

# 《嵌入式系统》

## （第六讲）

厦门大学信息学院软件工程系 曾文华

2024年10月15日

- 第1章：嵌入式系统概述
- 第2章：ARM处理器和指令集
- 第3章：嵌入式Linux操作系统
- 第4章：嵌入式软件编程技术
- 第5章：开发环境和调试技术
- 第6章：Boot Loader技术
- 第7章：ARM Linux内核
- 第8章：文件系统
- 第9章：设备驱动程序设计基础
- 第10章：字符设备和驱动程序设计
- 第11章：块设备和驱动程序设计
- 第12章：网络设备驱动程序开发
- 第13章：嵌入式GUI及应用程序设计
- 第14章：Android操作系统（增加）



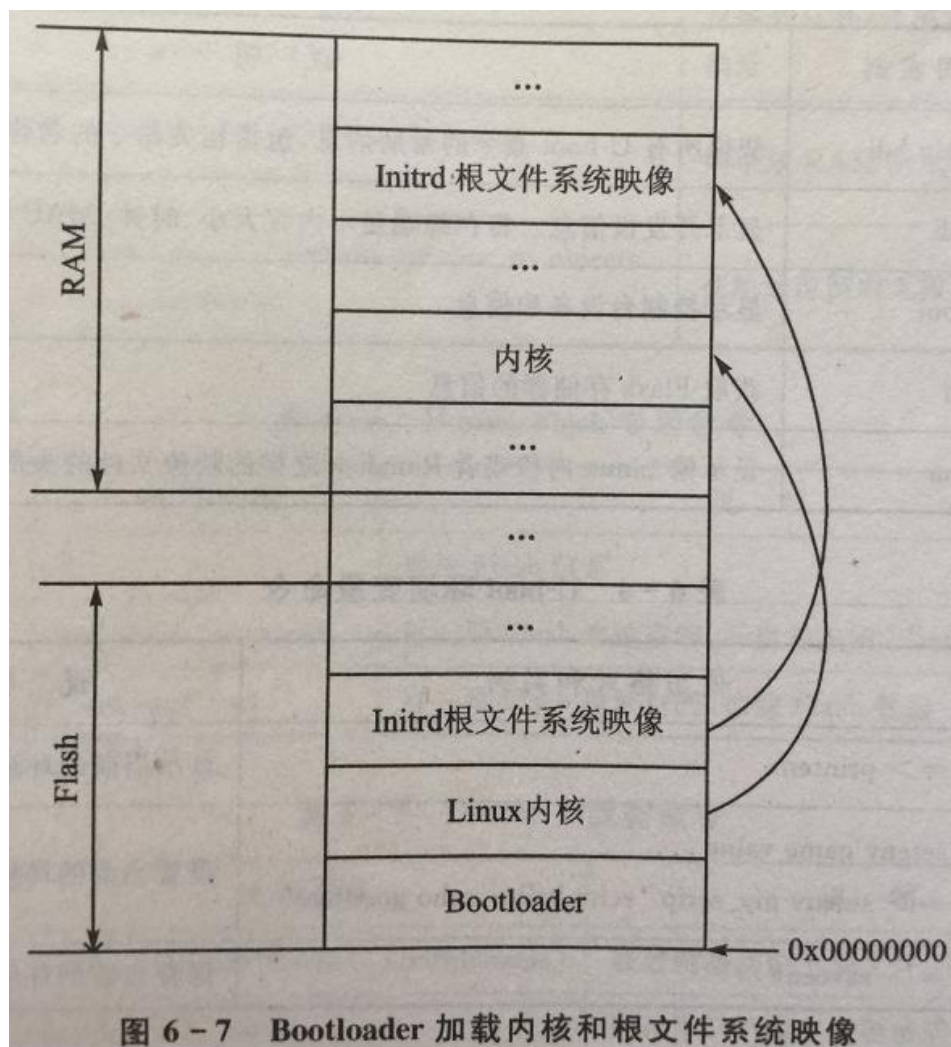
# 第6章 Boot Loader技术

- 6.1 Boot Loader基本概念
- 6.2 Boot Loader典型结构
- 6.3 U-Boot简介
- 6.4 vivi简介

# 6.1 Boot Loader基本概念

- **Boot Loader**（**引导程序**）是在操作系统内核运行之前运行的一段小程序，**Boot Loader**初始化硬件设备和建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。
- 嵌入式Linux系统启动后，先执行**Boot Loader**，进行硬件和内存的初始化工作，然后加载Linux内核和根文件系统，完成Linux系统的启动。

# Bootloader加载内核和根文件系统映像



## FS3399M4实验箱的Bootloader: U-Boot 2017.09

```
linux@localhost:~$  
linux@localhost:~$  
linux@localhost:~$ 鎚滾  . @NN^Nt 堞  
U-Boot 2017.09 (Aug 31 2023 - 02:03:09 +0000)  
  
Model: Rockchip RK3399 Evaluation Board  
PreSerial: 2  
DRAM: 2 GiB  
System: init  
Relocation Offset is: 7dbdf000  
Using default environment  
  
[debug] should_load_env()  
[debug] env_load()  
dwmmc@fe320000: 1, sdhci@fe330000: 0  
Bootdev(atags): mmc 0  
MMC0: HS400, 150Mhz  
PartType: EFI  
[debug]get env lcdtype:<NULL>  
get part misc fail -1  
boot mode: None  
init_resource_list: Load resource from boot second pos  
Load FDT from boot part  
*** Warning use default panel:mipi070_M4 ***  
**** fs3399_linux_M4_mipi070.dts
```

```
## Booting Android Image at 0x0027f800 ...
Kernel load addr 0x00280000 size 19133 KiB
## Flattened Device Tree blob at 08300000
   Booting using the fdt blob at 0x8300000
   XIP Kernel Image ... OK
   'reserved-memory' region@110000: addr=110000 size=f0000
   Using Device Tree in place at 0000000008300000, end 000000000831cd58
Adding bank: 0x00200000 - 0x08400000 (size: 0x08200000)
Adding bank: 0x0a200000 - 0x80000000 (size: 0x75e00000)
Total: 6337.175 ms
```

Starting kernel ...

