**实验一. Linux 内核移植与编译实验**

1. 实验目的

 了解 Linux 内核相关知识与内核结构

 了解 Linux 内核在 ARM 设备上移植的基本步骤和方法

 掌握 Linux 内核裁剪与定制的基本方法

2. 实验内容

 分析 Linux 内核的基本结构，了解 Linux 内核在 ARM 设备上移植的一些基本步骤及常识。

 学习Linux 内核裁剪定制的基本配置方法，利用IMX6 设备配套Linux 内核进行自定义功能(如helloworld 显示)的添加。并重新编译内核源码，生成内核压缩文件 zImage ，下载到 IMX6 设备中测试。

3. 实验环境

 硬件：IMX6 教学平台，PC 机酷睿 i3 以上, 硬盘 120G 以上, 内存 2G 以上

 软件：Vmware Workstation +Yocto 项目

4. 实验原理

**4.1** **Linux** **内核简介**

一个[计算机系统](http://baike.sogou.com/lemma/ShowInnerLink.htm?lemmaId=1695130&ss_c=ssc.citiao.link)是一个硬件和软件组成，它们互相依赖，不可分割。计算机的硬件，含有外围设备、 处理器、 内存、硬盘和其他的电子设备组成计算机的发动机。但是没有软件来操作和控制它， 自身是不能 工作的。完成这个控制工作的软件就称为操作系统，在 Linux 的术语中被称为“内核” ，也可以称为“核心”。 Linux 内核的主要模块（或组件）分以下几个部分：[存储管理](http://baike.sogou.com/lemma/ShowInnerLink.htm?lemmaId=276803&ss_c=ssc.citiao.link)、CPU 和进程管理、文件系统、设备管理和驱 动、网络通信，以及系统的初始化（引导）、[系统调用](http://baike.sogou.com/lemma/ShowInnerLink.htm?lemmaId=350141&ss_c=ssc.citiao.link)等。



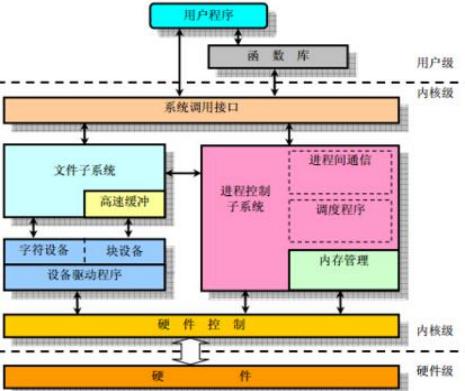


图 3.1.1 Linux 内核

最上面是用户（或应用程序）空间，这是用户应用程序执行的地方。用户空间之下是内核空间，Linux 内 核正是位于这里。

Linux 内核本身并不是操作系统，它是一个完整操作系统的组成部分。Red Hat、Novell、Debian 和 Gentoo 等 Linux 发行商都采用 Linux 内核，然后加入更多的工具、库和应用程序来构建一个完整的操作系统。

u **Linux 内核的发展**

自 1991 年 11 月由芬兰的Linus Ttorvalds 推出Linux 0. 1.0 版内核至今，Linux 内核已经升级到 Linux4.8. 11 （写本文档时，www.kernel.org 发布的最新版 Linux 内核）。其发展速度是如此的迅猛，是目前市场上唯一 可以挑战 Windows 的操作系统。

Linux 内核在其发展过程中得到分布于全世界的广大 OpenSource 项目追随者的大力支持。尤其是一些 曾经参与 Unix 开发的人员，他们把应用于 Unix 上的许多应用程序移植到 Linux 上来，使得 Linux 的功能得 到巨大的扩展。

本节用的内核版本是 Linux4.9.88 。在 Linux 的版本号中，第一个数为主版本号。第二个为次版本号。 第三个为修订号。次版本号为偶数表明是稳定发行版本，奇数则是在开发中的版本。

随着其功能不断加强，灵活多样的实现加上其可定制的特性以及开放源码的优势， Linux 在各个领域 的应用正变得越来越广泛。 目前 Linux 的应用正有舍去中间奔两头的趋势，即在 PC 机上 Linux 要真正取代 Windows ，或许还有很长的路要走，但在服务器市场上它已经牢牢站稳脚跟。而随着嵌入式领域的兴起更是 为 Linux 的长足发展提供了无限广阔的空间。目前专门针对嵌入式设备的 Linux 改版就有好几种。包括针对 无 MMU 的 uClinx 和针对有 MMU 的标准 LINUX 在各个硬件体系结构的移植版本，使用户应用程序的可靠 性得以提高，降低了用户的开发难度。

**4.2** **Linux** **内核目录结构**

IMX6 平台运行的 Linux 内核版本为 linux4.9.88 ，其源码目录结构如图所示：



图 3. 1.2 目录结构

\* **arch** 与体系结构相关的代码放在这里，例如：arm 、x86 、powerpc。

\* **block** 部分块设备驱动程序。

\* **crypto** 内核本身所用的加密 API ，实现了常用的加密和散列[算法](http://lib.csdn.net/base/datastructure)，还有一些压缩和CRC 校验算法。

\* **Documentation** 存放着内核的所有开发文档，其中的文件会随版本的演变发生变化，通过阅读这里的 文件是获得内核最新的开发资料的最好的地方。

\* **drivers** 此目录包括所有的驱动程序，下面又建立了多个目录，分别存放各个分类的驱动程序源代码， 是内核中最大的源代码存放处，大约占整个内核的一多半。其中我们经常会用到的目录有：

**drivers/char** 字符设备是 drivers 目录中最为常用，最为重要的目录，其中包含了大量的驱动程序代码。 通用的tty 层在这里实现，console.c 定义了 linux 终端类型，vt.c 中定义了虚拟控制台；lp.c 中实现了一个通 用的并口打印机的驱动，并保持设备无关性；led 控制驱动；kerboard.c 实现高级键盘处理，它导出

handle\_scancode 函数，以便于其他与平台相关的键盘驱动使用。我们的大部分实验也是放在这个目录下。

**driver/block** 其中存放所有的块设备驱动程序，保存了一些设备无关的代码。如 rd.c 实现了 RAM 磁盘， nbd.c 实现了网络块设备，loop.c 实现了回环块设备。

**drives/ide** 专门存放针对 IDE 设备的驱动。

**drivers/scsi** 存放 SCSI 设备的驱动程序，当前的 cd 刻录机、扫描仪、U 盘等设备都依赖这个 SCSI 的 通用设备。

**drivers/net** 存放网络接口适配器的驱动程序，还包括一些线路规程的实现，但不实现实际的通信协议， 这部分在顶层目录的 net 目录中实现。

**drivers/video** 这里保存了所有的帧缓冲区视频设备的驱动程序，整个目录实现了一个单独的字符设备驱 动。/dev/fb 设备的入口点在 fbmem.c 文件中，该文件注册主设备号并维护一个此设备的清单，其中记录了 哪一个帧缓冲区设备负责哪个次设备号。

**Drivers/media** 这里存放的代码主要是针对无线电和视频输入设备，比如目前流行的usb 摄像头。



\* **fs** 此目录下包括了大量的文件系统的源代码，其中在嵌入式开发中要使用的包括：devfs、cramfs、ext4、 jffs2 、romfs 、yaffs 、vfat 、nfs 、proc 等。

文件系统是 Linux 中非常重要的子系统，这里实现了许多重要的系统调用，比如 exec.c 文件中实现了 execve 系统调用；用于文件访问的系统调用在 open.c 、read\_write.c 等文件中定义，select.c 实现了 select 和 poll 系统调用，pipe.c 和 fifo.c 实现了管道和命名管道， mkdir 、rmdir 、rename 、link 、symlink 、mknod 等 系统调用在namei.c 中实现。

文件系统的挂装和卸载和用于临时根文件系统的initrd 在 super.c 中实现。Devices.c 中实现了字符设备 和块设备驱动程序的注册函数；file.c 、inode.c 实现了管理文件和索引节点内部数据结构的组织。Ioctl.c 实 现 ioctl 系统调用。

\* **include** 这里是内核的所有头文件存放的地方，其中的linux 目录是头文件最多的地方，也是驱动程序 经常要包含的目录。

\* **init**linux 的 main.c 程序，通过这个比较简单的程序，我们可以理解 LINUX 的启动流程。

\* **ipc** 进程间通信的原语实现，包括信号量、共享内存。

\* **kernel** 这个目录下存放的是除网络、文件系统、内存管理之外的所有其他基础设施，其中至少包括进 程调度 sched.c ，进程建立 fork.c ，定时器的管理 timer.c ，中断处理，信号处理等。

\* **lib** 包括一些通用支持函数，类似于标准 C 的库函数。其中包括了最重要的 vsprintf 函数的实现，它是 printk 和 sprintf 函数的核心。还有将字符串转换为长整形数的 simple\_atol 函数。

