE_SHIFT      12
#define PAGE_SIZE       (_AC(1,UL) << PAGE_SHIFT)
#define PAGE_MASK       (~((1 << PAGE_SHIFT) - 1))
```

可知 PAGE_SIZE = 4096, 所以 THREAD_START_SP = 8184

也就是在__mmap_switched 中将 SP 初始为 init_thread_union + 8184 的地址后，跳转 C 语言环境的 start_kernel 中继续执行。

start_kernel

set_task_stack_end_magic

在堆栈的最后一个地址设置一个 Magic Word， 当系统检查到这个地址的数据不再是 Magic Word 时，表明堆栈发生了溢出：

```
    stackend = end_of_stack(tsk);
    *stackend = STACK_END_MAGIC;    /* for overflow detection */
```

```
static inline unsigned long *end_of_stack(struct task_struct *p)
{
#ifdef CONFIG_STACK_GROWSUP
    return (unsigned long *)((unsigned long)task_thread_info(p) + THREAD_SIZE) - 1;
#else
    return (unsigned long *)(task_thread_info(p) + 1);
#endif
}
```

在进入 start_kernel 之前，系统首先将堆栈的指针设置为了 init_thread_union + 8184， 而 init_thread_union 的定义在 init/init_task.c 中：

```
/* Initial task structure */
struct task_struct init_task = INIT_TASK(init_task);
EXPORT_SYMBOL(init_task);

/*
 * Initial thread structure. Alignment of this is handled by a special
 * linker map entry.
 */
union thread_union init_thread_union __init_task_data =
    { INIT_THREAD_INFO(init_task) };
```

这里定义的 init_thread_union {.task = &init_task;}，而 init_task 的初始化则是：

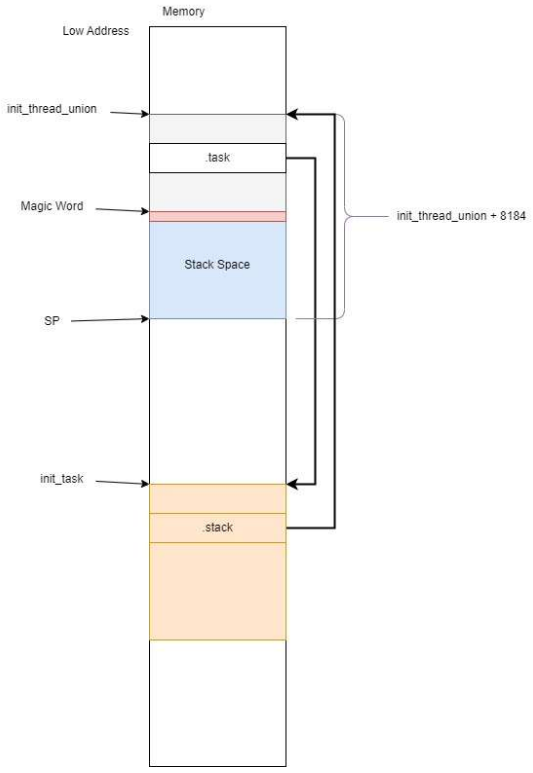
```
178 /*
179  *  INIT_TASK is used to set up the first task table, touch at
180  * your own risk!. Base=0, limit=0x1fffff (=2MB)
181  */
182 #define INIT_TASK(tsk)  \
183 {                                       \
184     .state      = 0,                    \
185     .stack      = &init_thread_info,            \
186     .usage      = ATOMIC_INIT(2),           \
187     .flags      = PF_KTHREAD,               \
188     .prio       = MAX_PRIO-20,              \
189     .static_prio    = MAX_PRIO-20,          \
190     .normal_prio    = MAX_PRIO-20,          \
```

```
88
89  #define init_thread_info    (init_thread_union.thread_info)
90  #define init_stack      (init_thread_union.stack)
```

而宏函数:

```
2673  #define task_thread_info(task)  ((struct thread_info *)(task)->stack)
```

也就是说宏函数获取了 init_task->stack 的地址，起始就是 init_thread_info 的位置，然后 end_of_stack()这个函数获取就是 init_thread_info 的指针(struct thread_info *), 然后指针偏移 1 就是偏移了整个 sizeof(struct thread_info)，也就是 init_thread_info 的末尾，对应着 stack 的顶部:



local_irq_disable

前面的早期的初始化 IRQ 已经是关闭的状态了，这里又设置了一边可能的目的时为了增强鲁棒性?

由于定义了 CONFIG_TRACE_IRQFLAGS_SUPPORT:

```
88 #ifdef CONFIG_TRACE_IRQFLAGS_SUPPORT
89 #define local_irq_enable() \
90     do { trace_hardirqs_on(); raw_local_irq_enable(); } while (0)
91 #define local_irq_disable() \
92     do { raw_local_irq_disable(); trace_hardirqs_off(); } while (0)
```

```
59 #define raw_local_irq_disable()     arch_local_irq_disable()
```

```
34 static inline void arch_local_irq_enable(void)
35 {
36     asm volatile(
37         "    cpsie i         @ arch_local_irq_enable"
38         :
39         :
40         : "memory", "cc");
41 }
42
43 static inline void arch_local_irq_disable(void)
44 {
45     asm volatile(
46         "    cpsid i         @ arch_local_irq_disable"
47         :
48         :
49         : "memory", "cc");
50 }
```

最终是通过 cpsid 指令，关闭了 ARM 中的总中断。

setup_arch - 准备 proc_info_list 等

与 ARM 架构相关的，尤其是中断、Memory、MMU 相关内容都在这里初始化，这个函数位于 arch/arm/kernel/setup.c

首先在 setup_processor()中获得 proc_info_list 的指针:

```
591     list = lookup_processor_type(read_cpuid_id());
```

lookup_processor_type 就是前面 2024/5/4 分析到的 head-common.S 中的 __lookup_processor_type 的 C 语言 API 的 wrapper:

```
128 ENTRY(lookup_processor_type)
129     stmfd   sp!, {r4 - r6, r9, lr}
130     mov r9, r0
131     bl  __lookup_processor_type
132     mov r0, r5
133     ldmfd   sp!, {r4 - r6, r9, pc}
134 ENDPROC(lookup_processor_type)
```

根据 AAPCS 规则，第一个入参会放入 r0 中，第一个返回值也是 r0, 而汇编 __lookup_processor_type 则是从 r9 中获取读取的 cpuid, 并匹配到对应的 proc_info_list 并把地址放在 r5 中。

所以此处就会返回 proc-arm920.S 中定义的__arm920_proc_info：

```
451     .section ".proc.info.init", #alloc, #execinstr
452
453     .type   __arm920_proc_info,#object
454 __arm920_proc_info:
455     .long   0x41009200
456     .long   0xff00fff0
457     .long   PMD_TYPE_SECT | PMD_SECT_BUFFERABLE | PMD_SECT_CACHEABLE | \
458             PMD_BIT4 | PMD_SECT_AP_WRITE | PMD_SECT_AP_READ
459     .long   PMD_TYPE_SECT | PMD_BIT4 | PMD_SECT_AP_WRITE | PMD_SECT_AP_READ
460     b   __arm920_setup
461     .long   cpu_arch_name
462     .long   cpu_elf_name
463     .long   HWCAP_SWP | HWCAP_HALF | HWCAP_THUMB
464     .long   cpu_arm920_name
465     .long   arm920_processor_functions
466     .long   v4wbi_tlb_fns
467     .long   v4wb_user_fns
468 #ifndef CONFIG_CPU_DCACHE_WRITETHROUGH
469     .long   arm920_cache_fns
470 #else
471     .long   v4wt_cache_fns
472 #endif
473     .size   __arm920_proc_info, . - __arm920_proc_info
```

其中 arm920_processor_functions 的定义是由 arch/arm/mm/proc-macros.S 中宏函数实现：

```
261 .macro define_processor_functions name:req, dabort:req, pabort:req, nommu=0, suspend=0
262     .type   \name\()_processor_functions, #object
263     .align 2
264 ENTRY(\name\()_processor_functions)
265     .word   \dabort
266     .word   \pabort
267     .word   cpu_\name\()_proc_init
268     .word   cpu_\name\()_proc_fin
269     .word   cpu_\name\()_reset
270     .word   cpu_\name\()_do_idle
271     .word   cpu_\name\()_dcache_clean_area
272     .word   cpu_\name\()_switch_mm
273
274     .if \nommu
275     .word   0
276     .else
```

```
440     @ define struct processor (see <asm/proc-fns.h> and proc-macros.S)
441     define_processor_functions arm920, dabort=v4t_early_abort, pabort=legacy_pabort, suspend=1
```

对于 v4wbi_tlb_fns 和 v4wt_cache_fns 都是类似的，宏函数 define_tlb_functions 和 define_cache_funcions 定义在 arch/arm/mm/proc-macros.S 中，而具体的实现则是：

和 arch/arm/mm/tlb-v4wbi.S:

```
64      /* define struct cpu_tlb_fns (see <asm/tlbflush.h> and proc-macros.S) */
65      define_tlb_functions v4wbi, v4wbi_tlb_flags
```

类似的对于 arch/arm/mm/cache-v4wt.S:

```
205     @ define struct cpu_cache_fns (see <asm/cacheflush.h> and proc-macros.S)
206     define_cache_functions v4wt
```

最后对于 S3C2410 的 proc_info_list 的具体内容：

```
struct proc_info_list {
    cpu_val            = 0x41009200
    cpu_mask           = 0xff00fff0
    __cpu_mm_mmu_flags = PMD_TYPE_SECT | \
                         PMD_SECT_BUFFERABLE | \
                         PMD_SECT_CACHEABLE | \
                         PMD_BIT4 | \
                         PMD_SECT_AP_WRITE | \
                         PMD_SECT_AP_READ
    __cpu_io_mmu_flags = PMD_TYPE_SECT | \
                         PMD_BIT4 | \
                         PMD_SECT_AP_WRITE | \
                         PMD_SECT_AP_READ
    __cpu_flush        = b  __arm920_setup
    *arch_name         = "armv4t"
    *elf_name          = "v4"
    elf_hwcap          = HWCAP_SWP | HWCAP_HALF | HWCAP_THUMB
    *cpu_name          = "ARM920T"

    struct processor {
        (*_data_abort)()        = v4t_early_abort
        (*_prefetch_abort)()    = legacy_pabort
        (*_proc_init)()         = cpu_arm920_proc_init
        (*_proc_fin)()          = cpu_arm920_proc_fin
        (*reset)()              = cpu_arm920_reset
        (*_do_idle)()           = cpu_arm920_do_idle
        (*dcache_clean_area)()  = cpu_arm920_dcache_clean_area
        (*switch_mm)()          = cpu_arm920_switch_mm
        (*set_pte_ext)()        = cpu_arm920_set_pte_ext
        suspend_size            = cpu_arm920_suspend_size // 12
        (*do_suspend)()         = cpu_arm920_do_suspend
        (*do_resume)()          = cpu_arm920_do_resume
    }*proc;

    struct cpu_tlb_fns {
        (*flush_user_range)()   = v4wbi_flush_user_tlb_range
        (*flush_kern_range)()   = v4wbi_flush_kern_tlb_range
        tlb_flags               = v4wbi_tlb_flags //(TLB_WB | TLB_DCLEAN | \
                                                  // TLB_V4_I_FULL | TLB_V4_D_FULL | \
                                                  //TLB_V4_I_PAGE | TLB_V4_D_PAGE
    } *tlb;

    struct cpu_user_fns {
        (*cpu_clear_user_highpage)() = v4wb_clear_user_highpage
        (*cpu_copy_user_highpage)()  = v4wb_copy_user_highpage
    } *user;

    struct cpu_cache_fns {
        (*flush_icache_all)()        = v4wt_flush_icache_all
        (*flush_kern_all)()          = v4wt_flush_kern_cache_all
        (*flush_kern_louis)()        = v4wt_flush_kern_cache_louis
        (*flush_user_all)()          = v4wt_flush_user_cache_all
        (*flush_user_range)()        = v4wt_flush_user_cache_range
        (*coherent_kern_range)()     = v4wt_coherent_kern_range
        (*coherent_user_range)()     = v4wt_coherent_user_range
        (*flush_kern_dcache_area)()  = v4wt_flush_kern_dcache_area
        (*dma_map_area)()            = v4wt_dma_map_area
        (*dma_unmap_area)()          = v4wt_dma_unmap_area
        (*dma_flush_range)()         = v4wt_dma_flush_range
    } *cache;
};
```

在 setup_processor()中，继续配置__cpu_architecture = CPU_ARCH_ARMv4T，这里需要注意，在 S3C2410 这个配置下，MULTI_CPU/MULTI_CACHE 被使能，MULTI_TLB/MULTI_USER 则没有被定义。