加载Linux内核

```
[ 0.000000] Booting Linux on physical CPU 0x0
[ 0.000000] Initializing cgroup subsys cpuset
[ 0.000000] Initializing cgroup subsys cpu
[ 0.000000] Initializing cgroup subsys cpuacct
[ 0.000000] Linux version 4.4.179 (
[ 0.000000] Boot CPU: AArch64 Processor [410fd034]
[ 0.000000] earlycon: Early serial console at MMI032 0xff1a0000 (options '')
[ 0.000000] bootconsole [uart0] enabled
[ 0.000000] psci: probing for conduit method from DT.
[ 0.000000] psci: PSCIv1.0 detected in firmware.
[ 0.000000] psci: Using standard PSCI v0.2 function IDs
[ 0.000000] psci: Trusted OS migration not required
[ 0.000000] PERCPU: Embedded 21 pages/cpu @fffffc07fedf000 s45800 r8192 d32024 u86016
```

实验箱Linux内核版本: 4.4.179

Jun 19 11:53:40 CST 2024

```
acik_perilp0 266666 KHz
hclk_perilp0 88888 KHz
pclk_perilp0 44444 KHz
hclk_perilp1 100000 KHz
pclk_perilp1 50000 KHz
Net: eth0: ethernet@fe300000
Hit key to stop autoboot('CTRL+C'): 0
=>
=>
=>
=>
=>
=>
=>
=>
```

进入“=>”状态

```
[ 28.924330] rc.local[733]: chmod: cannot access
[ 28.932467] rc.local[733]: chmod: cannot access
[ 28.940610] rc.local[733]: /home/linux/qt5applica

Ubuntu 16.04.7 LTS localhost.localdomain ttyFIQ0

localhost login: █
```

登录状态

```
Ubuntu 16.04.7 LTS localhost.localdomain ttyFIQ0

localhost login: linux
Password:
Last login: Thu Feb 11 16:30:02 UTC 2016 on ttyFIQ0
Welcome to Ubuntu 16.04.7 LTS (GNU/Linux 4.4.179 aarch64)

* Documentation:  https://help.ubuntu.com
* Management:    https://landscape.canonical.com
* Support:       https://ubuntu.com/advantage

50 packages can be updated.
30 of these updates are security updates.
To see these additional updates run: apt list --upgradable

linux@localhost:~$ █
```