\* **mm** 这个目录包含实现内存管理的代码，包括所有与内存管理相关的数据结构，其中我们在驱动中需 要使用的 kmalloc 和 kfree 函数在 slab.c 中实现，mmap 定义在 mmap.c 中的 do\_mmap\_pgoff 函数。将文件 映射到内存的实现在 filemap.c 中，mprotect 在 mprotect.c，remap 在 remap.c 中实现；vmscan.c 中实现了 kswapd 内核线程，它用于释放未使用和老化的页面到交换空间，这个文件对系统的性能起着关键的影响。

\* **net** 这个目录包含了套接字抽象和网络协议的实现，每一种协议都建立了一个目录，但是其中的 core、 bridge 、ethernet、sunrpc 、khttpd 不是网络协议，我们使用最多的是 ipv4 、ipv6 、802 、ipx 等。其中我们经 常会用到的目录有：

**net/core** 目录中实现了通用的网络功能：设备处理、防火墙、组播、别名等。

**net/ethernet** 和 **net/bridge** 实现特定的底层功能： 以太网相关的辅助函数以及网桥功能。 **net/sunrpc** 中提供了支持 NFS 服务器的函数。

\* **samples** 一些内核编程的范例。

\* **script** 这个目录存放许多脚本，主要用于配置内核。

\* **sound** 音频设备的驱动目录。

\* **tools** 工具文件夹。

\* **usr**cpio 命令的实现。 **主要文件如下：**

\* **.config** 内核里默认的隐藏配置文件，需要用 ls -a 查看。

\* **arch/arm/configs/**目录下所有的文件为不同平台的不同配置信息，可在主目录里 make \*\_defconfig 即 可，比如我们需要 imx6 默认配置，直接 make imx\_v7\_defconfig 或者拷贝到.config。

\* **Makefile** 编译时需要的主文件，各子目录也有类似的文件。

\* **README** 内核使用说明。

\***arch/arm/boot/dts/imx6qdl-sabresd.dtsi** 设备树文件，是对 imx6dl 所有功能的配置，比如 gpio 、i2c 总 线，串口等等。

\***arch/arm/boot/zImage** 内核编译完生成的文件，主要用于系统烧写。

\* **arch/arm/boot/dts/imx6dl-sabresd.dtb** 内核设备树编译完生成的文件，主要用于系统烧写。

**4.3 Linux 内核配置及裁剪**

Linux 内核的裁剪与编译看上去是个挺简单的过程。只是对配置菜单的简单选择。但是内核配置菜单本 身结构庞大， 内容复杂。具体如何选择却难住了不少人。因此熟悉与了解该菜单的各项具体含义就显得比 较重要。我们现在就对其作一些必要介绍：

Linux 内核的编译菜单有好几个版本，主要有：

1）make config：进入命令行，可以一行一行的配置，这个方式不友好所以我们不具体介绍。

2）make xxx\_defconfig：直接用该文件的配置。

3）make menuconfig：我们熟悉的 menuconfig 菜单，相信很多人对此都不陌生。 在选择相应的配置时，有三种选择方式，它们分别代表的含义如下：

\* －将该功能编译进内核 空格－不将该功能编译进内核

M －将该功能编译成可以在需要时动态插入到内核中的模块

这里使用的是make menuconfig ，所以需要使用空格键进行选取。在每一个选项前都有一个括号（有的 是中括号，有的是尖括号，还有圆括号）。中括号里要么是空，要么是"\*" ，尖括号里可以是空、"\*"和"M"。 圆括号的内容是要你在所提供的几个选项中选择一项。

(注：其中有不少选项是目标板开发人员加的，对于陌生选项， 自己不知道该选什么时建议使用默认)

u **内核中的** **Kconfig 和** **Makefile**

在内核的源码树目录下一般都会有两个重要文件：Kconfig 和Makefile 。分布在各目录下的 Kconfig 构 成了一个分布式的内核配置数据库，每个 Kconfig 分别描述了所属目录下源文件相关的内核配置菜单。在内 核配置make menuconfig 时，从 Kconfig 中读出配置菜单，用户配置完后保存到.config(在内核源码顶层目录 下生成)中。在内核编译时，主 Makefile 调用这个.config(隐藏文件) ，就知道了用户对内核的配置情况。

Kconfig 的作用就是对应着内核的配置菜单。假如要想添加新的驱动到内核的源码中，可以通过修改 Kconfig 来增加对我们驱动的配置菜单，这样就有途径选择我们的驱动。



如果想使这个驱动是否被编译进内核里，还要修改该驱动所在目录下的 Makefile 文件。该 Makefile 文 件定义和组织该目录下驱动源码在内核目录树中的编译规则。这样在make 编译内核的时候，内核源码目录 顶层 Makefile 文件会递归的连接相应子目录下的 Makefile 文件，进而对驱动程序进行编译。

如上所述，添加用户驱动程序(内核程序)到内核源码目录树中，一般需要修改 Konfig 及 Makefile 两个 文件。这要求我们用户要对上述连个文件的特殊语法有一定了解，如下：

**Kconfig 语法：**

每行都是以关键字开始，并可以接多个参数，最常见的关键字就是 config。 语法：

config symbol

options

<!-- [if !supportLineBreakNewLine]-->

<!-- [endif]-->

symbol 就是新的菜单项，options 是在这个新的菜单项下的属性和选项。 其中 options 部分有：

1 、类型定义：

每个 config 菜单项都要有类型定义，bool：布尔类型， tristate 三态：内建、模块、移除， string：字符串， hex：十六进制， integer：整型

例如：

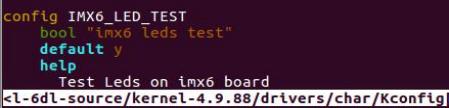


图 3.1.3 Kconfig 文件

bool 类型的只能选中或不选中，tristate 类型的菜单项多了编译成内核模块的选项，假如选择编译成内 核模块，则会在.config 中生成一个 CONFIG\_HELLO\_MODULE=m 的配置，假如选择内建，就是直接编译 成内核影响，就会在.config 中生成一个 CONFIG\_HELLO\_MODULE=y 的配置.

2 、依赖型定义 depends on 或 requires

指此菜单的出现是否依赖于另一个定义。



图 3.1.4 Kconfig 的

3 、帮助性定义

只是增加帮助用关键字 help 或---help---。

更多详细的 Kconfig 语法可参考内核目录/kbuild/kconfig-language.txt 文档。

**内核的** **Makefile 语法**

内核的 Makefile 分为 5 个组成部分：

Makefile 最顶层的 Makefile

.config 内核的当前配置文档，编译时成为顶层Makefile 的一部分

arch/$(ARCH)/Makefile 和体系结构相关的 Makefile

Makefile.\* 一些 Makefile 的通用规则

Kbuild 和Makefile 编译时根据上层 Makefile 传下来的宏定义和其他编译规则，将源代码编译成模 块或编入内核。

顶层的 Makefile 文档读取 .config 文档的内容，并总体上负责 build 内核和模块。Arch Makefile 则提供 补充体系结构相关的信息。（其中.config 的内容是在make menuconfig 的时候，通过 Kconfig 文档配置的结 果）

更多详细的 Makefile 语法可参考内核目录 Documentation/kbuild/makefiles.txt 文档。



/ {

node1 {

a-string-list-property = "first string", "second string";

first-child-property;

second-child-property = <1>;

};

child-node2 {

};

};

node2 {

an-empty-property;

a-cell-property = <1 2 3 4>; /\* each number (cell) is a uint32 \*/

child-node1 {

};

};

a-byte-data-property = [0x01 0x23 0x34 0x56];

a-string-property = "Hello, world";

a-string-property = "A string";

child-node1 {

};

1 个 root 根结点"/"；

root 结点下面含一系列子结点，本例中为"node1" 和 "node2" ；结点"node1"下又含有一系列子结点，本 例中为"child-node1" 和 "child-node2"；各结点都有一系列属性。这些属性可能为空，如" an-empty-property"； 可能为字符串，如"a-string-property" ；可能为字符串数组，如"a-string-list-property" ；可能为 Cells（由 u32 整数组成），如"second-child-property" ，可能为二进制数，如"a-byte-data-property"。

本平台的设备树是 arch/arm/boot/dts/imx6dl-sabresd.dts 和 arch/arm/boot/dts/imx6qdl-sabresd.dtsi ，我们来 分析源文件主要的结构。

root@uptech:~/fsl-6dl-source/kernel-4.9.88$ vim arch/arm/boot/dts/imx6qdl-sabresd.dtsi



图 3.1.5 设备树文件

/{//根节点

battery: max8903@0 { //battery 为子节点；max8903 为设备名称；@0 为设备地址；

compatible = "fsl,max8903-charger";//fsl 为平台名称，max8903-charger 为驱动名称，会匹配驱动文件。 pinctrl-names = "default";//引脚配置状态"default"和"sleep" ，对应 pinctrl-0

dok\_input = <&gpio2 24 1>;//gpio2\_24 高电平有效

//del cjf uok input = <&gpio1 27 1>;

chg\_input = <&gpio3 23 1>; flt\_input = <&gpio5 2 1>;

fsl,dcm\_always\_high;//bool 类型，只有 1 和 0 fsl,dc\_valid;

fsl,usb\_valid;

status = "okay";//okay 开启此设备，disabled 是关闭 };

