Linux Quick Boot

Dwf 2015

July 3rd, 2015

Tristan Lelong

Senior Embedded Software Engineer

TABLE OF CONTENTS

1. 介绍

2. Linux 启动过程

3. 开机时间优化

4. 如何保持一个完整系统

5. 总结

介绍

作者

- Tristan Lelong
- **Adeneo Embedded** 嵌入式工程师
- 法国
- 嵌入式软件，免费软件和 Linux 内核爱好者。

我们在讨论 FASTBOOT 吗?

这个演讲是关于 QUICK BOOT

- Fastboot 是在 Android 的一种刷机模式
- 把 *Fastboot* 输到谷歌：首 10 页百分之 90 的内容都是关于 Android

注意

Fastboot 不是这次演讲的主题

速度的重要性

- 关键系统
  - 汽车应用在开动后，需要在一个特定短的时间(x100ms) 内响应
  - 监控：当系统出现错误时，以最快的速度重新启动
  - 医疗
  - 长时间待机

速度的重要性

当消费者使用上个世代的电子产品，那些产品可以即开即用。

- 客户产品
    - 电视，相机：打开电源马上使用
    - 手机：深度休眠模式，允许有一个积极的电源管理策略，同时保持响应

速度的重要性

错觉

- 在引导程序中显示启动画面（x100ms）
- 在开启过程中，同时播放开机动画

这些技巧使得使用者有着快速启动的错觉，但这可能不适用于关键系统

设定目标

不同的市场有不同的需求，消费者一般是看不出低于一秒钟的差别

演讲目的

这次演讲的目的:

- 用不同的技考来缩短启动时间
- 如何整合
- 如何优化定制项目
- 尽可能保持完整的 Linux 系统

演讲目的

会

- 专注于标准的 Linux 的系统
- 专注于 ARM 的架构

不会

- 列出所有已知的方法
- 解释如何实现最好的启动时间但牺牲许多功能

直觉

直觉一般是对的

- 较小的空间使用
  - 加载更快
  - 从更快的存储中加载（NOR，MMC10 级）
- 删除不需要的功能
  - 小系统（见上图）
  - 不浪费时间

直觉

但直觉不是 100%对的

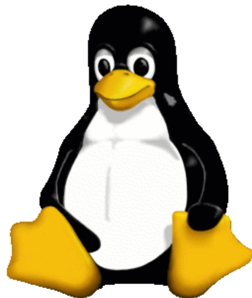
- 更高的频率
  - 更高的内存带宽
  - 可能需要初始化
- 更强大的 CPU 处理能力
  - 关键代码运行得更快
  - 高级的 SOC 通常会有更多的周边设备

Linux 启动过程

概述

嵌入式系统启动步骤

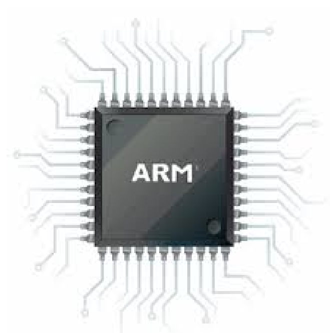
- ROM code
- Secondary Program Loader
- Bootloader
- Kernel
- Userland



ROM CODE

ROM code 是由 CPU 运行的第一个程序。

- 位于 SOC 里的 ROM
- 低层初始化
- 不能修改



SECONDARY PROGRAM LOADER

ROM code 只做基本初始化，一般不会初始化外部 RAM

- 位于永久性内存（NOR，NAND，MMC）
- 低层初始化
- 程式小，可在 SOC 内部 RAM 中运行
- 可以被修改（有些系统不需要 SPL）

BOOTLOADER

一旦外部 RAM 初始化后，就可以初始化其他的

- 位于永久性内存（NOR，NAND，MMC）
- 低层的初始化
- 高层功能
- 可以修改

KERNEL

Bootloader 把 Kernel 加载到内存, 跳进内核运行

- 内核本身可以解压缩自己
- 内核本身可以 relocate
- 内核可以初始化所有的模块
- 内核挂载 root 文件系统
- 内核执行 init 进程
- 可以被修改

USERLAND

当操作系统启动时，开始主要服务和应用程式。

- `init` 读取配置并启动所有服务
- 服务是由 shell 脚本启动的 (`inittab`, `rcS`, `SXX`, `KXX`)
- 传统的 init 进程（sysvinit）是一个串行化的进程
- 可以修改

