



CV180X & CV181X 双系统快启使用手册

Version: 1.0.1

Release date: 2023-02-06

©2023 北京晶视智能科技有限公司
本文件所含信息归北京晶视智能科技有限公司所有。

未经授权，严禁全部或部分复制或披露该等信息。

目录

1	声明	2
2	系统启动优化概述	4
2.1	免责声明	4
2.2	快速启动概述	4
2.3	快速启动的主要目标	5
2.4	快速启动总体参考方案	5
2.5	快速启动优化方法汇总表	5
2.6	时间统计方法	8
2.6.1	自定义添加统计时间戳	8
2.7	快速启动版本开发建议	8
3	硬件优化指南	12
3.1	Flash 器件选型	12
4	Fsbl 优化指南	13
4.1	FSBL 优化概述	13
4.2	FSBL 总体优化手段	13
4.2.1	调试打印裁剪	13
4.2.2	加载镜像大小裁剪	13
4.2.3	提高镜像加载速率	14
5	OpenSBI 优化指南	15
5.1	OpenSBI 优化概述	15
5.2	OpenSBI 总体优化手段	16
5.2.1	调试打印裁剪	16
5.2.2	OpenSBI 初始化时间优化	16
6	Kernel 优化指南	17
6.1	Kernel 快启动耗时统计	17
6.2	Kernel 优化概述	18
6.3	Kernel 配置说明	18
6.3.1	SDK 配置	18
6.4	Kernel 快启策略	20
6.5	deferred initcall 开发使用说明	22
6.5.1	概述	22
6.5.2	使用说明	23

7	Rootfs 优化指南	24
7.1	Rootfs 概述	24
7.2	init 进程执行顺序优化	24
7.3	busybox 功能裁剪与添加	25
7.4	ko 打包到 data 分区	28
7.5	文件系统注意事项	28

修订记录

Revision	Date	Description
1.0.0	2023/1/3	Initial version



法律声明

本数据手册包含北京晶视智能科技有限公司（下称“晶视智能”）的保密信息。未经授权，禁止使用或披露本数据手册中包含的信息。如您未经授权披露全部或部分保密信息，导致晶视智能遭受任何损失或损害，您应对因之产生的损失/损害承担责任。

本文件内信息如有更改，恕不另行通知。晶视智能不对使用或依赖本文件所含信息承担任何责任。

本数据手册和本文件所含的所有信息均按“原样”提供，无任何明示、暗示、法定或其他形式的保证。晶视智能特别声明未做任何适销性、非侵权性和特定用途适用性的默示保证，亦对本数据手册所使用、包含或提供的任何第三方的软件不提供任何保证；用户同意仅向该第三方寻求与此相关的任何保证索赔。此外，晶视智能亦不对任何其根据用户规格或符合特定标准或公开讨论而制作的可交付成果承担责任。

联系我们

地址 北京市海淀区丰豪东路 9 号院中关村集成电路设计园 (ICPARK) 1 号楼

深圳市宝安区福海街道展城社区会展湾云岸广场 T10 栋

电话 +86-10-57590723 +86-10-57590724

邮编 100094 (北京) 518100 (深圳)

官方网站 <https://www.sophgo.com/>

技术论坛 <https://developer.sophgo.com/forum/index.html>

系统启动优化概述

2.1 免责声明

本文档所述的快速启动技术主要为客户开发提供参考。采用这些优化策略前，用户需充分理解和评估潜在风险。您可选择不遵循文中提及的快速启动策略，但若决定用，则视为您已接受本声明，并明白使用这些方法可能涉及的风险。

2.2 快速启动概述

对消费类 Camera 产品来说，系统启动时长直接关系到用户体验和电池续航能力。因此，减少启动所需时间是提升产品性能的关键技术之一。

CV181x 和 CV180x 系列 SDK 提供了一套完整的快速启动解决方案。本文将详细介绍这些优化措施，以协助客户减少开发周期，并增强产品的市场竞争力。值得注意的是，由于每款芯片和产品规格的不同，启动过程和时间也会相应变化，因此本文介绍的快速启动方案应作为参考，并根据具体产品特性选择性采用。

CV181x 和 CV180x 芯片配置了双 CPU 架构，考虑到业务需求和性能要求，采用了双操作系统的软件解决方案。其中，AliOS 在 CPU1 上运行，主要负责媒体服务；而 Linux 则部署在 CPU0 上，主要处理用户界面交互、网络通讯和存储等任务。