......//省略部分代码

gpio-leds-test {//led 设备

compatible = "fsl,gpio-leds-test";//匹配驱动 pinctrl-names = "default";

pinctrl-0 = <&pinctrl\_gpio\_leds\_test>;//led 引脚配置 gpio0 = <&gpio1 13 0>; //led1 节点名称 gpio1 = <&gpio4 20 0>; //led2

gpio2 = <&gpio1 11 0>; //led3

};

};//根节点结束

pinctrl\_gpio\_leds\_test: gpio\_ledsgrp {//对应 leds 的 pinctrl-0 具体配置 fsl,pins = <

MX6QDL\_PAD\_DI0\_PIN4\_\_GPIO4\_IO20 0x80000000//将引脚配置成 GPIO 模式

MX6QDL\_PAD\_SD2\_DAT2 GPIO1\_IO13 0x80000000

MX6QDL PAD SD2 CMD GPIO1 IO11 0x80000000 >;

}; ......//省略部分代码 &weim {

pinctrl-names = "default";

pinctrl-0 = <&pinctrl\_weim\_nor\_ 1>;//引脚配置列表

#address-cells = <1>;//地址 #size-cells = <1>;//大小

status = "okay";//使能

nor@0,0 { compatible = "cfi-flash";

reg = <0 0x02000000>;//EIM 寄存器开始地是 0 大小为 0x2000000 //#address-cells = <1>;

//#size-cells = <1>; bank-width = <2>;

fsl,weim-cs-timing = <0x00620081 0x00000001 0x1c022000 0x0000c000 0>//对寄存的配置 };

文件中用到的标签如下： "/"：是 root 根节点

"**battery: max8903@0**" ：为根下子节点，遵循的组织形式为：<name>[@<unit-address>] ，<>中的内容是 必选项，[]中的则为可选项。name 是一个 ASCII 字符串，用于描述结点对应的设备类型，。

"**compatible**" ：定义的系统名称，它的组织形式为：<manufacturer>,<model> 。内核或者 uboot 依靠这个 属性找到相对应 driver ，若"compatible"出现多个属性，按序匹配 driver。

"**pinctrl-names**" ：有 "default", "sleep"两种状态，对应于 pinctrl-0()里配置。

"**reg**" ：是寄存器，格式是"<address1,length 1 ，address2,length2 … …>" ，作为平台内存资源，表明了设备 使用的一个地址范围。address 为 1 个或多个 32 位的整型（即 cell），而 length 则为 cell 的列表或者为空（若 #size-cells = 0）。address 和 length 字段是可变长的，父结点的#address-cells 和#size-cells 分别决定了子结 点的 reg 属性的 address 和 length 字段的长度。

"**#address-cells** "：是 address 的单位(32bit)； "**#size-cells**" ：是 length 的单位(32bit)；

"**aliase**" ：是别名，必须节点全称，可以通过地址引用获取； "**chosen**" ：是板级启动参数；

"**cpus**" ：是 SOC 的 CPU 信息，可以改变运行频率或者开关 CPU； "**memory**" ：是板级内存的信息。

"**interrupt-parent**" ：中断引脚所在的 gpio 组

"**interrupts**" ：是中断控制器，这里是<pin 脚触发方式> ，作为平台中断资源；触发方式有 4 种： 1 = low-to-high edge triggered



2 = high-to-low edge triggered 4 = active high level-sensitive

8 = active low level-sensitive "**interrupt-0**"：中断引脚配置。

"**device\_type**" ：设备类型，寻找节点可以依据这个属性；

"**status**" ：是开关节点设备的状态，取值"okay"或者"ok"表示使能，"disabled"表示失能。

"**reset-gpios = <&gpio1 15 1>**" ：这格式是什么意思呢？&gpio1 15 引用了 gpio1 节点，故此处含义为 gpio1\_ 15 这个引脚；最后一个参数 1 则代表低电平有效，0 则为高电平有效。

**引脚定义：**

设备树中引脚不同的功能配置不一样的参数。

|  |  |  |
| --- | --- | --- |
| MX6QDL\_PAD\_DI0\_PIN4 GPIO4\_IO20 | 0x80000000 | //GPIO 模式 |
| MX6QDL\_PAD\_CSI0\_DAT10 UART1\_TX\_DATA | 0x1b0b 1 | //串口模式 |
| MX6QDL\_PAD\_CSI0\_DAT8 I2C1\_SDA | 0x4001b8b 1 | //i2c 模式 |
| MX6QDL\_PAD\_EIM\_D16 EIM\_DATA16 | 0xb0b 1 | //EIM 总线模式 |

MX6QDL\_PAD\_DI0\_PIN4 GPIO4\_IO20：在 arch/arm/boot/dts/imx6dl-pinfunc.h 定义



图 3.1.6 引脚功能配置

将管脚的宏定义配置展开即： 0x0ac 0x3c0 0x000 0x5 0x0 0x80000000 ，前五个变量在该头文件的开始 有定义，代表引脚的地址，最后一个变量代表对寄存器的配置。

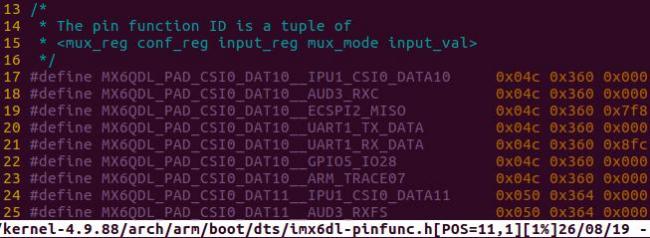


图 3.1.7 引脚功能配置 可以在数据手册里确定，分为 MUX 寄存器和控制寄存器。

打开光盘\硬件\数据手册\Cortex-A 系列核心板\IMX6\ IMX6SDLRM.pdf 的 2078 页为 MUX 寄存器。

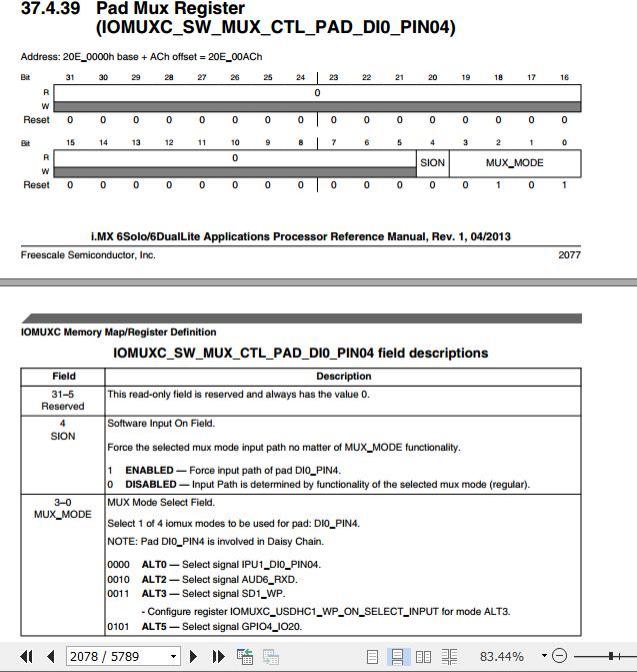


图 3.1.8 数据手册中引脚信息

2283 页为控制寄存器。



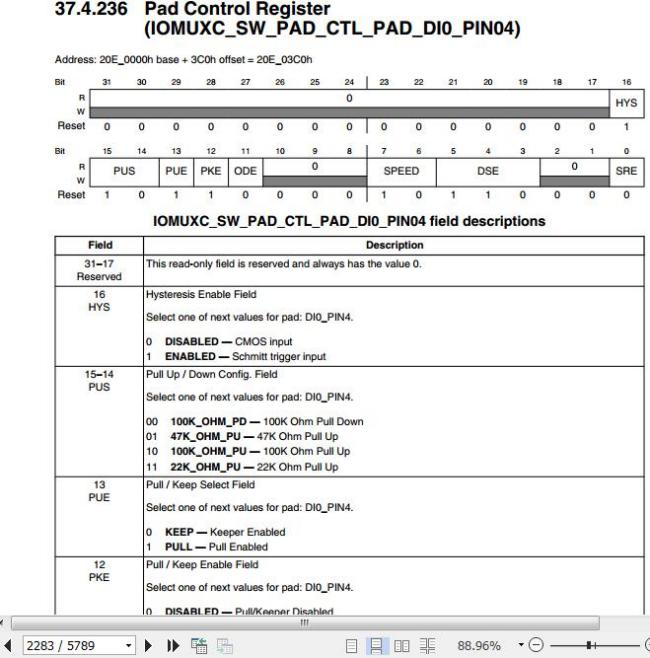


图 3.1.9 数据手册中引脚信息

其中对应关系如下:：

mux\_reg=0x0AC，MUX 寄存器的偏移地址，2078 页，Address: 20E\_0000h base + ACh offset = 20E\_00Ach。 conf\_reg=0x3C0，控制寄存器的偏移地址，2283 页，Address: 20E\_0000h base + 3C0h offset = 20E\_03C0h input\_reg=000 ，MUX 模式，bit0~3 ，范围 0~7。