登录进去后



- **6.1.1 Boot Loader所支持的硬件环境**

- **Boot Loader依赖于：**

- ① 嵌入式CPU（实验箱MPU的RK3399处理器）。

- ② 嵌入式板级设备的配置（实验箱上的各种硬件）。

## • 6.1.2 Boot Loader的安装地址

– Boot Loader的安装地址（即ARM处理器的复位启动地址）：**0x00000000**

– 固态存储器（Flash存储器）的典型空间分配结构：

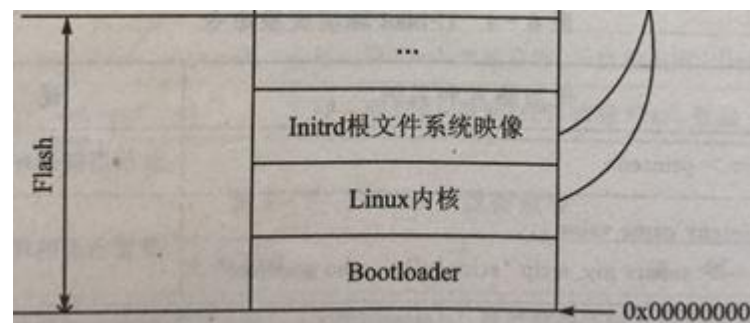
① Boot Loader

② 内核的启动参数（Boot parameters）

③ 内核映像

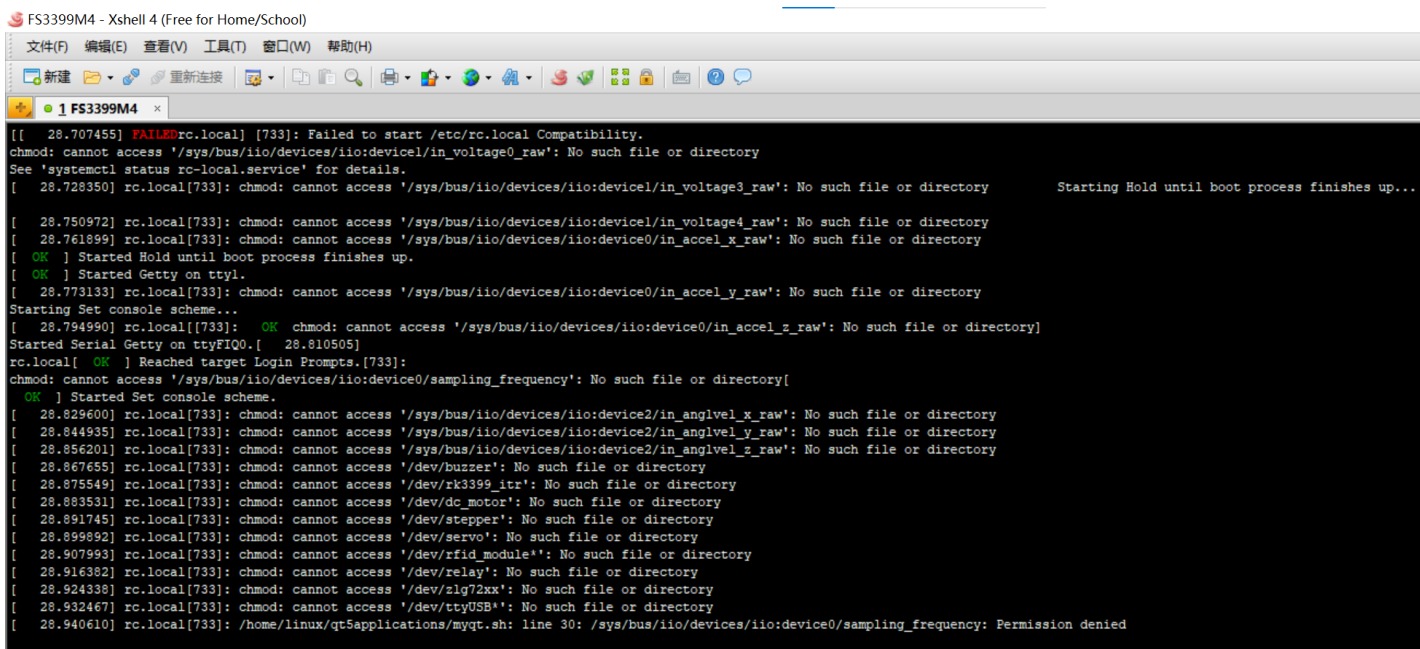
④ 根文件系统映像

Linux内核



## • 6.1.3 Boot Loader相关的设备和基址

- 主机（宿主机，Ubuntu）和目标机（目标板，实验箱）之间一般通过**串口（Xshell串口超级终端）**建立连接，**Boot Loader**软件在执行时通常会通过串口来进行输入、输出，比如：输出打印信息到串口，从串口读取用户控制字符等。
- **Boot Loader**的基址：**0x00000000**



```
FS3399M4 - Xshell 4 (Free for Home/School)
文件(F) 编辑(E) 查看(V) 工具(T) 窗口(W) 帮助(H)
新建 重新连接
1 FS3399M4 x
[[ 28.707455] FAILEDrc.local[733]: Failed to start /etc/rc.local Compatibility.
chmod: cannot access '/sys/bus/iio/devices/iio:device1/in_voltage0_raw': No such file or directory
See 'systemctl status rc-local.service' for details.
[ 28.728350] rc.local[733]: chmod: cannot access '/sys/bus/iio/devices/iio:device1/in_voltage3_raw': No such file or directory
Starting Hold until boot process finishes up...
[ 28.750972] rc.local[733]: chmod: cannot access '/sys/bus/iio/devices/iio:device1/in_voltage4_raw': No such file or directory
[ 28.761899] rc.local[733]: chmod: cannot access '/sys/bus/iio/devices/iio:device0/in_accel_x_raw': No such file or directory
[ OK ] Started Hold until boot process finishes up.
[ OK ] Started Getty on tty1.
[ 28.773133] rc.local[733]: chmod: cannot access '/sys/bus/iio/devices/iio:device0/in_accel_y_raw': No such file or directory
Starting Set console scheme...
[ 28.794990] rc.local[733]: OK chmod: cannot access '/sys/bus/iio/devices/iio:device0/in_accel_z_raw': No such file or directory
Started Serial Getty on ttyE100.[ 28.810505]
rc.local[ OK ] Reached target Login Prompts.[733]:
chmod: cannot access '/sys/bus/iio/devices/iio:device0/sampling_frequency': No such file or directory
[ OK ] Started Set console scheme.
[ 28.829600] rc.local[733]: chmod: cannot access '/sys/bus/iio/devices/iio:device2/in_anglvel_x_raw': No such file or directory
[ 28.844935] rc.local[733]: chmod: cannot access '/sys/bus/iio/devices/iio:device2/in_anglvel_y_raw': No such file or directory
[ 28.856201] rc.local[733]: chmod: cannot access '/sys/bus/iio/devices/iio:device2/in_anglvel_z_raw': No such file or directory
[ 28.867655] rc.local[733]: chmod: cannot access '/dev/buzzer': No such file or directory
[ 28.875549] rc.local[733]: chmod: cannot access '/dev/rk3399_itr': No such file or directory
[ 28.883531] rc.local[733]: chmod: cannot access '/dev/dc_motor': No such file or directory
[ 28.891745] rc.local[733]: chmod: cannot access '/dev/stepper': No such file or directory
[ 28.899892] rc.local[733]: chmod: cannot access '/dev/servo': No such file or directory
[ 28.907993] rc.local[733]: chmod: cannot access '/dev/rfid_module': No such file or directory
[ 28.916382] rc.local[733]: chmod: cannot access '/dev/relay': No such file or directory
[ 28.924338] rc.local[733]: chmod: cannot access '/dev/zlg72xx': No such file or directory
[ 28.932467] rc.local[733]: chmod: cannot access '/dev/ttyUSB': No such file or directory
[ 28.940610] rc.local[733]: /home/linux/Qt5Applications/myqt.sh: line 30: /sys/bus/iio/devices/iio:device0/sampling_frequency: Permission denied
```