开机时间优化

配置环境：BUILDSYSTEM

在这之前首先建立一个系统

- 小改动
- 很多重新编译
- 没有无谓的错误
- 有更多的时间用于优化上

配置环境：BUILDSYSTEM

- Buildroot: http://buildroot.uclibc.org
- Yocto: https://www.yoctoproject.org
- Custom scripts: do-it-yourself

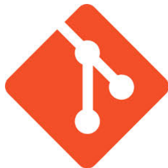
Toolchain

Buildsystem 可以用预先编译好的 toolchain 或在 Buildsystem 中编译

很多重新编译

代码管理是非常重要的

- 追踪修改
- 还原不工作的修改
- 还原不能节费时间的修改



配置环境: 硬件

整个过程都需要分析和测量

- 示波器
- GPIO/ LED
- 串口
- JTAG
- 相机

评估需求

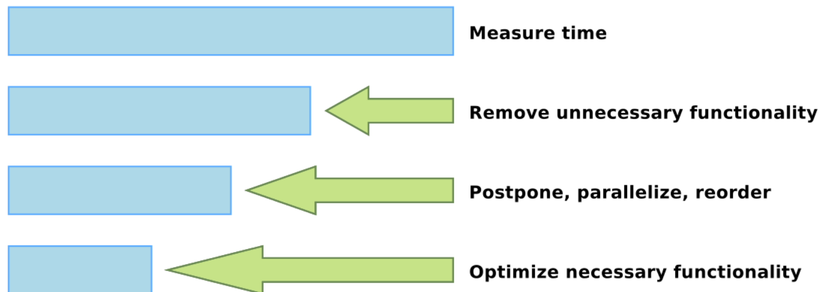
不是所以的产品都有一样的要求

- 在几秒内启动
- 在一秒内给予一些响应
- 在大约一秒内启动 (用户就可直接使用)
- 在一秒内给予一些响应和进行一些基本运作（关键系统）

评估需求

- 所需功能是什么？
- 关键功能是什么？
- 可选功能是什么？

方法



Free Electron

测量和分析

测量使我们在优化上不用浪费时间

- 测量五个开机步骤而得 知该从哪开始
- 记住我们的目标 知道何时要停止
- 决定是否保留或删除一些功能

测量和分析

在不修改的情况下，开机时间是多少，我们拿 Freescale i.MX6Q Nitrogen 运行 Linux 内核 3.10.17 做个比较

- buildroot 生成 image
  - System 大小: 2MB (initramfs)
  - Kernel 大小: 4.6MB
  - 开机时间: 4.466984 秒

- yocto 生成 image
  - System 大小: 4.6MB (ext3)
  - Kernel 大小: 3.6MB
  - 开机时间: 7.095716 秒

测量工具

几个开源工具可用于测量不同的开机步骤时间。

- Printk time (kernel): 在列印 `printk` 讯息前先把时间打印出来
- Grabserial (http://eLinux.org/Grabserial): 把串口的信息读出来，然后在前面加个时间
- Bootgraph (`kernel/scripts`): 用 kernel 的 `dmesg` 去产生一个 kernel startup grap `initcall_debug` + `CONFIG_PRINTK_TIME` + `CONFIG_KALLSYMS`
- Bootchart (http://www.bootchart.org): bootchart 从 `/proc` 中把开机的资讯取出来分析

测量工具

- Gpio: 在需要测量的地方切换 GPIO 输出高低

```
1  /* C code to set GPIO2_2 */
2  *(volatile unsigned long*)0x20a0004 = 0x00000002;
3  *(volatile unsigned long*)0x20a0000 = 0xc000f07f;
4
5  /* ARM Assembly  to clear GPIO2_2 */
6  ldr     r1, =0x20a0000  @ GPIO2 base register
7  ldr     r5, =0x2        @ gpio2_2 as output
8  str     r5, [r1, #4]    @ set gpio2_2 direction
9  ldr     r5, =0xc000f07d @ gpio2_2 cleared
10 str     r5, [r1, #0]    @ clear gpio2_2
```