mux\_mode=0x5 ，SELECT\_INPUT 寄存器偏移地址(16 进制) ，2078 页，0101ATL5 为 GPIO4\_IO20. input\_val=0 ，Daisy Chain 模式, bit0~ 1,范围 0~3。

config=0x80000000, 2283 页，直接取默认值的话是 config=0x1b0b0 ，可根据自己硬件需求修改。

input\_reg 补充：查找 IOMUXC 章节以 SELECT\_INPUT 结尾的部分，这没有，input\_reg 为 0 。如果有 例如 IOMUXC\_UART2\_UART\_RX\_DATA\_SELECT\_INPUT ，即 uart 的 rx 管脚配置，如下图，所以 RX 的 sel\_input\_ofs=0x904 ，这里选择对应的值“110 SD4\_DATA4\_ALT2 — Selecting ALT2 mode ofpad SD4\_DAT4 for UART2\_RX\_DATA“所以 RX（MX6QDL\_PAD\_SD4\_DAT4\_\_UART2\_RX\_DATA）的 input\_reg=0x6。

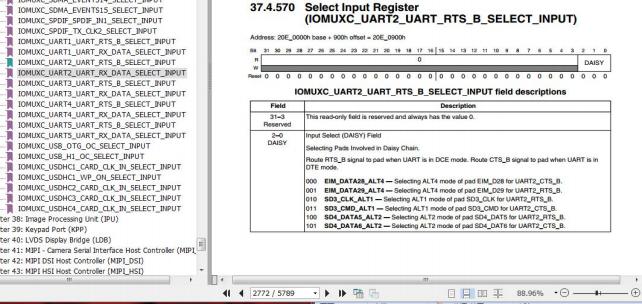


图 3.1.10 数据手册中引脚信息

config 补充：当我们查找相应的 pad 值时是 0x1b0b0，这明显不合常理，在上图中显示高 15 位全部置 0，

取值也没什么用，那么为什么设置为 0x80000000 呢？查找设备树文件的源码， drivers/pinctrl/freescale/pinctrl-imx.c 中：



图 3.1.11 pinctrl-imx.c 文件

可以看出来确实如注释（/\* no pin config need \*/）所述，表示该管脚的配置 config（pad\_ctrl）无效，或 者说不需要。

同理 0x40000000 表示设置了 SION。设备树里有 0x4001b8b 1 表示什么意思呢，从上一个注释（/\* The bits in CONFIG cell defined in binding doc\*/）可以找到方向，即取 binding doc 中找，所以打开

Documentation/devicetree/bindings/pinctrl 目录下的 fsl,imx-pinctrl.txt 文件，里面有关于 SION 的介绍，如下：

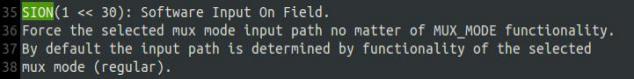


图 3.1.12 fsl,imx-pinctrl.txt 文件

再从芯片的参考手册中查阅可知，SION 就相当于一个标志为（第 30 位），去掉这一位后 config=0x1b8b 1， 这个值就是从 pad\_ctrl 一节找到的。



u **DTC (device tree compiler)**

将.dts 编译为.dtb 的工具。DTC 的源代码位于内核的 scripts/dtc 目录，在 Linux 内核使能了 Device Tree 的情况下，编译内核的时候主机工具 dtc 会被编译出来，对应 scripts/dtc/Makefile 中的“hostprogs-y := dtc ” 这一 hostprogs 编译 target 。在 Linux 内核的 arch/arm/boot/dts/Makefile 中，描述了当某种 SoC 被选中后，哪 些.dtb 文件会被编译出来。

u **.dtb(Device Tree Blob)**

.dtb 是.dts 被 DTC 编译后的二进制格式的 Device Tree 描述，可由 Linux 内核解析。通常在我们为电路 板制作 NAND 、SD 启动 image 时，会为.dtb 文件单独留下一个很小的区域以存放之，之后 bootloader 在引 导 kernel 的过程中，会先读取该.dtb 到内存。

u **常用** **OF API 函数**

在 Linux 的 BSP 和驱动代码中，还经常会使用到 Linux 中一组 Device Tree 的 API,这些 API 通常被冠以 of\_前缀，它们的实现代码位于内核的 drivers/of 目录。这些常用的 API 包括：

**int of\_device is compatible(const struct device\_node \*device,const char \*compat)；**判断设备结点的

compatible 属性是否包含 compat 指定的字符串。当一个驱动支持 2 个或多个设备的时候，这些不同.dts 文 件中设备的 compatible 属性都会进入驱动 OF 匹配表。因此驱动可以透过 Bootloader 传递给内核的 Device Tree 中的真正结点的 compatible 属性以确定究竟是哪一种设备，从而根据不同的设备类型进行不同的处理。 如 drivers/pinctrl/pinctrl-sirf.c 即兼容于"sirf,prima2-pinctrl"，又兼容于"sirf,prima2-pinctrl"，在驱动中就有相应 分支处理：

**struct device\_node \*of\_find\_compatible\_node(struct device\_node \*from, const char \*type, const char**

**\*compatible)；**根据 compatible 属性，获得设备结点。遍历 Device Tree 中所有的设备结点，看看哪个结点的 类型、compatible 属性与本函数的输入参数匹配，大多数情况下，from 、type 为 NULL。

**int of\_property\_read\_u8\_array(const struct device\_node \*np, const char \*propname, u8 \*out\_values, size\_t sz);**

**int of\_property\_read\_u16\_array(const struct device\_node \*np, const char \*propname, u16 \*out\_values, size\_t sz); int of\_property\_read\_u32\_array(const struct device\_node \*np, const char \*propname, u32 \*out\_values, size\_t sz); int of\_property\_read\_u64(const struct device\_node \*np, const char \*propname, u64 \*out\_value);**

读取设备结点 np 的属性名为 propname ，类型为 8 、16 、32 、64 位整型数组的属性。对于 32 位处理器 来讲，最常用的是 of\_property\_read\_u32\_array()。

**int of\_property\_read\_string(struct device\_node \*np, const char\*propname, const char \*\*out\_string);**

**int of\_property\_read\_string\_index(struct device\_node \*np, const char\*propname, int index, const char \*\*output);**

前者读取字符串属性，后者读取字符串数组属性中的第 index 个字符串。

**static inline bool of\_property\_read\_bool(const struct device\_node \*np, const char \*propname);**如果设 备结点 np 含有 propname 属性，则返回 true ，否则返回 false 。一般用于检查空属性是否存在。

**void iomem \*of\_iomap(struct device\_node \*node, int index);**通过设备结点直接进行设备内存区间的 ioremap()，index 是内存段的索引。若设备结点的 reg 属性有多段，可通过 index 标示要 ioremap 的是哪一段，

只有 1 段的情况，index 为 0 。采用 Device Tree 后，大量的设备驱动通过 of\_iomap()进行映射，而不再通过 传统的 ioremap。

**unsigned int irq of parse\_and\_map(struct device\_node \*dev, int index);** 透过 Device Tree 或者设备的 中断号，实际上是从.dts 中的 interrupts 属性解析出中断号。

还有一些 OF API ，这里不一一列举，详细可参考 include/linux/of.h 头文件。

**4.2** **内核启动流程分析**

zImage： 由压缩后的内核 piggy.o ，连接上一段初始化及解压功能的代码（head.o misc.o）组成的。

vmlinux：elf 格式内核，或者是加上了 piggy.o/misc.o./head.o 后形成的自启动的内核。 zImage/bzImage：用 zip 和 big zip 压缩的 vmlinux 内核

内核整个流程文件如下：

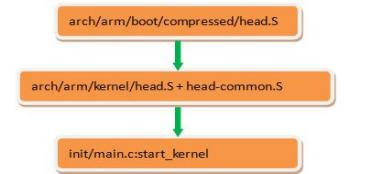


图 3.1.13 内核流程文件

zImage 生成的过程如下：

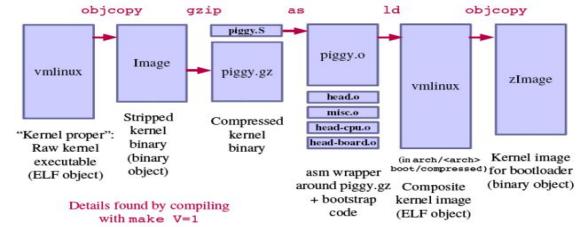


图 3.1.14 zImage 生成流程

1.根据内核配置文件，将 linux 中的子系统编译成一个个build-in.o。

2.将所有的 build-in.o 链接生成 vmlinx ，使用的链接脚本为 arch/arm/kernel/vmlinux.lds。

3.将生成的 vmlinux 通过 arm-fsl-linux-gnueabi-objcopy 去掉符号表，调试信息等，生成 arch/arm/boot/Image。