```
597
598     cpu_name = list->cpu_name;
599     __cpu_architecture = __get_cpu_architecture();
600
601 #ifdef MULTI_CPU
602     processor = *list->proc;
603 #endif
604 #ifdef MULTI_TLB
605     cpu_tlb = *list->tlb;
606 #endif
607 #ifdef MULTI_USER
608     cpu_user = *list->user;
609 #endif
610 #ifdef MULTI_CACHE
611     cpu_cache = *list->cache;
612 #endif
613
```

后面的代码大多没有实际执行，主要的就是：

- 通过 init_default_cache_policy()更新了 cache_policies 数组中的.pmd
- 将 cacheid 设置为 CACHEID_VIVT
- cpu_init()中通过不断切换 cpsr_c 的状态，在不同的 CPU 模式下配置对应的异常模式的 sp 指针的位置，在第一阶段中只用保存少量的寄存器，所以这个堆栈很小，只有 3 个 DWORD 大小：

```
133 struct stack {
134     u32 irq[3];
135     u32 abt[3];
136     u32 und[3];
137     u32 fiq[3];
138 } ____cacheline_aligned;
139
```

setup_arch - 准备 machine_desc 与 atags 传递

在前面的启动参数的传递的分析中，U-Boot 会向 Linux 传递三个参数，其中第二个参数是 machine_id, 第三个参数是 atags 的指针。着两个参数，分别会存储在__machine_arch_type 与__atags_pointer 中，其中__machine_arch_type 会被 Linux 内核用来定位对应的 machine_desc, 而 atags 则会被进一步的 parser。

由于 S3C2410 这个配置下还没有引入设备树，仍然使用原有的 ATAGS 的方式从 U-Boot 传递配置信息，所以就会在 setup_arch 阶段执行:

```
905     mdesc = setup_machine_tags(__atags_pointer, __machine_arch_type);
906     machine_desc = mdesc;
907     machine_name = mdesc->name;
```

首先看__machine_arch_type 的转换，在 arch/arm/include/asm/mach/arch.h 中:

```
75  */
76  extern const struct machine_desc __arch_info_begin[], __arch_info_end[];
77  #define for_each_machine_desc(p)            \
78      for (p = __arch_info_begin; p < __arch_info_end; p++)
79
```

而__arch_info_begin 和__arch_info_end 的定义则来自 vmlinux.lds.S:

```
191     .init.arch.info : {
192         __arch_info_begin = .;
193         *(.arch.info.init)
194         __arch_info_end = .;
195     }
```

对应的，在 arch/arm/include/asm/mach/arch.h 中:

```
84  #define MACHINE_START(_type,_name)          \
85  static const struct machine_desc __mach_desc_##_type    \
86   __used                          \
87   __attribute__((__section__(".arch.info.init"))) = {    \
88      .nr     = MACH_TYPE_##_type,        \
89      .name       = _name,
90
91  #define MACHINE_END             \
92  };
```

也就是任何对应一个 Linux 支持的平台，就需要定义 MACHINE_START()和 MACHINE_END，例如在 arch/arm/mach-s3c24xx/mach-smdk2410.c 中:

```
120 MACHINE_START(SMDK2410, "SMDK2410") /* @TODO: request a new identifier and switch
121                     * to SMDK2410 */
122     /* Maintainer: Jonas Dietsche */
123     .atag_offset    = 0x100,
124     .map_io     = smdk2410_map_io,
125     .init_irq   = s3c2410_init_irq,
126     .init_machine   = smdk2410_init,
127     .init_time  = smdk2410_init_time,
128 MACHINE_END
```

这样就会定义一个名为"SMDK2410"和设备号为 MACH_TYPE_SMDK2410(就是 193， 参见启动参数的传递)

那么定义了的所有的 machine_desc，都会被存储在.arch.info.init 这个 section 中，而 Linux 则会通过__arch_info_begin 和__arch_info_end, 也就是前面提到的 for_each_machine_desc(p) 来遍历这个表，回到 setup_machine_tags：

```
193     for_each_machine_desc(p)
194         if (machine_nr == p->nr) {
195             pr_info("Machine: %s\n", p->name);
196             mdesc = p;
197             break;
198         }
```

就在此处确定了 U-Boot 传递来的 machine_id 对应的具体的 machine_desc。

而对于 atags，在前面的分析中，__atags_pointer 是在__mmap_switched_data 阶段初始化的，指向从 U-Boot 传递来的结构体，而相对的，在 Linux 中则要实现一套机制，类似于{tag-name, function}这样的结构体列表，对于从 U-Boot 传递来的 ATAGS，每一个 tag 如果能匹配上 tag-name, 则执行对应的 function。

与前面提到的 machine_desc 一样，在 vmlinux.lds.S 中：