- 相机: 高速相机记录启动过程

测量工具

Figure: bootgraph

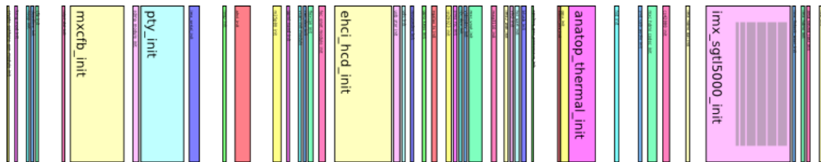


Figure: bootgraph 放大图

测量工具

```
[    2.486193] calling  proc_uptime_init+0x0/0x40 @ 1
[    2.491022] initcall proc_uptime_init+0x0/0x40 returned 0 after 7 usecs
[    2.497672] calling  proc_version_init+0x0/0x40 @ 1
[    2.502575] initcall proc_version_init+0x0/0x40 returned 0 after 5 usecs
[    2.509312] calling  proc_softirqs_init+0x0/0x40 @ 1
[    2.514303] initcall proc_softirqs_init+0x0/0x40 returned 0 after 6 usecs
[    2.521126] calling  proc_kmsg_init+0x0/0x40 @ 1
[    2.525767] initcall proc_kmsg_init+0x0/0x40 returned 0 after 5 usecs
[    2.532241] calling  proc_page_init+0x0/0x60 @ 1
[    2.536886] initcall proc_page_init+0x0/0x60 returned 0 after 8 usecs
[    2.543360] calling  init_devpts_fs+0x0/0x54 @ 1
[    2.548055] initcall init_devpts_fs+0x0/0x54 returned 0 after 44 usecs
[    2.554607] calling  init_cramfs_fs+0x0/0x38 @ 1
[    2.559298] initcall init_cramfs_fs+0x0/0x38 returned 0 after 38 usecs
[    2.565848] calling  init_ramfs_fs+0x0/0x1c @ 1
[    2.570414] initcall init_ramfs_fs+0x0/0x1c returned 0 after 2 usecs
[    2.576791] calling  init_nls_cp437+0x0/0x1c @ 1
[    2.581439] initcall init_nls_cp437+0x0/0x1c returned 0 after 0 usecs
[    2.587913] calling  init_nls_iso8859_1+0x0/0x1c @ 1
[    2.592899] initcall init_nls_iso8859_1+0x0/0x1c returned 0 after 1 usecs
[    2.599719] calling  ipc_init+0x0/0x2c @ 1
[    2.603838] msgmni has been set to 1500
[    2.607713] initcall ipc_init+0x0/0x2c returned 0 after 3788 usecs
[    2.613917] calling  proc_genhd_init+0x0/0x60 @ 1
[    2.618663] initcall proc_genhd_init+0x0/0x60 returned 0 after 11 usecs
[    2.625302] calling  noop_init+0x0/0x20 @ 1
[    2.629511] io scheduler noop registered
[    2.633450] initcall noop_init+0x0/0x20 returned 0 after 3844 usecs
[    2.639749] calling  deadline_init+0x0/0x20 @ 1
[    2.644294] io scheduler deadline registered
[    2.648591] initcall deadline_init+0x0/0x20 returned 0 after 4193 usecs
[    2.655227] calling  cfq_init+0x0/0xc0 @ 1
[    2.659399] io scheduler cfq registered (default)
[    2.664123] initcall cfq_init+0x0/0xc0 returned 0 after 4662 usecs
[    2.670339] calling  percpu_counter_startup+0x0/0x24 @ 1
[    2.675674] initcall percpu_counter_startup+0x0/0x24 returned 0 after 3 usecs
[    2.682843] calling  fb_console_init+0x0/0x15c @ 1
[    2.687826] initcall fb_console_init+0x0/0x15c returned 0 after 156 usecs
[    2.694639] calling  pwm_backlight_init+0x0/0x1c @ 1
[    2.700012] initcall pwm_backlight_init+0x0/0x1c returned 0 after 339 usecs
[    2.707042] calling  mxc_hdmi_i2c_init+0x0/0x20 @ 1
[    2.712071] initcall mxc_hdmi_i2c_init+0x0/0x20 returned 0 after 124 usecs
[    2.719021] calling  mxc_hdmi_init+0x0/0x1c @ 1
[    2.724044] initcall mxc_hdmi_init+0x0/0x1c returned 0 after 459 usecs
[    2.730639] calling  ldb_init+0x0/0x1c @ 1
[    2.734906] initcall ldb_init+0x0/0x1c returned 0 after 147 usecs
[    2.741073] calling  mipi_dsi_init+0x0/0x48 @ 1
```