4.将生成的 arch/arm/boot/Image 通过 gzip 压缩成 arch/arm/boot/compressed/piggy.gzip

5.根据链接脚本 arch/boot/compressed/vmlinux.lds,将 piggy.gzip 和 arch/boot/compressed 里的自解压程序 链接成 arch/arm/boot/compressed/vmlinux。

6.将生成的 vmlinux 通过 arm-fsl-linux-gnueabi-objcopy 去掉符号表，调试信息等，生成 arch/arm/boot/zImage。

内核从什么地方开始运行？要看 lds 文件。zImage 的生成经历了两次大的链接过程：

一次是顶层根目录下 vmlinux 生成，由 arch/arm/boot/vmlinux.lds（这个 lds 文件是由 arch/arm/kernel/v mlinux.lds.S 生成的）决定；另一次是 arch/arm/boot/compressed/vmlinux 的生成，是由 arch/arm/boot/compre ssed/vmlinux.lds（这个 lds 文件是由 arch/arm/boot/compressed/vmlinux.lds.in 生成的）决定。

**入口** **start：**

zImage 的入口点应该由 arch/arm/boot/compressed/vmlinux.lds 决定，截取/vmlinux.lds 部分内容，如下所 示。从中可以看出入口点为‘\_start ’。

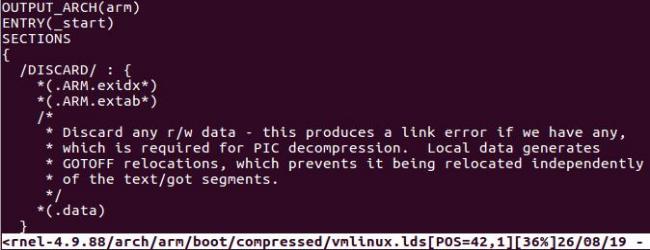


图 3.1.15 vmlinux.lds 文件 查看 arch/arm/boot/compressed/Makefile 里有如下两句：



图 3.1.16 Makefile 文件

所以，zImage 的入口点便是 arch/arm/boot/compressed/head.S 里的 start ，下面来进行详细分析：

root@uptech:~/fsl-6dl-source/kernel-4.9.88$ vim arch/arm/boot/compressed/head.S



图 3.1.17 head.S 文件

使用.type 标号来指明 start 的符号类型是函数类型，然后重复执行.rept 到.endr 之间的指令 7 次，这里一 共执行了 7 次 mov r0, r0 指令，共占用了 4\*7 = 28 个字节，这是用来存放 ARM 的异常向量表的。向前跳转 到标号为 1 处执行：

这里将 CPU 的工作模式保存到 r9 寄存器中，将 uboot通过 r1 传入的机器码保存到 r7 寄存器中，将启 动参数 tags 的地址保存到 r8 寄存器中。

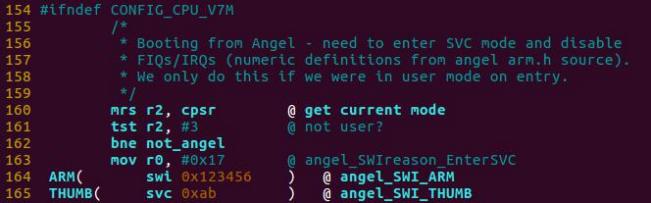


图 3.1.18 head.S 文件

这里将 CPU 的工作模式保存到 r2 寄存器中，然后判断是否是 SVC 模式，如果是 USER 模式就会通过 swi 指令产生软中断异常的方式来自动进入 SVC 模式。由于在uboot 中已经将 CPU 的模式设置为 SVC 模式 了，所以就直接跳到 not\_angel 符号处执行。



图 3.1.19 head.S 文件



safe\_svcmode\_maskall 是一个宏，定义在 arch/arm/include/asm/assembler.h 中，主要是强制将 CPU 的工 作模式切换到 SVC 模式，并且关闭 IRQ 和 FIQ 中断。然后将 r9 中保存的原始 CPU 配置保存到 SPSR 中。



图 3.1.20 head.S 文件

内核配置项 AUTO\_ZRELDDR 表示自动计算内核解压地址，这里没有选择这个配置项，所以保存到 r4 中的内核解压地址就是 zreladdr ，这个参数在 linux/arch/arm/boot/compressed/Makefile 中。

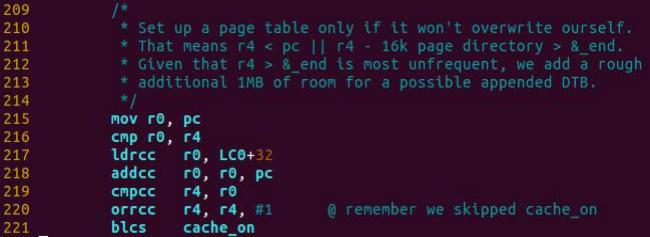


图 3.1.21 head.S 文件

这里将比较当前 PC 地址和内核解压地址，只有在不会自覆盖的情况下才会创建一个页表，如果当前运 行地址 PC <解压地址 r4 ，则读取 LC0+32 地址处的内容加载到 r0 中，LC0 表的定义为：



图 3.1.22 head.S 文件 然后跳转到cache\_on 处执行缓存初始化和MMU 初始化。

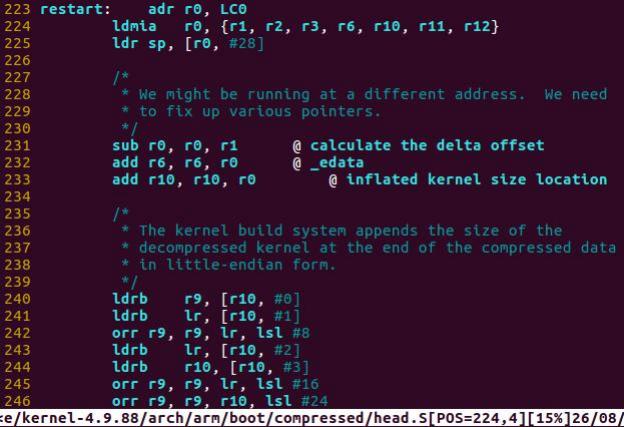


图 3.1.23 head.S 文件

通过前面 LC0 地址表的内容可见，这里 r0 中的内容就是编译时决定的 LC0 的实际运行地址（特别注意 不是链接地址），然后调用 ldmia 命令依次将 LC0 地址表处定义的各个地址加载到 r1、r2、r3、r6、r10、r11、 r12 和 SP 寄存器中去。执行之后各个寄存器中保存内容的意义如下：

（1） r0：LC0 标签处的运行地址

（2） r1：LC0 标签处的链接地址

（3） r2：\_\_bss\_start 处的链接地址

（4） r3：\_ednd 处的链接地址（即程序结束位置）

（5） r6：\_edata 处的链接地址（即数据段结束位置）

（6） r10：压缩后内核数据大小位置

（7） r11：GOT 表的启示链接地址

（8） r12：GOT 表的结束链接地址

（9） sp：栈空间结束地址

在获取了 LC0 的链接地址和运行地址后，就可以通过计算这两者之间的差值来判断当前运行的地址是 否就是编译时的链接地址。

将运行地址和链接地址的偏移保存到 r0 寄存器中，然后更新 r6 和 r10 中的地址，将其转换为实际的运 行地址。

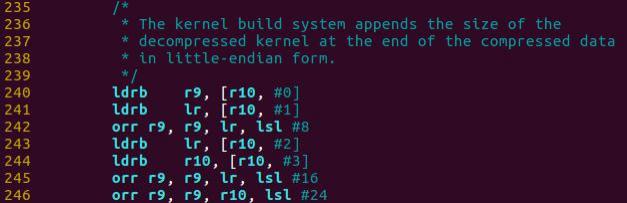


图 3.1.24head.S 文件

注释中说明了，内核编译系统在压缩内核时会在末尾处以小端模式附上未压缩的内核大小，这部分代 码的作用就是将该值计算出来并保存到 r9 寄存器中去。

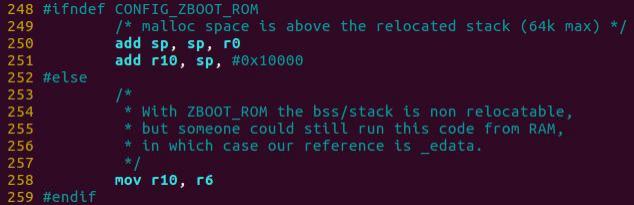


图 3.1.25head.S 文件

这里将镜像的结束地址保存到 r10 中去，这里并没有定义 ZBOOT\_ROM（如果定义了 ZBOOT\_ROM 则 bss 和 stack 是非可重定位的），这里将 r10 设置为 sp 结束地址上 64kb 处（这 64kB 空间是用来作为堆空间 的）。