- **6.1.4 Boot Loader的启动过程**

- 启动过程包括两个阶段：

- 阶段1

- 阶段2

- 阶段1完成初始化硬件，为阶段2准备内存空间，并将阶段2复制到内存中，设置堆栈，然后跳转到阶段2。

## • 6.1.5 Boot Loader的操作模式

### – Boot Loader的两种模式：

- **启动加载模式**：也称为自主模式，也即Boot Loader从目标机（实验箱）上的某个固态存储设备（通常为Flash存储器）上将操作系统加载到RAM（通常为SDRAM）中运行，整个过程并没有用户的介入。这种模式是Boot Loader的正常工作模式。
- **下载模式**：在这种模式下目标机（实验箱）上的Boot Loader将通过串口连接或网络连接等通信手段从主机（宿主机，Ubuntu）下载文件（内核映像、根文件系统映像），从主机下载的文件通常首先被Boot Loader保存到目标机的RAM（SDRAM）中，然后再被Boot Loader写（烧写）到目标机上的固态存储设备（Flash存储器）中。

# 启动加载模式

```
[ 28.924338] rc.local[733]: chmod: cannot access  
[ 28.932467] rc.local[733]: chmod: cannot access  
[ 28.940610] rc.local[733]: /home/linux/qt5applica  
  
Ubuntu 16.04.7 LTS localhost.localdomain ttyFIQ0  
  
localhost login: █
```

登录状态（启动加载模式，正常工作模式）

```
Ubuntu 16.04.7 LTS localhost.localdomain ttyFIQ0  
  
localhost login: linux  
Password:  
Last login: Thu Feb 11 16:30:02 UTC 2016 on ttyFIQ0  
Welcome to Ubuntu 16.04.7 LTS (GNU/Linux 4.4.179 aarch64)  
  
* Documentation:  https://help.ubuntu.com  
* Management:    https://landscape.canonical.com  
* Support:        https://ubuntu.com/advantage  
  
50 packages can be updated.  
30 of these updates are security updates.  
To see these additional updates run: apt list --upgradable  
  
linux@localhost:~$ █
```

登录进去后

# 下载模式

```
ac1k_perilp0 266666 KHz  
hclk_perilp0 88888 KHz  
pclk_perilp0 44444 KHz  
hclk_perilp1 100000 KHz  
pclk_perilp1 50000 KHz  
Net: eth0: ethernet@fe300000  
Hit key to stop autoboot('CTRL+C'): 0  
=>  
=>  
=>  
=>  
=>  
=>  
=>  
=>  
=>
```

下载模式：“=>” 状态

- 实验箱的两种模式的**切换**：在打开实验箱电源（或者按下实验箱的**Reset**按钮）后，出现倒计时后，按**Ctrl+C**，则会进入下载模式（“=>”状态）；否则进入启动加载模式；在下载模式下（“=>”状态），执行“**reset**”命令，也会进入启动加载模式。

```
pc1k_perilp0 44444 KHz
hclk_perilp1 100000 KHz
pclk_perilp1 50000 KHz
Net:  eth0: ethernet@fe300000
Hit key to stop autoboot('CTRL+C'):  0
=>
=>
=>
=>
=>
=>
=>
=>
=> reset
```



## – 常用的Boot Loader:

- ① **U-Boot**: 全称 Universal Boot Loader, 是由开源项目 PPCBoot发展起来的, ARMboot并入了PPCBoot, 和其他一些arch的Loader合称U-Boot。
- ② **vivi**: 是韩国mizi 公司开发的Boot Loader, 适用于ARM9处理器。vivi有两种工作模式: 启动加载模式和下载模式。启动加载模式可以在一段时间后(这个时间可更改)自行启动Linux内核, 这是vivi的默认模式。在下载模式下, vivi为用户提供一个命令行接口, 通过接口可以使用vivi提供的一些命令。
- ③ **Blob**: 全称Boot Loader Object, 是由Jan-Derk Bakker和Erik Mouw发布的, 是专门为StrongARM 构架下的LART设计的Boot Loader。

- **6.1.6 Boot Loader与主机之间的通信设备及协议**

- **串口**：目标机使用串口与主机相连，这时的传输协议通常是xmodem/ymodem/zmodem中的一种。
- **网口**：目标机使用网口与主机相连，用网络连接的方式传输文件，这时使用的协议多为**TFTP**（简单文件传输协议）。

## 6.2 Boot Loader典型结构

- **Boot Loader的阶段1**：主要包含依赖于CPU的体系结构硬件初始化的代码，通常都用汇编语言来实现。这个阶段的任务有（5个任务）：
  - ① 基本的硬件设备初始化（屏蔽所有的中断、关闭处理器内部指令/数据Cache等）
  - ② 为加载Boot Loader的阶段2准备RAM空间
  - ③ 复制Boot Loader的阶段2代码到RAM
  - ④ 设置堆栈
  - ⑤ 跳转到阶段2的c程序入口点
- **Boot Loader的阶段2**：通常用c语言完成，以便实现更复杂的功能，也使程序有更好的可读性和可移植性。这个阶段的任务有（5个任务）：
  - ① 初始化本阶段要使用到的硬件设备
  - ② 检测系统内存映射
  - ③ 将内核映像和根文件系统映像从Flash读到RAM
  - ④ 为内核设置启动参数
  - ⑤ 调用内核

## • 6.2.1 Boot Loader阶段1介绍（5个任务）

### – 任务1：基本的硬件设备初始化

- 屏蔽所有的中断（通过写CPU的中断屏蔽寄存器或状态寄存器（如CPSR）来完成）
- 设置CPU的速度和时钟频率
- RAM初始化
- 初始化LED（通过LED显示系统的状态是OK还是Error）