Figure: Bootgraph dmesg

```
[0.120954 0.000037] U-Boot 2014.04-rc2-00065-ge252fd5 (Apr 09 2014 - 20:55:33)
[0.004917 0.004917]
[0.004976 0.000059] CPU:   Freescale i.MX6Q rev1.2 at 792 MHz
[0.008887 0.003911] Reset cause: POR
[0.010785 0.001898] Board: Nitrogen6X
[0.011958 0.001173] I2C:   ready
[0.013008 0.001050] DRAM:  1 GiB
[0.044810 0.031802] MMC:   FSL_SDHC: 0, FSL_SDHC: 1
[0.048911 0.004101] SF: Detected SST25VF016B with page size 256 Bytes, erase size 4 KiB, total 2 MiB
[0.111782 0.062871] WARNING: Make sure the PCIe #PERST line is connected!
[0.177754 0.065972] No panel detected: default to HDMI
[0.186735 0.008981] Display: HDMI (1024x768)
[0.221727 0.034992] In:    serial
[0.222888 0.001161] Out:   serial
[0.224740 0.001852] Err:   serial
[0.225833 0.001093] Net:   using phy at 4
[0.254749 0.028916] FEC [PRIME], usb_ether
[0.256790 0.002041] Hit any key to stop autoboot:  0
[1.345577 1.088787] mmc0 is current device
[1.347639 0.002062]
[1.347685 0.000046] MMC read: dev # 0, block # 4096, count 16384 ... 16384 blocks read: OK
[1.741680 0.393995] ## Booting kernel from Legacy Image at 12000000 ...
[1.746510 0.004830]    Image Name:   Linux-3.0.35-4.1.0+yocto+g580993
[1.750676 0.004166]    Image Type:   ARM Linux Kernel Image (uncompressed)
[1.755456 0.004780]    Data Size:    3739424 Bytes = 3.6 MiB
[1.759417 0.003961]    Load Address: 10008000
[1.761634 0.002217]    Entry Point:  10008000
[1.763783 0.002149]    Verifying Checksum ... OK
[1.805632 0.041849]    Loading Kernel Image ... OK
[1.903452 0.097820]
[1.903807 0.000355] Starting kernel ...
[1.905652 0.001845]
[1.921459 0.015807] Uncompressing Linux... done, booting the kernel.
[2.385327 0.463868] ------------ Board type Nitrogen6X/W
[2.441333 0.056006] machine_constraints_voltage: VDDD: unsupportable voltage constraints
[2.447460 0.006127] reg-fixed-voltage reg-fixed-voltage.2: Failed to register regulator: -22
[3.270131 0.822671] _regulator_get: get() with no identifier
[3.428119 0.157988] mxc_sdc_fb mxc_sdc_fb.3: ipu1-di1 already in use
[3.721037 0.292918] tsc2004_prepare_for_reading: write_cmd -5
[3.725162 0.004125] egalax_ts 2-0004: egalax_ts: failed to read firmware version
[3.730907 0.005745] ft5x06-ts 2-0038: ft5x06: Could not detect touch screen.
[3.885989 0.155082] imx-hdmi-soc-dai imx-hdmi-soc-dai.0: Failed: Load HDMI-video first.
[3.892905 0.006916] sgtl5000 0-000a: Failed to get supply 'VDDD': -19
[3.985962 0.093057] Initialize HDMI-audio failed. Load HDMI-video first!
[4.017854 0.031892] rtc-isl1208 0-006f: hctosys: invalid date/time
[4.197942 0.180088] INIT: version 2.88 booting
[4.416865 0.218923] Starting udev
[6.032501 1.615636] Starting Bootlog daemon: bootlogd.
[6.083496 0.050995] Populating dev cache
[6.605298 0.521802] Configuring network interfaces... udhcpc (v1.21.1) started
```

Figure: Grabserial 输出

测量工具

- bootcmd `initcall_debug printk.time=y quiet init=/sbin/bootchartd`
- 需要在主脚本内加 `/sbin/bootchart`

测量工具

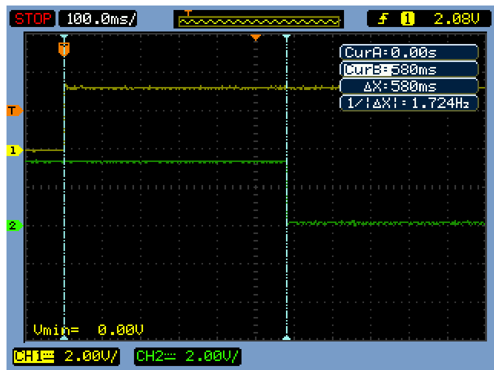


Figure: GPIO: 测量 boot rom 用的时间

- 黄色线: 复位按钮: 按下时为低位
- 绿色线: `GPIO2_2`: 系统开机预设为高位

串口输出

串口输出一般配置为 115200 bauds

- 一般 u-boot 输出约为 500 字符: 4ms
- 一般 kernel 输出约为 30000 字符: 260ms
- 串口输出可以使开机时间多一秒以上

加入 `quiet` 内核指令或简单的移除 printk 的支持

移除 printk 支持

移除 prink 支持会因为下面二个原因把开机速度提升

- 没有串口输出
- 较小的内核（500KB 未压缩）

预先定义 LPJ

在开机过程中 Linux 内核会计算 `loop_per_jiffy`

- 在一些系统中 CPU 可能用时 250 ms 来计算 LPJ
- 在 i.MX6 需要 80ms 左右
- 可以在 bootargs 内预先定义 LPJ 来省下这时间