接下来内核如果配置为支持设备树（DTB）会做一些特别的工作：



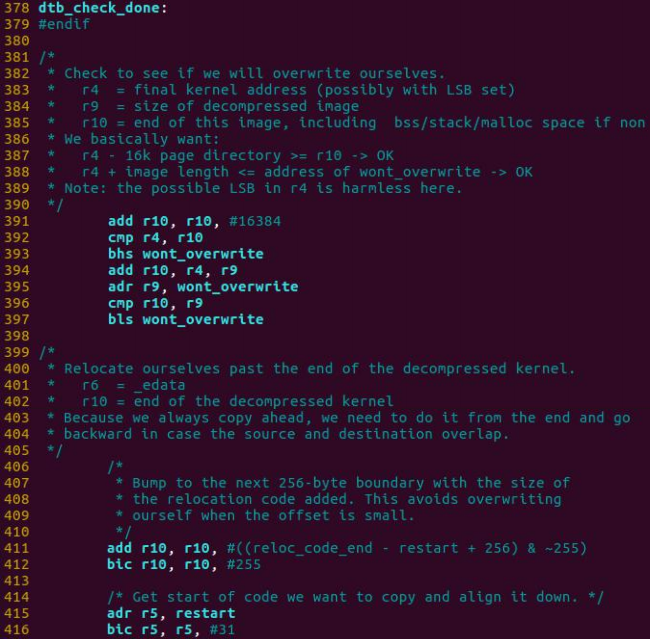


图 3.1.26 head.S 文件

这部分代码用来分析当前代码是否会和最后的解压部分重叠，如果有重叠则需要执行代码搬移。首先 比较内核解压地址 r4-16Kb（这里是 0x00004000，包括 16KB 的内核页表存放位置）和 r10，如果 r4 – 16kB >= r10 ，则无需搬移，否则继续计算解压后的内核末尾地址是否在当前运行地址之前，如果是则同样无需搬移， 不然的话就需要进行搬移了。

再往下代码就是将镜像搬移到解压的内核地址之后，首先将解压后的内核结束地址进行扩展，扩展大 小为代码段的大小（reloc\_code\_end 定义在 head.s 的最后）保存到 r10 中，即搬运目的起始地址，然后 r5 保 存了 restart 的起始地址，并进行对齐，即搬运的原起始地址。



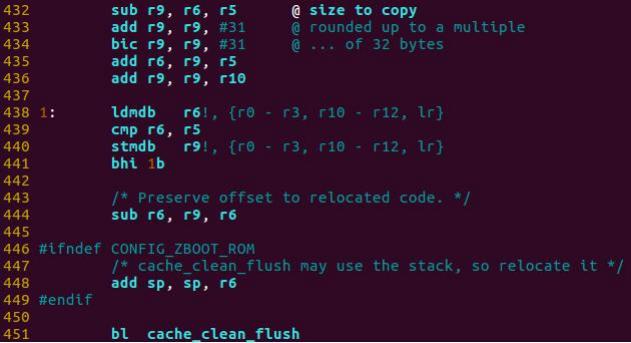


图 3.1.27head.S 文件

首先计算出搬运的大小保存到 r9 中，搬运的原结束地址到 r6 中，搬运的结束地址到 r9 中。注意这里 只搬运代码段和数据段，并不包含bss 、栈和堆空间。接下来开始执行代码搬移，从后往前搬移，一直到 r6 == r5 结束，然后 r6 中保存了搬移前后的偏移，并重定向栈指针（cache\_clean\_flush 可能会使用到栈）。

首先调用 cache\_clean\_flush 清楚缓存，然后将 PC 的值设置为搬运后 restart 的新地址，然后重新从 restart 开始执行。这次由于进行了代码搬移，所以会在检查自覆盖时进入 wont\_overwrite 处执行。

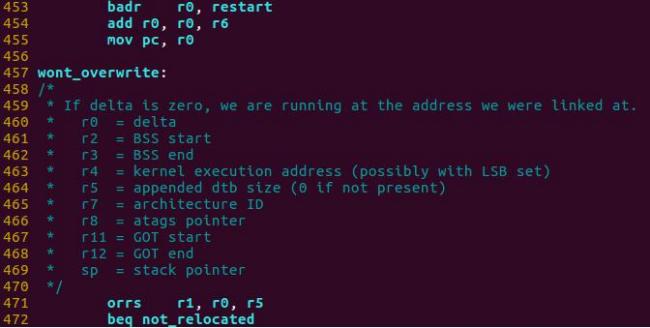


图 3.1.28head.S 文件

这里的注释列出了现有所有寄存器值得含义，如果 r0 为 0 则说明当前运行的地址就是链接地址，无需 进行重定位，跳转到 not\_relocated 执行，但是这里运行的地址已经被移动到内核解压地址之后，显然不会 是链接地址 0x00000168（反汇编代码中得到），所以这里需要重新修改 GOT 表中的变量地址来实现重定位。

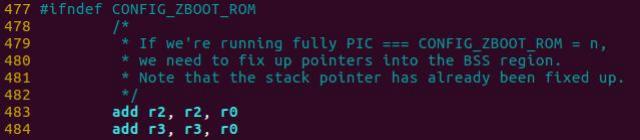


图 3.1.29head.S 文件

这里更新 GOT 表的运行起始地址到 r11 和结束地址到 r12 中去，然后同样更新 BSS 段的运行地址（需 要修正 BSS 段的指针）。接下来开始执行重定位：

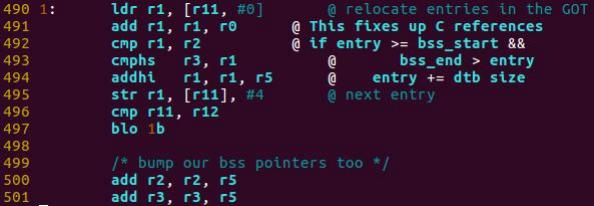


图 3.1.30head.S 文件

通过 r1 获取 GOT 表中的一项，然后对这一项的地址进行修正，循环执行直到遍历完 GOT 表。

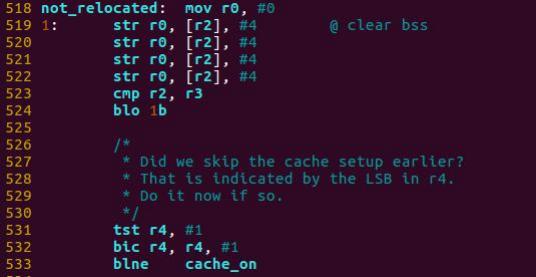


图 3.1.31head.S 文件



在重定位完成后，继续执行 not\_relocated 部分代码，这里循环清零 BSS 段。检测 r4 中的最低位，没有 的话执行 cache\_on。



图 3.1.32head.S 文件

到此为止，[C 语言](http://lib.csdn.net/base/c)的执行环境已经准备就绪，设置一些指针就可以开始解压内核了（这里的内核解压 部分是使用C 代码写的）。

这里 r0~r3 的 4 个寄存器是 decompress\_kernel()函数传参用的，r0 传入内核解压后的目的地址，r1 传入 堆空间的起始地址，r2 传入堆空间的结束地址，r3 传入机器码，然后就开始调用 decompress\_clean\_flush() 函数执行内核解压操作，具体函数实现在同目录下 misc.c 中。



图 3.1.33head.S 文件

解压完成后就刷新缓存，然后将缓存（包括 MMU 关闭），这里之所以要打开缓存和 MMU 是为了加 速内核解压。然后将机器码和内启动参数 atags 恢复到 r1 和 r2 寄存器中，为跳转到解压后的内核代码做准 备。

**b enter\_kernel**



图 3.1.34head.S 文件

在 r4 地址开始的就是内核执行文件，也就是 arch/arm/boot/Image 文件。接着给 r0 、r1 、r2 、r3 寄存器 分布赋值，这三个值就是 Uboot传进来的三个值，最后跳转到 r4 处。

**入口** **stext：**

r4 就是跳转到 arch/arm/boot/Image 的入口，或者说跳转到 vmlinux 的入口。 先看 vmlinux 的链接脚本，在 arch/arm/kernel/vmlinux.lds：