### – 任务2：为加载阶段2准备RAM（SDARM）空间

- 阶段2加载到RAM中，通常准备1MB的RAM空间，放在整个RAM的最顶端，并且要对这个1MB的空间进行测试

### – 任务3：复制阶段2的代码到RAM（SDRAM）

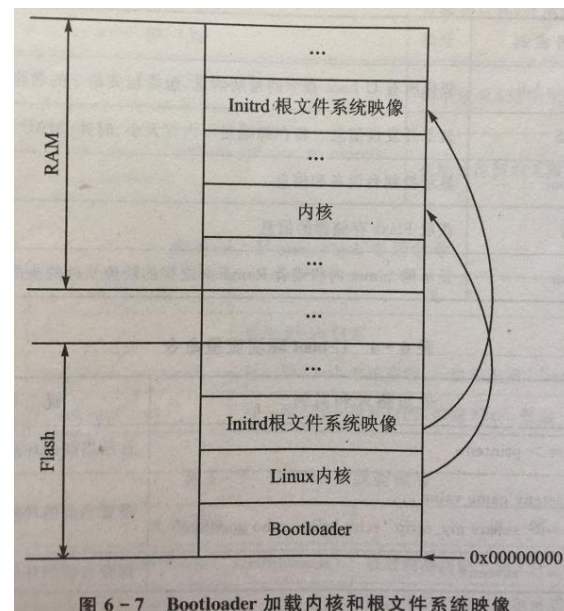
- 阶段2的映像Flash中的起始地址和终止地址
- RAM空间的起始地址

### – 任务4：设置堆栈指针

- $SP = stage2\_end - 4$

### – 任务5：跳转到阶段2的C程序入口点

- 通过修改PC的值来实现



## • 6.2.2 Boot Loader阶段2介绍（5个任务）

- 利用trampoline（弹簧床）的概念，即用汇编语言写一段 **trampoline小程序**，并将这段trampoline小程序来作为阶段2可执行映象的执行入口点；然后我们可以在trampoline汇编小程序中用CPU跳转指令跳入 main() 函数中去执行，而当 main() 函数返回时，CPU 执行路径显然再次回到我们的 trampoline 程序。
- trampoline小程序：

.text	@代码段
.globl _trampoline	@声明全局
_trampoline:	
bl main	@跳转到main
b _trampoline	@跳转到_trampoline

## – 任务1：初始化本阶段要使用到的硬件设备

- 初始化至少一个串口，以便和终端用户进行 I/O 输出信息
- 初始化计时器等

## – 任务2：检测系统内存映射

- 所谓内存映射就是指在整个 4GB 物理地址空间中有哪些地址范围被分配用来寻址系统的 RAM 单元
- 虽然 CPU 通常预留出一大段足够的地址空间给系统 RAM，但是在搭建具体的嵌入式系统时却不一定会实现 CPU 预留的全部 RAM 地址空间，嵌入式系统往往只把 CPU 预留的全部 RAM 地址空间中的一部分映射到 RAM 单元上，而让剩下的那部分预留 RAM 地址空间处于未使用状态

- 内存映射的描述

- typedef struct memory\_area\_struct {  
    u32 start;  
    u32 size;  
    int used;  
} memory\_area\_t;

- 这段 RAM 地址空间中的连续地址范围可以处于两种状态之一：
      - » (1)used=1, 则说明这段连续的地址范围已被实现, 也即真正地被映射到 RAM 单元上。
      - » (2)used=0, 则说明这段连续的地址范围并未被系统所实现, 而是处于未使用状态。

- 基于上述 memory\_area\_t 数据结构, 整个 CPU 预留的 RAM 地址空间可以用一个 memory\_area\_t 类型的数组来表示, 如下所示:

- memory\_area\_t memory\_map[NUM\_MEM\_AREAS] = {  
    [0 ... (NUM\_MEM\_AREAS - 1)] = {  
        .start = 0,  
        .size = 0,  
        .used = 0  
    },  
};

## • 内存映射的检测

- 下面我们给出一个可用来检测整个 RAM 地址空间内存映射情况的简单而有效的算法：

```
for(i = 0; i < NUM_MEM_AREAS; i++)
    memory_map[i].used = 0;

for(addr = MEM_START; addr < MEM_END; addr += PAGE_SIZE)
    * (u32 *)addr = 0;
for(i = 0, addr = MEM_START; addr < MEM_END; addr += PAGE_SIZE) {

    调用3.1.2节中的算法test_mempage();
    if ( current memory page isnot a valid ram page) {

        if(memory_map[i].used )
            i++;
        continue;
    }

    if(* (u32 *)addr != 0) {

        if ( memory_map[i].used )
            i++;
        continue;
    }

    if (memory_map[i].used == 0) {
        memory_map[i].start = addr;
        memory_map[i].size = PAGE_SIZE;
        memory_map[i].used = 1;
    } else {
        memory_map[i].size += PAGE_SIZE;
    }
}
```

- 在用上述算法检测完系统的内存映射情况后，**Boot Loader** 也可以将内存映射的详细信息打印到串口。



## – 任务3：将内核映像和根文件系统映像从Flash读到RAM (SDRAM)

- 规划内存占用的布局

- 这里包括两个方面：

- » 内核映像所占用的内存范围：MEM\_START+0x8000开始、大约1MB大小的内存范围内（嵌入式Linux的内核一般都不超过1MB）
    - » 根文件系统所占用的内存范围：MEM\_START+0x0010,0000开始、大约1MB大小的内存范围内（如果用Ramdisk作为根文件系统映像，则其解压后的大小一般是1MB）