SMP

启动一个 SMP 系统需要大量的时间。

- 启动 1 个 CPU 用时 80ms
- Bootargs `maxcpus=1`
- Init script
  `echo 1 > /sys/devices/system/cpu/cpu[123]/online`

U-BOOT 倒计时

在 uboot 启动后会倒计时 CONFIG_BOOTDELAY，如果没有收到用户的输入则自动去执行宏CONFIG_BOOTCOMMAND中设置的命令这个倒计时一般为一到十秒

```
1 /* nitrogen6x.h */
2 #define CONFIG_BOOTDELAY                1
```

Bootdelay (sec)	Percentage
0	8%
1	30%
2	3%
3	25%
5	30%
10	4%

Table: Use of the autoboot timeout

KERNEL 大小

把 kernel 从永久存储加载到 RAM 是需要一个不可忽略的时间.
把 kernel 的大小减半可省下不少时间 Kernel 的大小可以通用两种方法来减小

- 压缩
- 配置

内核压缩

Kernel 可以使用不同的演算法来进行压缩。每一个都具有不同的特性

- 压缩速度
- 解压缩速度
- 压缩比

压缩方法

用那种演算法来进行压缩取决于 CPU 速度和记忆体频宽。一般通过测量不同的配置来决定

内核压缩

- 没有: 不需要解压, 但内核最大
- GZIP: 标准压缩比、标准解压/ 压缩速度
- LZMA: 最佳压缩比, 但解压和压缩都比较慢
- XZ (LZMA2): 和 LZMA 差不多
- LZO: 压缩比差, 但解压和压缩都比较快

内核压缩

内核提供了一个配置机制，这使 Linux 可用于嵌入式系统和超级计算机小心配置是减少内核大小和运行的代码数量的最佳方式，

注意

- 去掉一些选项可能会使系统无法启动
- 把之前去掉的选项重新选上可能也无法解决问题

每次修改内核配置前应先 commit 之前的改动和每次只修改一个选项

内核配置

- Mtd support
  - Device Drivers -> Memory Technology Device (MTD) support
  - 省掉大约 **700kB**
- Block support
  - Device Drivers -> Enable the block layer
  - 省掉大约 **1.2MB**
- Sound support
  - Device Drivers -> Sound card support
  - 省掉大约 **300kB**
- Misc drivers
  - Device Drivers
  - USB, SATA, Network, MMC, Staging

内核配置

- Networking stack
  - Networking support
  - 省掉大约 **2MB**
- Kernel .config support
  - General setup -> Kernel .config support
  - 省掉大约 **80kB**
- Optimize for size
  - General setup -> Optimize for size
  - 省掉大约 **500kB**

内核配置

- Printk support
  - General setup -> Configure standard kernel features -> Enable support for printk
  - 省掉大约 **500kB**
- BUG() support
  - General setup -> Configure standard kernel features -> BUG() support
  - 省掉大约 **100kB**
- Debug Filesystem
  - Kernel hacking -> Compile-time checks and compiler options -> Debug Filesystem
  - 省掉大约 **80kB**
- Debug symbols
  - General setup -> Configure standard kernel features -> Load all symbols for debugging/ksymoops
  - 省掉大约 **700kB**

INITRAMFS 大小

当 kernel 启动时。`initramfs` 会被用作为初始 rootfs 它会被用作为`tmpfs`(RAM 文件系统), 因此 init 进程会运行得比较快它只需要包含一些关键服务, 其他非关键的文件可以放在其他分区内

initramfs 压缩

如果你打算把 initramfs 附加到 Linux 内核, 请不要压缩 initramfs, 因为它会包含在压缩内核内

INITRAMFS 大小

为了保持小规模的 initramfs, BusyBox 是完美搭配

- http://www.busybox.net
- http://wiki.musl-libc.org/wiki/Alternative_libraries

MKLIBS[步骤 1]

为了使 initramfs 尽可能小，有一个工具可以帮助我们：`mklibs`

- 它可以分析 ELF 可执行档来检测 symbols 和依赖性
- 从依赖性中, 只复制所需要的库
- 但它不能检测 dlopen

有 2 个版本

- Debian: python
- Gentoo: shell

使用 mklibs 可以很棘手，因此，建议使用 buildroot `BR2_ROOTFS_POST_BUILD_SCRIPT` 的实例。

MKLIBS [步骤 1]