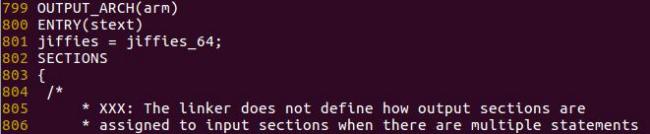


图 3.1.35vmlinux.lds 文件

此处的 TEXT\_OFFSET 表示内核起始地址相对于 RAM 地址的偏移值，定义在 arch/arm/Makefile 中，值 为 0x00008000：

stext 的定义在 arch/arm/kernel/head.S 中，注释都比较详细：

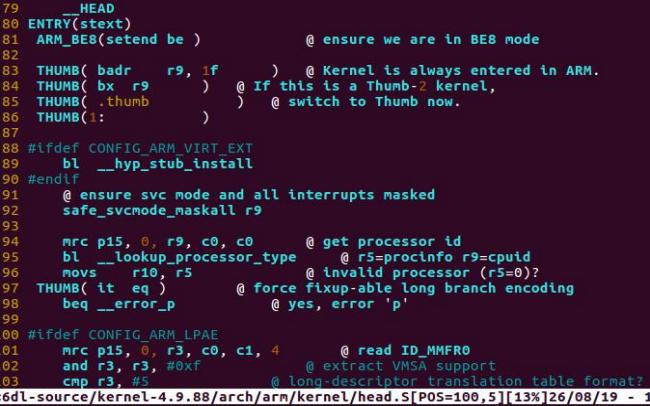


图 3.1.36 vmlinux.lds 文件

**分析重点部分**

**lookup\_processor\_type：**

在进入 SVC 模式，关闭中断之后，接着：



图 3.1.37 vmlinux.lds 文件

将处理器ID 保存在r9 寄存器，然后调用\_\_lookup\_processor\_type。该函数定义在同目录的head-common.S 中，主要的作用就是根据 r9 寄存器里面保存的处理器类型判断该处理器是否被支持。



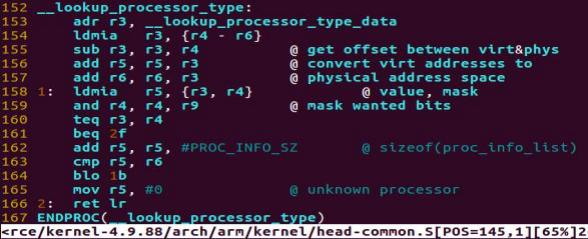


图 3.1.38 head-common.S 文件

将\_\_lookup\_processor\_type\_data 处的数据加载到 r3～r6 寄存器内，计算运行地址和链接地址的偏移量。 跳转到 1。

接下来就是遍历支持的CPU 类型数据结构表中，判断是否支持本 CPU 类型：此时 r5 寄存器中保存的 是.proc.info.init 的起始地址，r6 寄存器保存的是.proc.info.init 段的终止地址。

首先将 r5 地址处的两个数据加载到 r3,r4 寄存器内，根据定义我们可以知道此时加载的是 cpu\_val 和 cpu\_mask ，然后将 r9 寄存器与 cpu\_mask 与操作再与 cpu\_val 比较。

如果相等则说明支持该 CPU ，跳转到 2 ，即返回。不相等则跳转到 add r5, r5, #PROC\_INFO\_SZ @ sizeof(proc\_info\_list) ，跳到下一个 struct proc\_info\_list 再对比，遍历所有的 struct proc\_info\_list 结构体。

分析完\_\_lookup\_processor\_type 函数，程序继续往下执行。



图 3.1.39 head.S 文件

计算出 PAGE\_OFFSET 的物理地址，调用\_\_vet\_atags、\_\_fixup\_smp、\_\_create\_page\_tables ，接下来我们 只分析\_\_vet\_atags 和\_\_create\_page\_tables。

**vet\_atags：**

定义在head-common.S 中，主要作用就是检查 UBoot传进来的参数是否合法。即是否为 ATAG\_CORE 开头的内核参数或者以 OF\_DT\_MAGIC 开头的设备树。如果不合法，则将 r2 的寄存器值赋为 0。

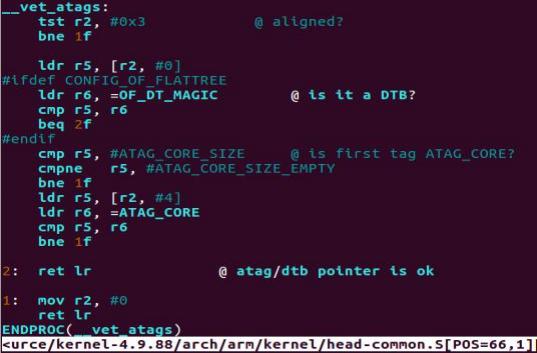


图 3.1.40 head-common.S 文件

**create\_page\_tables**

在开启MMU 之前，要先做好一个页表，作为虚拟地址跟物理地址的映射，要将虚拟地址

0x000000000~0xFFFFFFFF 这 4G 的地址映射到实际的物理地址上面，由于是临时的，只是为了执行内核（因 为内核编译的时候链接地址是 PAGE\_OFFSET+TEXT\_OFFSET），与开启MMU 前执行地址(即内核被

bootload 加载的地方，或者说内核自解压的地方)PHY\_OFFSET（本开发板为 0x10008000）不一致，涉及到 一些变量的存放地址都是虚拟地址，所以要开启MMU 做映射，才能正确执行内核。

定义在 head.S 中,作用如下：



图 3.1.41head.S 文件

1.清零用来保存页表的 16K 数据。

2.对\_\_enable\_mmu 和\_\_enable\_mmu\_end 之间做平等映射，所谓平等映射就是虚拟地址跟物理地址一样。 这段代码是开启 MMU ，在开启MMU 时 PC 的地址为物理地址，开启之后，PC 只是继续读取下一条指令 （即 PC+=4）,但是此时已变成虚拟地址，如果没有做平等映射，此时 PC 的值为虚拟地址，就会跑飞掉， 但是有做平等映射后，虚拟地址跟物理地址一致，所以 PC 会正常执行，直到一个跳转指令的到来。

3.做内核起始地址和终止地址之间的映射。

4.经过这些映射，开启MMU 之后，内核依然可以正常执行。



程序继续往下分析，注释里解释很清楚：

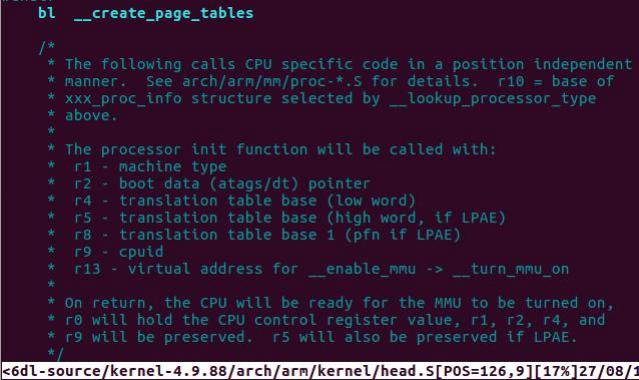


图 3.1.42head.S 文件

1. 将\_\_mmap\_switched 的地址保存在 r13 ，然后保存页表地址到 r8,跳转到 struct proc\_info\_list->\_\_cpu\_flush。

2. struct proc\_info\_list->\_\_cpu\_flush 的定义在 arch/arm/mm/proc-\*\*.S ，对我来说就是 arch/arm/mm/proc-v7.S

3. \_\_enable\_mmu 函数在head.S 中定义，它的作用就是开启 MMU ，最后跳到 r13 地址去执行 \_\_mmap\_switched。

4. \_\_mmap\_switched 的定义在 head-common.S 中：

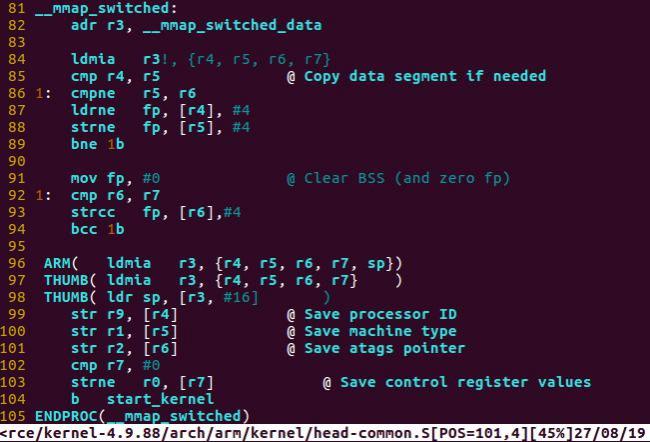


图 3.1.43 head-common.S 文件 清除 BSS 段，跳转到 start\_kernel。

**start\_kenel：**

start\_kernel 定义在 init/main.c 中，就是内核启动的第二阶段（C 语言）。在该函数里面做着大量的初始 化工作，是一个非常重要的函数。主要流程如下：

1 、void lockdep\_init(void) 有些体系结构有自己的 start\_kernel 也会调用 lockdep\_init ，这里只会调用一 次，来初始化 hash 表。这个 hash 表是个全局的锁链表，lock dependency 哈希表。