- 从Flash上拷贝

- 由于像ARM这样的嵌入式CPU通常都是在统一的内存地址空间中寻址Flash等固态存储设备的，因此从Flash上读取数据与从RAM单元中读取数据并没有什么不同。用一个简单的循环就可以完成从Flash设备上拷贝映像的工作：

```
while(count) {  
    *dest++ = *src++;  
    count -= 4;  
};
```

## – 任务4：为内核设置启动参数（Boot parameters）

- **Linux 2.4.x** 以后的内核都期望以标记列表（tagged list）的形式来传递启动参数，启动参数标记列表以标记 **ATAG\_CORE** 开始，以标记 **ATAG\_NONE** 结束
- 在嵌入式 Linux 系统中，通常需要由 **Boot Loader** 设置的常见启动参数有：**ATAG\_CORE**、**ATAG\_MEM**、**ATAG\_CMDLINE**、**ATAG\_RAMDISK**、**ATAG\_INITRD**、**ATAG\_NONE**

## – 任务5：调用内核

- **Boot Loader** 调用 **Linux** 内核的方法是直接跳转到内核的第一条指令处，也即直接跳转到 **MEM\_START+0x8000** 地址处
- 如果用 **C** 语言，可以像下列示例代码这样来调用内核：

```
void (*theKernel)(int zero, int arch, u32 params_addr) = (void (*)(int, int, u32))KERNEL_RAM_BASE;  
.....  
theKernel(0, ARCH_NUMBER, (u32) kernel_params_start);
```
- 注意，**theKernel()**函数调用应该永远不返回的，如果这个调用返回，则说明出错

## • 6.2.3 关于串口终端

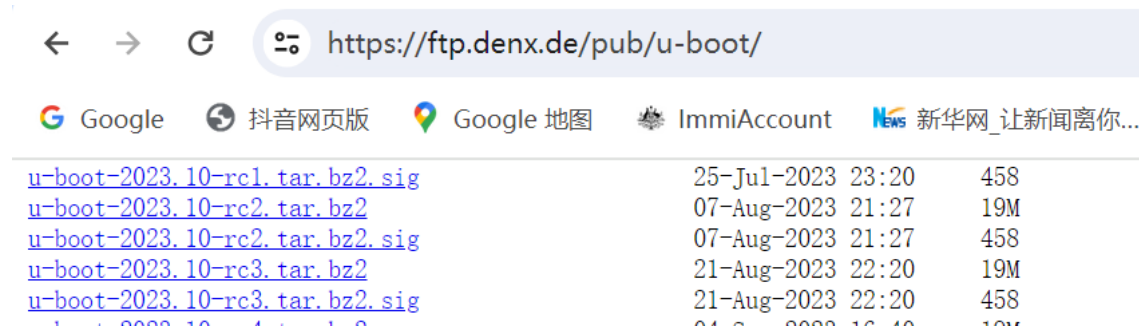
- 经常会碰到**串口终端显示乱码**或**根本没有显示**的问题，造成这个问题主要有两种原因：
  - ① **Boot Loader** 对串口的初始化设置不正确。
  - ② 运行在 **host** 端（主机端，**PC**机）的终端仿真程序（**Xshell**程序）对串口的设置不正确，这包括：波特率、奇偶校验、数据位和停止位等方面的设置。
- 此外，有时也会碰到这样的问题，那就是：在 **Boot Loader** 的运行过程中我们可以正确地向串口终端输出信息，但当 **Boot Loader** 启动内核后却**无法看到内核的启动输出信息**。对这一问题的原因可以从以下几个方面来考虑：
  - ① 首先请确认你的内核在编译时配置了对串口终端的支持，并配置了正确的串口驱动程序。
  - ② 你的 **Boot Loader** 对串口的初始化设置可能会和内核对串口的初始化设置不一致。此外，对于诸如 **s3c44b0x** 这样的 **CPU**，**CPU** 时钟频率的设置也会影响串口，因此如果 **Boot Loader** 和内核对其 **CPU** 时钟频率的设置不一致，也会使串口终端无法正确显示信息。
  - ③ 最后，还要确认 **Boot Loader** 所用的内核基地址必须和内核映像编译时所用的运行基地址一致，尤其是对于 **uClinux** 而言。假设你的内核映像编译时用的基地址是 **0xc0008000**，但你的 **Boot Loader** 却将它加载到 **0xc0010000** 处去执行，那么内核映像当然不能正确地执行了。

## 6.3 U-Boot简介

### • 6.3.1 认识U-Boot

- **U-Boot**，全称 **Universal Boot Loader**（通用的引导程序），是遵循 **GPL**条款的开放源码项目。U-Boot的作用是**系统引导**。U-Boot从 **FADSROM**、**8xxROM**、**PPCBOOT**逐步发展演化而来。其源码目录、编译形式与**Linux**内核很相似，事实上，不少**U-Boot**源码就是根据相应的**Linux**内核源程序进行简化而形成的，尤其是一些设备的驱动程序，这从**U-Boot**源码的注释中能体现这一点。