```
196     .init.tagtable : {
197         __tagtable_begin = .;
198         *(.taglist.init)
199         __tagtable_end = .;
200     }
```

在 arch/arm/include/asm/setup.h 中：

```
20  #define __tag __used __attribute__((__section__(".taglist.init")))
21  #define __tagtable(tag, fn) \
22  static const struct tagtable __tagtable_##fn __tag = { tag, fn }
23
```

也就是任何一个实现了__tagtable(name, callback)的地方，就会产生一个__tagtable_callback = {name, callback}的结构体，存储在.taglist.init 这个 section 中， 比如对于 ATAG_CMDLINE, 就有对应的定义：

```
140
141 __tagtable(ATAG_CMDLINE, parse_tag_cmdline);
142
```

或者对于 ATAG_MEM：

```
71  __tagtable(ATAG_MEM, parse_tag_mem32);
```

也就是匹配到 ATAG_CMDLINE 时，将会执行 parse_tag_cmdline()这个 API 或者匹配到 ATAG_MEM 时执行 parse_tag_mem32()。

再看 setup_machine_tags 中的代码：

```
227     if (tags->hdr.tag == ATAG_CORE) {
228         if (memblock_phys_mem_size())
229             squash_mem_tags(tags);
230         save_atags(tags);
231         parse_tags(tags);
232     }
```

Linux 会对传入的每一个 atag，通过__tagtable_begin 和__tagtable_end 遍历 Linux 中的 ATAG 列表：

```
148 static int __init parse_tag(const struct tag *tag)
149 {
150     extern struct tagtable __tagtable_begin, __tagtable_end;
151     struct tagtable *t;
152
153     for (t = &__tagtable_begin; t < &__tagtable_end; t++)
154         if (tag->hdr.tag == t->tag) {
155             t->parse(tag);
156             break;
157         }
158
159     return t < &__tagtable_end;
160 }
161
162 /*
163  * Parse all tags in the list, checking both the global and architecture
164  * specific tag tables.
165  */
166 static void __init parse_tags(const struct tag *t)
167 {
168     for (; t->hdr.size; t = tag_next(t))
169         if (!parse_tag(t))
170             pr_warn("Ignoring unrecognised tag 0x%08x\n",
171                 t->hdr.tag);
172 }
```

了解了这样的机制后，再看一下 command line 是如何传递的，在 Linux 编译时，可以通过 Kconfig 指定一个默认的启动参数：

```
(0x0) Compressed ROM boot loader BSS address
(noinitrd root=/dev/mtdblock2 init=/linuxrc console=ttySAC0) Default kernel command string
      Kernel command line type (Use bootloader kernel arguments if available)  --->
[ ] Kernel Execute-In-Place from ROM
```

这个参数就对应着 default_command_line:

```
34  static char default_command_line[COMMAND_LINE_SIZE] __initdata = CONFIG_CMDLINE;
```

而在 U-Boot 的 env 中，也可以配置这个 bootarg，在启动参数的传递中分析过，这个字符串会被 U-Boot 放到名为 ATAG_CMDLINE 的 atag 下。在 Linux 下，由于定义了：

```
140
141  __tagtable(ATAG_CMDLINE, parse_tag_cmdline);
142
```

所以在上面 parse_tag 的时候，由于匹配了 ATAG_CMDLINE, 那么 Linux 就会执行:

```
126 static int __init parse_tag_cmdline(const struct tag *tag)
127 {
128 #if defined(CONFIG_CMDLINE_EXTEND)
129     strlcat(default_command_line, " ", COMMAND_LINE_SIZE);
130     strlcat(default_command_line, tag->u.cmdline.cmdline,
131         COMMAND_LINE_SIZE);
132 #elif defined(CONFIG_CMDLINE_FORCE)
133     pr_warn("Ignoring tag cmdline (using the default kernel command line)\n");
134 #else
135     strlcpy(default_command_line, tag->u.cmdline.cmdline,
136         COMMAND_LINE_SIZE);
137 #endif
138     return 0;
139 }
```

就会把 U-Boot 中传递的 TAG 名为 ATAG_CMDLINE 中的数据，替换掉 Linux 默认的 default_command_line 中的内容。最后 Linux 又会把这个字串拷贝至 boot_command_line 中并最终拷贝到 cmd_line 中：

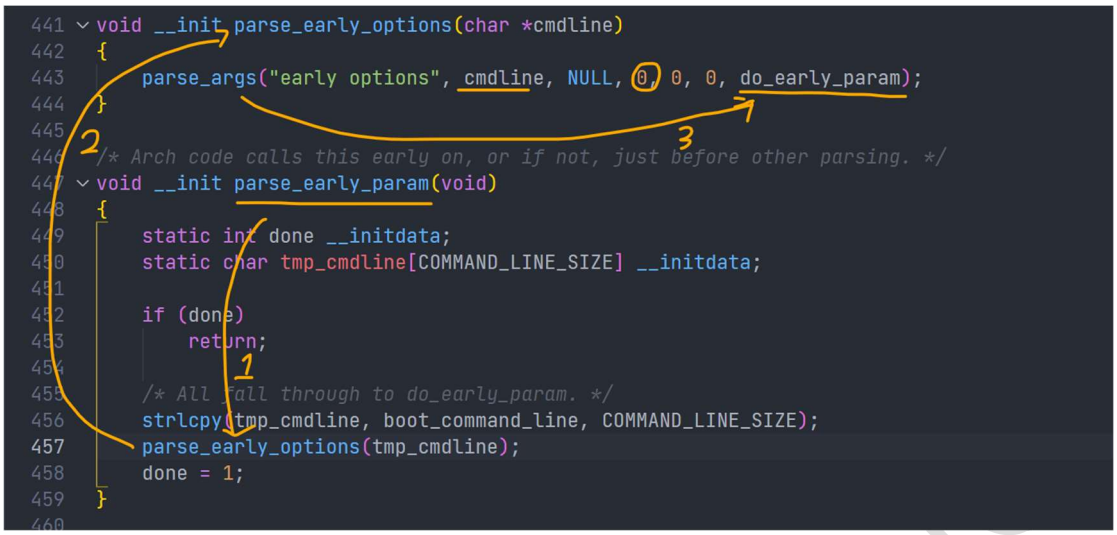
```
234     /* parse_early_param needs a boot_command_line */
235     strlcpy(boot_command_line, from, COMMAND_LINE_SIZE);
236
```

```
918     /* populate cmd_line too for later use, preserving boot_command_line */
919     strlcpy(cmd_line, boot_command_line, COMMAND_LINE_SIZE);
920     *cmdline_p = cmd_line;
```

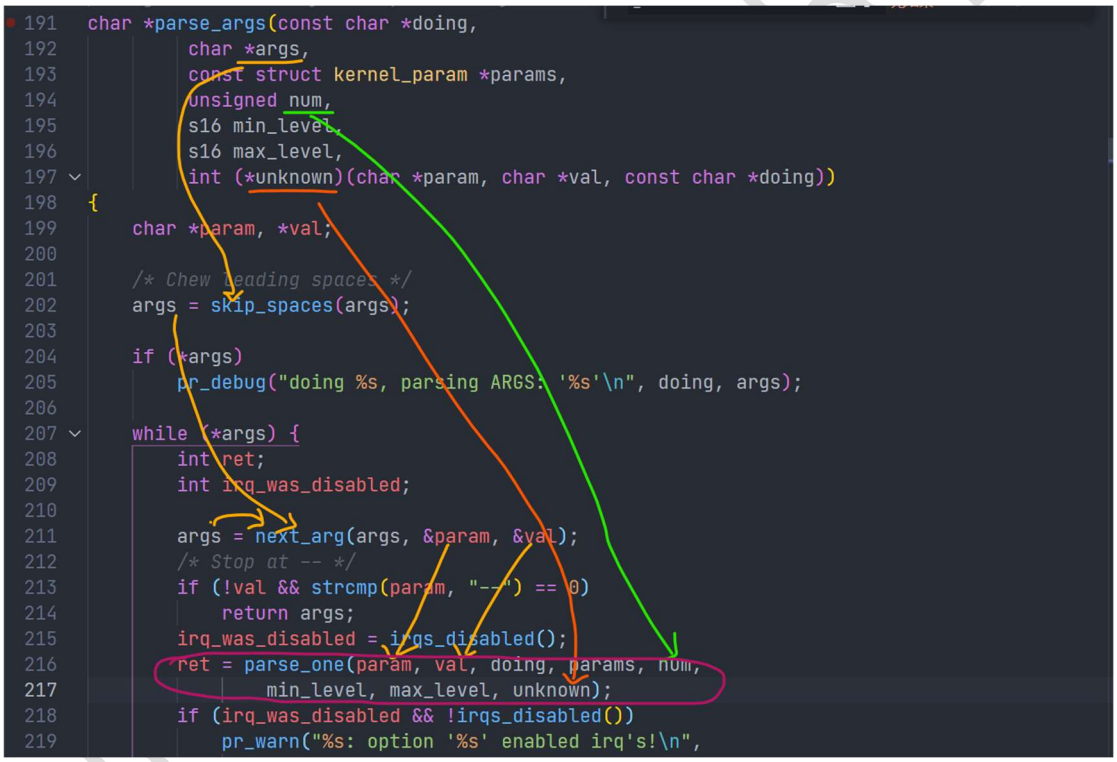
在准备好了 command_line 字串后，就开始了早期 bootargs 的读取和执行：