例子:

```
1  MKLIBS=$(which mklibs)
2  SYSROOT="-L $BASE_DIR/target.full/lib  $BASE_DIR/target.full/usr/
       lib"
3  OUTPUT="$BASE_DIR/target/lib/"
4  BIN="$BASE_DIR/target/bin/*"
5
6  export OBJDUMP=arm-buildroot-Linux-uclibcgnueabi-objdump
7  export OBJCOPY=arm-buildroot-Linux-uclibcgnueabi-objcopy
8  export GCC=arm-buildroot-Linux-uclibcgnueabi-gcc
9
10 $MKLIBS $SYSROOT -d $OUTPUT $BINS
```

INIT 脚本定制

Buildsystem/distributions 提供了一个 init 来完成引导进程尽可能提早启动一些关键的服务, 并确保他们没有依靠大多其他的服务.

INIT 脚本定制

sysV init inittab 和 rcS 在很多嵌入式系统中都使用。

- init
  - busybox 内有提供init
  - init 是一个 ELF 可执行文件
- inittab
  - inittab 是一个设定档
  - inittab 提供重启功能
  - inittab 最终会加载 rcS
- rcS
  - 是一个 shell 脚本
  - fork/exec 所有使用者服务

INIT 脚本定制

```
1  # Startup the system
2  null::sysinit:/bin/mount -t proc proc /proc
3  null::sysinit:/bin/mkdir -p /dev/pts
4  null::sysinit:/bin/mkdir -p /dev/shm
5  null::sysinit:/bin/mount -a
6  null::sysinit:/bin/hostname -F /etc/hostname
7  # now run any rc scripts
8  ::sysinit:/etc/init.d/rcS
9
10 # Put a getty on the serial port
11 ttyS0::respawn:/sbin/getty -L ttyS0 115200 vt100
12
13 # Stuff to do for the 3-finger salute
14 ::ctrlaltdel:/sbin/reboot
15
16 # Stuff to do before rebooting
17 null::shutdown:/etc/init.d/rcK
18 null::shutdown:/bin/umount -a -r
19 null::shutdown:/sbin/swapoff -a
```

进一步深入

进一步深入

U-BOOT FALCON 模式

MX6 系列的 SoC 可以不使用 SPL 因为 BOOT ROM 可读取用 IVT header 和 DCD table 来初始化外部 RAM

- Bootloader entry point
- Device Configuration Data

当 bootloader 开始时，外部 ram 和主时钟都初始化好了

U-BOOT FALCON 模式

SPL 有另一个特色名为 *Falcon* 模式, 它允许跳过 bootloader 步骤，并直接运行操作系统。

```
1  /* SPL target boot image */
2  #define CONFIG_CMD_SPL
3  #define CONFIG_SPL_OS_BOOT                /* falcon mode */
4  #define CONFIG_SYS_SPL_ARGS_ADDR          0x4f542000
5
6  /* SPL Support for MMC */
7  #define CONFIG_SPL_MMC_SUPPORT
8  #define CONFIG_SPL_GPIO_SUPPORT
9  #define CONFIG_SPL_MMC_MAX_CLK 198000000
10 #define CONFIG_SPL_BOOT_DEVICE BOOT_DEVICE_MMC1
11 #define CONFIG_SPL_BOOT_MODE MMCSD_MODE_RAW
12 #define CONFIG_SYS_MMCSD_RAW_MODE_U_BOOT_SECTOR 2
13 #define CONFIG_SYS_MMCSD_RAW_MODE_ARGS_SECTOR 0x400
14 #define CONFIG_SYS_MMCSD_RAW_MODE_ARGS_SECTORS 1
15 #define CONFIG_SYS_MMCSD_RAW_MODE_KERNEL_SECTOR 0x800
```

U-BOOT FALCON 模式

u-boot SPL 模式在大部分 SOC/CPU/reference 版子上都支持

- Generic
  - Common/spl/spl.c
  - Common/spl/spl_fat.c
  - Common/spl/spl_mmc.c
  - Common/spl/spl_nand.c
  - Common/spl/spl_nor.c
  - ...
- ARM specific
  - Arch/arm/lib/spl.c
- SOC specific
  - Arch/arm/cpu/armv7/mx6/spl.c
  - Arch/arm/cpu/armv7/omap-common/boot-common.c
- Board specific
  - Board/freescale/p1022ds/spl.c