2、void \_\_init \_\_weak smp\_setup\_processor\_id(void) 指定当前 cpu 的逻辑号（判断是否是 smp 系统，如 果是则从 arm协处理器读取当前 cpuid ，否则为0），这个函数对应于对称多处理器的设置，当系统中只有 一个 cpu 的情况，此函数为空，什么也不做。

3 、void page\_address\_init(void) 当定义了 CONFIG\_HIGHMEM 宏，并且没有定义

WANT\_PAGE\_VIRTUAL 宏时，非空函数。其他情况为空函数。ARM9 不支持高端地址（大于 896M），一 般的嵌入式产品也不会用高端地址，所以，在 ARM 体系结构下，此函数为空！

4、void \_\_init setup\_arch(char \*\*cmdline\_p) 这个setup\_arch()函数是start\_kernel 阶段最重要的一个函数， 每个体系都有自己的 setup\_arch()函数，是体系结构相关的，具体编译哪个体系的 setup\_arch()函数，由顶层 Makefile 中的 ARCH 变量决定。该函数的主要功能有，分析 uboot 传进来的参数(tags) 、初始化内存结构、 创建页表、开启mmu 等。



5 、static void \_\_init setup\_command\_line(char \*command\_line) 保存未改变的 comand\_line 到字符数组 static\_command\_line［］ 中。保存 boot\_command\_line 到字符数组 saved\_command\_line［］ 中。

6 、void \_\_init setup\_per\_cpu\_areas(void) 如果没有定义 CONFIG\_SMP 宏，则这个函数为空函数。如果 定义了 CONFIG\_SMP 宏，则这个 setup\_per\_cpu\_areas()函数给每个 CPU 分配内存，并拷贝.data.percpu 段的 数据。为系统中的每个 CPU 的 per\_cpu 变量申请空间。

7 、void sched\_init(void) 核心进程调度器初始化，调度器的初始化的优先级要高于任何中断的建立， 并且初始化进程 0 ，即 idle 进程，但是并没有设置 idle 进程的 NEED\_RESCHED 标志，所以还会继续完成 内核初始化剩下的事情。

8、void init\_IRQ(void) 初始化 IRQ 中断和终端描述符。初始化系统中支持的最大可能的中断描述结构 struct irqdesc 变量数组 irq\_desc[NR\_IRQS],把每个结构变量 irq\_desc[n]都初始化为预先定义好的坏中断描述 结构变量 bad\_irq\_desc,并初始化该中断的链表表头成员结构变量 pend。

9、void \_\_init vfs\_caches\_init(unsigned long mempages) 初始化虚拟文件系统 VFS，创建一个 rootfs，这 是个虚拟的 rootfs ，即内存文件系统，后面还会指向真实的文件系统。

10 、static noinline void \_\_init\_refok rest\_init(void) 完成了基本的初始化工作后，最后调用了 rest\_init 函 数。这个函数创建了一个入口点是 kernel\_init()函数的内核线程，该线程里面会使用 run\_init\_process()接口来 实现启动用户进程的工作，然后调用 cpu\_idle()函数进入空闲状态。新创建的内核线程是系统的 1 号任务(pid = 1 的进程) ，放入了调度队列中，而原先的初始化代码是系统的0 号任务，是不在调度队列中的。1 号进程 会启动一个内核线程来运行其中的/init 脚本，完成真正根文件系统的挂载。

5. 实验步骤

实验目录：/home/uptech/fsl-6dl-source/kernel-4.9.88

在宿主机端为编写 IMX6 设备 Linux4.9.88 内核编写简单的测试驱动（内核）程序helloworld.c 并修改 内核目录中相关文件，添加对测试驱动程序的支持。

1、使用 vim 编辑器手动编写实验代码 helloworld.c

root@uptech-virtual-machine:/home/uptech#**cd fsl-6dl-source/kernel-**4.9.88**/drivers/char**

root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88#**vim helloworld.c**

helloworld.c 内如如下：

#include <linux/init.h>

#include <linux/module.h>

MODULE\_LICENSE ("Dual BSD/GPL"); //驱动程序入 口函数

static int hello init (void)

\_

{

printk (KERN\_ALERT "##############Hello, world############\n");

return 0;

}

//驱动程序出 口函数

static void hello exit (void)

\_

{

printk (KERN\_ALERT "###############Goodbye, world#########\n");

return 0;

}

module\_init (hello\_init);

module exit (hello\_exit);

有关驱动程序的编写规范，请参考后续驱动实验内容，本实验只在编写简单的驱动(内核)程序并加入到 Linux 内核目录树中。该驱动程序是向终端输出相关程序信息。

2 、进入实验内核源码目录修改 driver/char/目录下的 Kconfig 文件，按照 Kconfig 语法添加 helloworld 程序的菜单支持。

|  |  |
| --- | --- |
| root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/#cd kernel-4.9.88  root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88#vi  drivers/char/Kconfig | |
| 例如：在 Kconfig 文件中最后一行 endmenu 的前面添加如下： | |
| config HELLO\_MODULE | |
| bool "Hello World Test" | |
| help | |
| This is a demo to test kernel experiment On IMX6. | |
| **注意** **config** **HELLO\_MOULDE** **段要与前后段有空格隔开，且** **bool** **等变量要与行开头有** **TAB** **符号位隔开。** **注意Kconfig** **的格式(用户可以拷贝原有内容进行修改)。**  3 、进入实验内核源码目录修改 driver/char/目录下的 Makefile 文件，按照内核中 Makefile 语法添加 helloworld 程序的编译支持 | |
| root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88#vi | |
| drivers/char/Makefile | |
| 在 Makefile 中最后一行添加如下代码。 | |
| obj-$ (CONFIG\_\_HELLO\_\_MODULE) += helloworld .o | |
| 4 、运行 make menuconfig(注意：如运行失败请加权限) 配置内核对 helloworld 程序的支持。 | |
| [root@uptech:~/kernel-4.9.88]$make menuconfig | |
| 进入到 Device Drivers ---> | |
| Character devices | ---> |
| [\*]Hello World | Test |
| 按下空格，选择\*将其静态编译进内核 | |

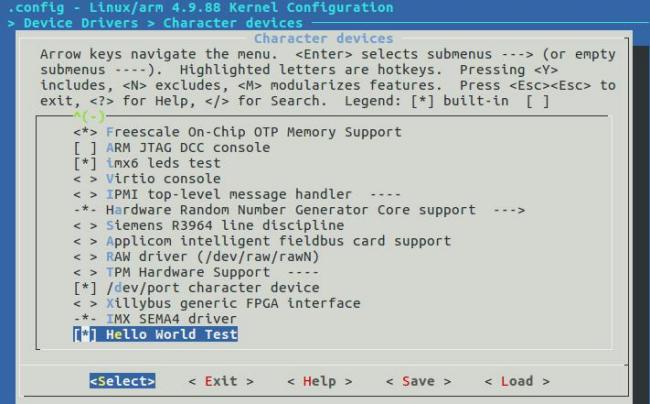


图 3. 1.44 Hello World Test 选项

退出<Exit>保存内核配置

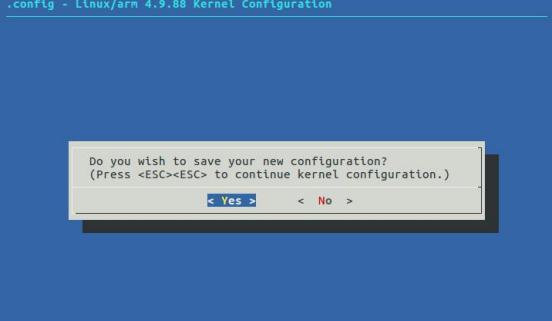


图 3. 1.45 保存配置

5 、重新编译内核

在内核源码的顶层目录下编译内核和设备树文件。

root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88# make

root@uptech-virtual-machine:/home/uptech/fsl-6dl-source/kernel-4.9.88# make

imx6dl-sabresd.dtb

编译完后会有 **arch/arm/boot/zImage** 和 **arch/arm/boot/dts/imx6dl-sabresd.dtb**

将内核 **zImage** 和 **imx6dl-sabresd.dtb**(重名为 zImage-imx6dl-sabresd.dtb)拷贝到光盘 IMX6 烧写工具 \mfgtools\Profiles\Linux\OS Firmware\files 目录下，替换即可，具体烧写步骤请查看配套的系统烧写文档。

6 、开机启动，查看输出信息，会有##############Hello, world############显示字样，说明我们静态

加载成功。

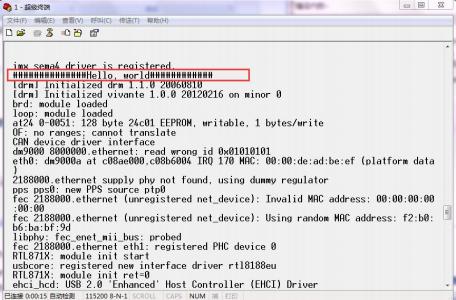


图 3.1.46 开启启动信息