```
921
922     parse_early_param();
923
```

这里执行的流程：



```c
void __init parse_early_options(char *cmdline)
{
    parse_args("early options", cmdline, NULL, 0, 0, 0, do_early_param);
}

/* Arch code calls this early on, or if not, just before other parsing. */
void __init parse_early_param(void)
{
    static int done __initdata;
    static char tmp_cmdline[COMMAND_LINE_SIZE] __initdata;

    if (done)
        return;

    /* All fall through to do_early_param. */
    strlcpy(tmp_cmdline, boot_command_line, COMMAND_LINE_SIZE);
    parse_early_options(tmp_cmdline);
    done = 1;
}
```



```c
char *parse_args(const char *doing,
         char *args,
         const struct kernel_param *params,
         unsigned num,
         s16 min_level,
         s16 max_level,
         int (*unknown)(char *param, char *val, const char *doing))
{
    char *param, *val;

    /* Chew leading spaces */
    args = skip_spaces(args);

    if (*args)
        pr_debug("doing %s, parsing ARGS: '%s'\n", doing, args);

    while (*args) {
        int ret;
        int irq_was_disabled;

        args = next_arg(args, &param, &val);
        /* Stop at -- */
        if (!val && strcmp(param, "--") == 0)
            return args;
        irq_was_disabled = irqs_disabled();
        ret = parse_one(param, val, doing, params, num,
                min_level, max_level, unknown);
        if (irq_was_disabled && !irqs_disabled())
            pr_warn("%s: option '%s' enabled irq's!\n",
```

在 parse_args 中，将传递的 command_line, 例如"noinitrd root=/dev/mtdblock2 init=/linuxrc console=ttySAC0"（不含双引号）会被按照空格分段，分别把分开的部分按照 param=val 的格式，送入 parse_one()函数进一步执行，但是由于 num=0, 再 parse_one()函数中会直接执行传入和回调函数(*unknown)(), 也就是前面的 do_early_param():

```
/* check for early params. */
static int __init do_early_param(char *param, char *val, const char *unused)
{
    const struct obs_kernel_param *p;

    for (p = __setup_start; p < __setup_end; p++) {
        if ((p->early && parameq(param, p->str)) ||
            (strcmp(param, "console") == 0 &&
             strcmp(p->str, "earlycon") == 0)
        ) {
            if (p->setup_func(val) != 0)
                pr_warn("Malformed early option '%s'\n", param);
        }
    }
    /* We accept everything at this stage. */
    return 0;
}
```

这里又一次的出现了 linker list:

```
#define INIT_SETUP(initsetup_align)                 \
        . = ALIGN(initsetup_align);             \
        VMLINUX_SYMBOL(__setup_start) = .;          \
        *(.init.setup)                      \
        VMLINUX_SYMBOL(__setup_end) = .;
```

```
#define __setup_param(str, unique_id, fn, early)            \
    static const char __setup_str_##unique_id[] __initconst \
        __aligned(1) = str; \
    static struct obs_kernel_param __setup_##unique_id  \
        __used __section(.init.setup)           \
        __attribute__((aligned((sizeof(long)))))    \
        = { __setup_str_##unique_id, fn, early }

#define __setup(str, fn)                    \
    __setup_param(str, fn, fn, 0)

/* NOTE: fn is as per module_param, not __setup!  Emits warning if fn
 * returns non-zero. */
#define early_param(str, fn)                    \
    __setup_param(str, fn, fn, 1)
```

也就是 Linux 中会通过__setup(str, fn) 和 early_param(str, fn)注册各种 param 和对应的回调函数:

```
__setup("noinitrd", no_initrd);
```

```
__setup("root=", root_dev_setup);
```

```
__setup("init=", init_setup);
```

```
__setup("console=", console_setup);
```

```
early_param("initrd", early_initrd);
```

```
early_param("boot_delay", boot_delay_setup);
```

但只有通过 early_param 注册的才会在结构体中把标志位"early"置位，然后再 do_early_param()中判断这个标志位，从而决定是否进一步执行对应的回调函数。对前面提到的例子而言，这些参数都不会被 do_early_param 执行。

在 setup_arch 的 atags 传递阶段，另一个重要的参数传递就是 memory info，在 U-Boot 阶段 board_f.c 配置了 bd->bi_dram[]的信息，配置物理的内存为 start=0x3000_0000, size=0x0400_0000(64MB):

```
227  __weak void dram_init_banksize(void)
228  {
229  #if defined(CONFIG_NR_DRAM_BANKS) && defined(CONFIG_SYS_SDRAM_BASE)
230      gd->bd->bi_dram[0].start = CONFIG_SYS_SDRAM_BASE;
231      gd->bd->bi_dram[0].size = get_effective_memsize();
232  #endif
233  }
```

这些信息最终会传递给 ATAGS:

在 boot_prep_linux():

```
226          if (BOOTM_ENABLE_MEMORY_TAGS)
227              setup_memory_tags(gd->bd);
```

```
104  static void setup_memory_tags(bd_t *bd)
105  {
106      int i;
107
108      for (i = 0; i < CONFIG_NR_DRAM_BANKS; i++) {
109          params->hdr.tag = ATAG_MEM;
110          params->hdr.size = tag_size (tag_mem32);
111
112          params->u.mem.start = bd->bi_dram[i].start;
113          params->u.mem.size = bd->bi_dram[i].size;
114
115          params = tag_next (params);
116      }
117  }
```

在 Linux 中，在 parser_tag()阶段会匹配到 ATAG_MEM，然后执行对应的回调:

```
66  static int __init parse_tag_mem32(const struct tag *tag)
67  {
68      return arm_add_memory(tag->u.mem.start, tag->u.mem.size);
69  }
70
71  __tagtable(ATAG_MEM, parse_tag_mem32);
```

在 arm_add_memory()中，最终会执行 memblock.c 中的 memblock_add(start, size), 将这个内存的信息存储在 Linux 全局的 memblock.memory 中：

```
583  int __init_memblock memblock_add(phys_addr_t base, phys_addr_t size)
584  {
585      return memblock_add_range(&memblock.memory, base, size,
586                                MAX_NUMNODES, 0);
587  }
```

HIGHMEM 的确定

在完成了 parse_early_param()后，继续执行后续代码：

```
922      parse_early_param();
923
924      early_paging_init(mdesc, lookup_processor_type(read_cpuid_id()));
925      setup_dma_zone(mdesc);
926      sanity_check_meminfo();
927      arm_memblock_init(mdesc);
```

由于没有定义 CONFIG_ARM_LPAE，注册 machine_desc.init_meminfo 和 CONFIG_ZONE_DMA，所以实际执行的第一个函数则是 sanity_check_meminfo()，在这个函数中 Linux 将会设定 highmem 的位置，highmem 的空间将会是用户空间，而相对的 lowmem 则对应着内核空间。

首先在 arch/arm/mm/mmu.c 中定义了 vmalloc_min:

```
1038  static void * __initdata vmalloc_min =
1039      (void *)(VMALLOC_END - (240 << 20) - VMALLOC_OFFSET);
```

这个初始化的数值是 0xef80_0000。

由于前面分析，ATAG 传入的 memory info 是 base=0x3000_0000, size=0x0400_0000, 则可以获得如下计算结果：

```
for_each_memblock(memory, reg) {
    phys_addr_t block_start = reg->base; //0x3000_0000
    phys_addr_t block_end = reg->base + reg->size; //0x3400_0000
    phys_addr_t size_limit = reg->size; //0x0400_0000
```

```
    if (reg->base >= vmalloc_limit)
        highmem = 1;
    else
        size_limit = vmalloc_limit - reg->base; //size_limit=0x4080_0000
```

那么在后面的计算就能得到 arm_lowmem_limit=0x0400_0000:

```
if (!highmem) {
    if (block_end > arm_lowmem_limit) {
        if (reg->size > size_limit)
            arm_lowmem_limit = vmalloc_limit;
        else
            arm_lowmem_limit = block_end; //0x0400_0000
    }
```

相应的，high_memory 就可以计算得到 0xc400_0000:

```
1139
1140    high_memory = __va(arm_lowmem_limit - 1) + 1;
1141
```

paging_init