U-BOOT FALCON 模式

只有一些功能需要实现

- `board_init_f`: SPL
- `spl_board_init`: SPL
- `spl_boot_device`: SPL
- `spl_boot_mode`: SPL
- `spl_start_uboot`: Falcon

U-BOOT FALCON 模式

用一个命令来产生: `#define CONFIG_CMD_SPL`

- 运行 bootcmd
- 读取 bootargs
- 产生 ATAGS
- 产生 FDT

```
1  U-boot> spl export atags ${loadaddr}
2  ## Booting kernel from Legacy Image at 12000000 ...
3    Image Name:   Linux-3.0.35
4    Image Type:   ARM Linux Kernel Image (uncompressed)
5    Data Size:    4865520 Bytes = 4.6 MiB
6    Load Address: 10008000
7    Entry Point:  10008000
8    Verifying Checksum ... OK
9    Loading Kernel Image ... OK
10   Argument image is now in RAM at: 0x10000100
```

U-BOOT FALCON 模式

`spl_start_uboot` 功能可以是静态的，但也可以读取 GPIO 来决定在运行时启动操作系统或跳到完整的 u-boot 里

自定 INIT

我们定制了一个`init`应用程式

- 它启动关键应用所依赖的服务
- 避免 forking：这需要时间（大约 100us ）
- 执行关键应用

主要的应用程式运行在 PID1

要注意,PID1 的应用是不可以崩溃的, 不然 kernel 会 panic.

DEFERRED INITCALLS

Deferred initcall 是可以分阶段初始化内核

1. 除了 deferred 以外, 所有 `initcalls` 都会先执行
2. 执行`init`进程
3. `init` 之后会通知 kernel 去执行剩下的 initcalls (deferred)

deferred initcall 补丁

Mainline linxu kernel 中没有 deferred initcalls 支援，须要打补丁 (http://eLinux.org/Deferred_Initcalls):

- 2.6.26, 2.6.27: patch by Tim Bird
- 2.6.28: patch by Simonas Leleiva
- 3.10: patch by Alexandre Belloni

DEFERRED INITCALLS

`initcall` 是一个 marco, 它把所有在 module 中 inicall 的函数都放到 ELF 的一个特定的 section 上

```
1 module_init(my_module_init);
2 module_exit(my_module_exit);
```

在启动时，内核上运行的所有`initcall`，之后启动`init` 进程。关键应用可能不需要一些内核模组。使用一个小的代码修改，可以告诉内核跳过那些模组, 它会被放到一个新的 section 上。

```
1 #define deferred_initcall(fn) \
2         static initcall_t __initcall_##fn \
3         __used __section(.deferred_initcall.init) = fn
```

DEFERRED INITCALLS

确定那个模组 deferred initcalls 后, 每个模组都需要特定修改

```
1 deferred_module_init(my_module_init);
2 module_exit(my_module_exit);
```

当 init 进行后, 可以通过以下的命令, 通知内核初始化 deferred initcalls

```
1 root@target:~# cat /proc/deferred_initcalls
```

自订 TOOLCHAIN

不是所有的工具都是一样的。用不同的 toolchain，通常会有几百 ms 差别

- Prebuilt toolchain 是通用的，支援许多 SoC
- 可用这些参数优化 `-mcpu=` `-march=` `-mtune=` options

不同的 c library 也会有不同影响

- Uclibc: http://www.uclibc.org/
- Musl libc: http://www.musl-libc.org
- Dietlibc: http://www.fefe.de/dietlibc
- Newlib: http://sourceware.org/newlib

MKLIBS [步骤二]

mklibs 可以把 library 中没有调用到的函数拿掉

- 需要一个特定的 toolchain 包含 *libxxxx_pic.a*

```
1  MKLIBS=$(which mklibs)
2  SYSROOT="$BASE_DIR/target.full/lib"
3  OUTPUT="$BASE_DIR/target/lib/"
4  BIN="$BASE_DIR/target/bin/*"
5
6  export OBJDUMP=arm-buildroot-Linux-uclibcgnueabi-objdump
7  export OBJCOPY=arm-buildroot-Linux-uclibcgnueabi-objcopy
8  export GCC=arm-buildroot-Linux-uclibcgnueabi-gcc
9
10 $MKLIBS -L $SYSROOT -d $OUTPUT $BINS
```

STATIC /DEV

- udev
- mdev (`mdev -s` 可以花上 100ms)

如果系统不需要热插拔，静态 device node 或`devtmpfs` 会更快
Linux 系统需求最少两个 device nodes:

- `/dev/null`
- `/dev/console`

```
mknod -m 622 /dev/console c 5 1
mknod -m 666 /dev/null c 1 3
```

真是 INITRAMFS 更快吗?