- <http://ftp.denx.de/pub/u-boot/>（下载地址）



# U-Boot 2017.09

al Disk (C:) > FS3399M4 > u-boot

名称	修改日期	类型	大小
.git	2024/10/7 16:25	文件夹	
.vscode	2024/10/7 16:26	文件夹	
api	2024/10/7 16:26	文件夹	
arch	2024/10/7 16:25	文件夹	
board	2024/10/7 16:26	文件夹	
cmd	2024/10/7 16:25	文件夹	
common	2024/10/7 16:26	文件夹	
configs	2024/10/7 16:25	文件夹	
disk	2024/10/7 16:26	文件夹	
doc	2024/10/7 16:25	文件夹	
Documentation	2024/10/7 16:26	文件夹	
drivers	2024/10/7 16:25	文件夹	
dts	2024/10/7 16:25	文件夹	
env	2024/10/7 16:25	文件夹	
examples	2024/10/7 16:26	文件夹	
fs	2024/10/7 16:25	文件夹	
include	2024/10/7 16:26	文件夹	
lib	2024/10/7 16:25	文件夹	
Licenses	2024/10/7 16:25	文件夹	
net	2024/10/7 16:25	文件夹	
post	2024/10/7 16:26	文件夹	
rkbin	2024/10/7 16:25	文件夹	
scripts	2024/10/7 16:25	文件夹	
spl	2024/10/7 16:25	文件夹	
test	2024/10/7 16:25	文件夹	
tools	2024/10/7 16:26	文件夹	
tpl	2024/10/7 16:25	文件夹	
.checkpatch.conf	2018/11/26 16:15	CONF 文件	1 KB
.config	2023/4/18 14:37	CONFIG 文件	34 KB
.config.old	2023/4/3 17:31	OLD 文件	34 KB
.gitignore	2018/11/26 16:15	GITIGNORE 文件	1 KB
.mailmap	2018/11/26 16:15	MAILMAP 文件	2 KB
.travis.yml	2018/11/26 16:15	YML 文件	12 KB
.u-boot.bin.cmd	2023/4/18 14:38	Windows 命令脚本	1 KB
.u-boot.cmd	2023/4/18 14:37	Windows 命令脚本	2 KB
.u-boot.img.cmd	2023/4/18 14:38	Windows 命令脚本	1 KB
.u-boot.lds.cmd	2023/4/3 17:31	Windows 命令脚本	8 KB
.u-boot.srec.cmd	2023/4/18 14:38	Windows 命令脚本	1 KB
.u-boot.sym.cmd	2023/4/18 14:38	Windows 命令脚本	1 KB

## • 6.3.2 U-Boot特点

- ① 开放源码
- ② 支持多种嵌入式操作系统内核，如Linux、NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS、Android
- ③ 支持多个处理器系列，如PowerPC、ARM、x86、MIPS
- ④ 较高的可靠性和稳定性
- ⑤ 高度灵活的功能设置，适合U-Boot调试、操作系统不同引导要求、产品发布等
- ⑥ 丰富的设备驱动源码，如串口、以太网、SDRAM、FLASH、LCD、NVRAM、EEPROM、RTC、键盘等
- ⑦ 较为丰富的开发调试文档与强大的网络技术支持

## • 6.3.3 U-Boot代码结构分析

### – U-Boot 2017.09的目标结构:

- ① **api:** 存放u-boot提供的接口函数
- ② **arch:** 与体系结构相关的代码
- ③ **board:** 根据不同开发板所定制的代码
- ④ **cmd:** 命令
- ⑤ **common:** 通用的代码, 涵盖各个方面, 已对命令行的处理为主
- ⑥ **configs:** 配置文件
- ⑦ **disk:** 磁盘分区相关代码
- ⑧ **doc:** 文档, **readme**
- ⑨ **drivers:** 驱动相关代码, 每种类型的设备驱动占用一个子目录
- ⑩ **dts:** 设备树文件
- ⑪ **examples:** 示例程序
- ⑫ **fs:** 文件系统, 支持嵌入式开发板常见的文件系统
- ⑬ **include:** 头文件, 以通用的头文件为主
- ⑭ **lib:** 通用库文件 (14个)
- ⑮ **Licenses:** 许可证
- ⑯ **net:** 网络相关的代码, 小型的协议栈
- ⑰ **post:** 上电自检程序
- ⑱ **scripts:** 脚本文件
- ⑲ **test:** 测试文件
- ⑳ **tools:** 辅助功能程序, 用于制作u-boot镜像等



– **ARM926 EJ-S系列处理器（ARM 9系列）的U-boot两阶段代码：**

- **第一阶段（用汇编语言编写）：** **start.S**
  - 位于/u-boot/arch/arm/cpu/arm926ejs/start.S
- **第二阶段（用C语言编写）：** **board.c**
  - 位于/u-boot/arch/arm/mach-rockchip/board.c

# start.S

start.S - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
.globl    reset

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

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

    bl      _main

    /*-----*/

.globl    c_runtime_cpu_setup
c_runtime_cpu_setup:
    bx      lr

    /*
     *****
     *
     * CPU_init_critical registers
     *
     * setup important registers
     * setup memory timing
     *
     *****
     */
#ifdef CONFIG_SKIP_LOWLEVEL_INIT
cpu_init_crit:
    /*
     * flush D cache before disabling it
     */
    mov     r0, #0
flush_dcach:
    mrc     p15, 0, r15, c7, c10, 3
    bne     flush_dcach
```

# board.c

board.c - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
static int rockchip_set_ethaddr(void)
{
#ifdef CONFIG_ROCKCHIP_VENDOR_PARTITION
    int ret;
    u8 ethaddr[ARP_HLEN];
    char buf[ARP_HLEN_ASCII + 1];

    ret = vendor_storage_read(VENDOR_LAN_MAC_ID, ethaddr, sizeof(ethaddr));
    if (ret > 0 && is_valid_ethaddr(ethaddr)) {
        sprintf(buf, "%pM", ethaddr);
        env_set("ethaddr", buf);
    }
#endif
    return 0;
}

static int rockchip_set_serialno(void)
{
    char serialno_str[VENDOR_SN_MAX];
    int ret = 0, i;
    u8 cpuid[CPUID_LEN] = {0};
    u8 low[CPUID_LEN / 2], high[CPUID_LEN / 2];
    u64 serialno;

    /* Read serial number from vendor storage part */
    memset(serialno_str, 0, VENDOR_SN_MAX);
#ifdef CONFIG_ROCKCHIP_VENDOR_PARTITION
    ret = vendor_storage_read(VENDOR_SN_ID, serialno_str, (VENDOR_SN_MAX-1));
    if (ret > 0) {
        env_set("serial#", serialno_str);
    } else {
#endif
#ifdef CONFIG_ROCKCHIP_EFUSE
        struct udevice *dev;

        /* retrieve the device */
        ret = uclass_get_device_by_driver(UCLASS_MISC,
                                           DM_GET_DRIVER(rockchip_efuse), &dev);

        if (ret) {
            printf("%s: could not find efuse device\n", __func__);
            return ret;
        }

        /* read the cpu_id range from the efuses */
        ret = misc_read(dev, CPUID_OFF, &cpuid, sizeof(cpuid));
        if (ret) {
            printf("%s: reading cpuid from the efuses failed\n", __func__);
            return ret;
        }
    }
}
```