`initramfs` 中从 RAM 中运行，并因此有更好的带宽

- 它使内核更大
- 复制数据两次
- 不管是否需要, 整个 initramfs 内的数据都会被加载

一些比较：

- 使用 1MB 的 cpio （500KB CRAMFS ）, initramfs 会比 cramfs 快 400ms
- 使用 10MB 的 cpio （5MB CRAMFS ）, initramfs 会比 cramfs 快 100ms
- 使用 18MB 的 cpio （13MB CRAMFS ）, cramfs 会比 initramfs 快 300ms

如何保持一个完整系统

如何保持一个完整功能的系统

- 减少内核的大小会导致放弃支援一些周边设备
- 减少应用程式的大小会导致失去额外功能

在一个特定的系统上只做一个特定任务是没有问题的

TRADEOFF WHEN DOING QUICK BOOT

现在的智能设备需要的功能越来越多了

- 连线: WiFi, 蓝牙, NFC
- 丰富的用户介面：网络服务器，图形工具，OpenGL 库

这些都需要完整的 Linux 系统。

内核方面

通常有 3 种配置

- Disabled
- Built-in
- Module

第三个选择是我们有兴趣的：热插拔

modules

在内核配置，不是所有的条目可以作为外部模组。只有 *tristate* 的可以例如：*Networking support* 是不支持模组

内核方面

我们的目标是把额外的功能都编译为外部内核模块

- 编译命令为 `make modules`
- Install 时用 `make modules_install` 陪随 `INSTALL_MOD_PATH=<rootfs path>`
- 把它们存储在其他分区内
- 运行关键应用程式后加载或用 udev/ MDEV 来加载

USERLAND SIDE

从

- 小的 tmpfs 的根文件系统
- 自定义 init 启动
- 运行主要应用

到

- 全功能的根文件系统
- 运行标准的 init

USERLAND SIDE

解决的办法是:

- 加载一个新的根文件系统
- 准备内容
- 取代以往的 /
- 执行 `init`

有两个方案:

- `pivot_root`
- `switch_root`

PIVOT_ROOT

pivot_root 是一个简单但比较老的工具

- 它只会改变当前进程的 rootfs 文件系统
- 它可以从 old-root 切换成 new-root
- 需要用 chroot 来做一个完整的切换
- 不允许运行新的 init

```
1 root@target:~# mount /dev/mmcblk0p1 /new-root
2 root@target:~# mount --move /sys /newroot/sys
3 root@target:~# mount --move /proc /newroot/proc
4 root@target:~# mount --move /dev /newroot/dev
5 root@target:~# cd /new-root
6 root@target:~# pivot_root . old-root
7 root@target:~# exec chroot . sh <dev/console >dev/console 2>&1
8 root@target:~# umount /old-root
```

SWITCH_ROOT

switch_root 比较完整, 在initramfs 里用了这个解决方法

- 它可以切换 rootfs
- 它执行 chroot 然后释放之前的 console
- 它清除了 old root 所有的文档
- 然后执行 new-root 文件系统内的 init

```
1 root@target:~# mount /dev/mmcblk0p1 /new-root
2 root@target:~# mount --move /sys /newroot/sys
3 root@target:~# mount --move /proc /newroot/proc
4 root@target:~# mount --move /dev /newroot/dev
5 root@target:~# exec switch_root /newroot /sbin/init
```

总结

结果与演示

进展

- 标准系统: **4.466984 秒**
- bootdelay + bootargs: *2.324392* 秒
- Falcon 模式: *1.737441* 秒
- Stripped kernel: (down to 1.9MB with initramfs)*1.405953* 秒
- Stripped initramfs: *1.162203* 秒
- Deferred initcall: **0.886289 秒**

总结

Quick boot 总是需要取舍的

Quick boot 总是需要取舍的

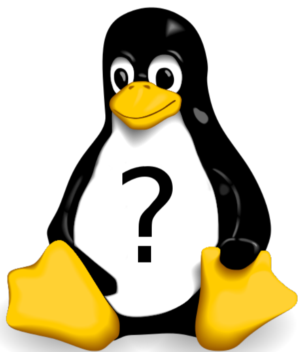
CONCLUSION

- quick boot 一般都是定做的

但

- 也有很多 open source 方案存在

问题



参考文献

- http://www.denx.de/wiki/U-Boot
- https://www.kernel.org/
- http://www.etalabs.net/compare_libcs.html
- http://eLinux.org/Deferred_Initcalls
- http://eLinux.org/Boot_Time
- http://free-electrons.com/doc/training/boot-time/slides.pdf