## 6.4 vivi简介

- 6.4.1 认识vivi

- vivi是韩国mizi公司设计的一款主要针对S3C2410平台的Boot Loader，其特点是体积小，功能强大，运行效率高和使用方便。vivi代码虽然比较小巧，但麻雀虽小，五脏俱全，用来学习bootloader还是不错的。











## • 6.4.2 vivi代码导读

– vivi的代码目录包括：

- ① **arch** 存放一些平台相关的代码文件
- ② **CVS** 存放CVS工具相关的文件（**CVS: Concurrent Version System**，版本管理工具）
- ③ **Documentation** 存放一些使用vivi的帮助文档
- ④ **drivers** 存放vivi相关的驱动代码
- ⑤ **include** 存放所有vivi源码的头文件
- ⑥ **init** 存放vivi初始化代码
- ⑦ **lib** 存放vivi实现的库函数文件
- ⑧ **scripts** 存放vivi脚本配置文件
- ⑨ **test** 存放一些测试代码文件
- ⑩ **util** 存放一些NAND Flash烧写image相关的工具实现代码

• **Makefile** 用来告诉make怎样编译和连接成一个程序

cal Disk (C:) > FS3399M4 > vivi >

<input type="checkbox"/> 名称	修改日期	类型
 arch	2024/10/7 20:28	文件夹
 CVS	2024/10/7 20:28	文件夹
 Documentation	2024/10/7 20:28	文件夹
 drivers	2024/10/7 20:28	文件夹
 include	2024/10/7 20:28	文件夹
 init	2024/10/7 20:28	文件夹
 lib	2024/10/7 20:28	文件夹
 scripts	2024/10/7 20:28	文件夹
 test	2024/10/7 20:28	文件夹
 util	2024/10/7 20:28	文件夹
 .config	2005/5/16 11:29	CONFIG 文件
 .cvsignore	2005/5/16 11:29	CVSIGNORE 文件
 COPYING	2005/5/16 11:29	文件
 Makefile	2005/5/16 11:29	文件
 Makefile.newSDK	2005/5/16 11:29	NEWSDK 文件
 Rules.make	2005/5/16 11:29	MAKE 文件
 up-netarm3000-2004-5-30	2005/5/16 11:29	文件

## – vivi的两阶段代码

- 第一阶段（用汇编语言编写）：**head.S**
  - /vivi/arch/s3c2410/head.S
- 第二阶段（用C语言编写）：**main.c**
  - /vivi/init/main.c

# head.S

```
head.S x
@
@ Start VIVI head
@
Reset:
    @ disable watch dog timer
    mov     r1, #0x53000000
    mov     r2, #0x0
    str     r2, [r1]

#ifdef CONFIG_S3C2410_MPORT3
    mov     r1, #0x56000000
    mov     r2, #0x00000005
    str     r2, [r1, #0x70]
    mov     r2, #0x00000001
    str     r2, [r1, #0x78]
    mov     r2, #0x00000001
    str     r2, [r1, #0x74]
#endif

    @ disable all interrupts
    mov     r1, #INT_CTL_BASE
    mov     r2, #0xffffffff
    str     r2, [r1, #oINTMSK]
    ldr     r2, =0x7ff
    str     r2, [r1, #oINTSUBMSK]

    @ initialise system clocks
    mov     r1, #CLK_CTL_BASE
    mvn     r2, #0xff000000
    str     r2, [r1, #oLOCKTIME]
```



# main.c

```
head.S x main.c x
#include "machine.h"
#include "mmu.h"
#include "heap.h"
#include "serial.h"
#include "printk.h"
#include "command.h"
#include "priv_data.h"
#include "getcmd.h"
#include "vivi_string.h"
#include "mtd/mtd.h"
#include "processor.h"
#include <reset_handle.h>
#include <types.h>

extern char *vivi_banner;

void
vivi_shell(void)
{
#ifdef CONFIG_SERIAL_TERM
    serial_term();
#else
#error there is no terminal.
#endif
}

void run_autoboot(void)
{
    while (1) {
        exec_string("boot");
        printk("Failed 'boot' command. reentering vivi shell\n");
        /* if default boot fails, drop into the shell */
    }
}
```

# 小结

- **Boot Loader的工作机制:**
  - 两个阶段:
    - 阶段1（汇编语言编写）
    - 阶段2（C语言编写）
  - 两种模式:
    - 下载模式（=>）
    - 启动加载模式
- **两种常用的Boot Loader:**
  - **U-Boot**
  - **vivi**

# 进一步探索

- 阅读**U-Boot**和**vivi**的源代码，分析具体函数的机制和功能。
- 试着修改某个**Boot Loader**，并进行移植。

**Thanks**