2.3 快速启动的主要目标

消费类 Camera 产品的快速启动主要致力于实现以下目标：

- 设备开机后迅速进入视频预览界面。
- 开机后可快速进行人机交互。
- 开机后能快速进行拍照或录像，以减少等待时间并延长电池使用寿命。

本文旨在围绕上述目标展开对快速启动技术的优化措施进行详细阐述, 接下来的内容将详细介绍实现这些目标的具体优化策略。

2.4 快速启动总体参考方案

消费类 Camera 产品的快速启动参考方案主要采用硬件和软件两方面的优化措施。其中，为了更有效地启动，采用了双系统架构，具体的启动流程如图所示。

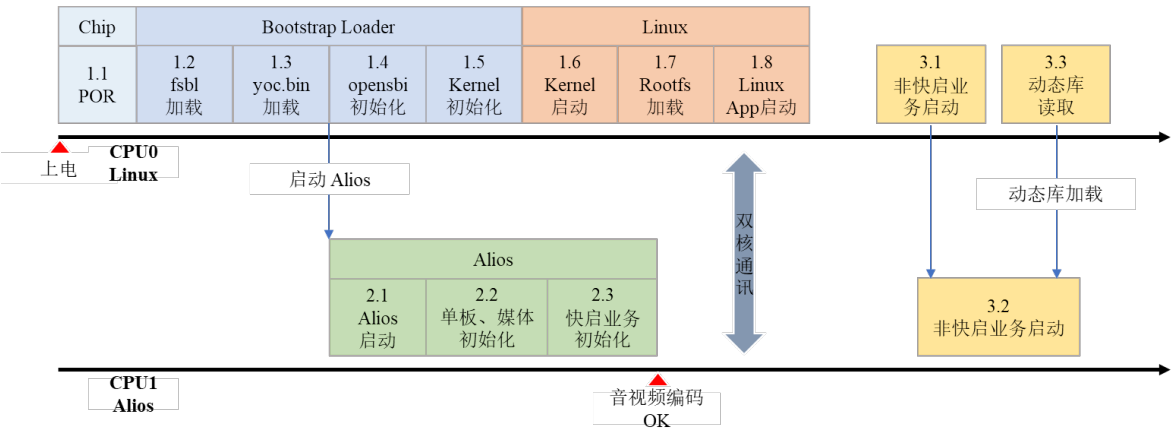


图 2.1: 双系统快速启动流程

2.5 快速启动优化方法汇总表

下方列出了 SDK 中用到的快速启动优化方法。

表 2.1: 硬件快速启动优化方法

分类 1	分类 2	优化说明	优化对象	产品优化工作	收益 (ms)
采用高性能 Flash 器件	/	Flash 存储的读取性能对系统的启动速度有显著影响。通过提升镜像和文件的读取效率，可以有效地改善整个系统的启动性能，详见“硬件优化指南”。	硬件（会影响 AliOS、Kernel 启动时间）	请根据产品要求做选型。	-
DDR 选项	/	CV181x 和 CV180x 芯片的 DDR Training 时间大约为 6ms，目前不能进行优化。	硬件	请根据产品要求做选型。	~32
ROM 阶段优化	关闭 uart & download	对于量产产品，如果不再需要裸烧功能时，可以关闭 ROM 阶段部分功能，详见“硬件优化指南”。	硬件	请根据产品要求做选型。	~1000
boot 阶段优化	/	提升 CV181x 和 CV180x 芯片 boot 阶段的 CPU 频率，目前不能进行优化。	硬件	请根据产品要求做选型。	~40

表 2.2: 软件快速启动优化方法

分类 1	分类 2	优化说明	优化对象	产品优化工作	收益 (ms)
通用	删除调试信息 Flash 时间。	优化镜像尺寸以达到减少读 Flash 时间。	所有软件组件	已提供参考实现, 请根据产品要求落实。	~100
通用	关闭打印	开机打印对启动性能有很大的影响, 快速启动时尽可能把所有打印关闭。	所有软件组件	已提供参考实现, 请根据产品要求落实。	~800
通用	组件裁剪	在不影响产品功能的情况下, 删除不必要的模块或采用更好的算法, 如跳过 uboot 阶段。	部分软件组件	已提供参考实现, 请根据产品要求落实。	~300
OS 选项	FSBL 优化	减少 FSBL 启动时间。详见“FSBL 优化指南”。	FSBL	已提供参考实现, 请根据产品要求落实。	~200
OS 选项	OpenSBI 裁剪	减少 OpenSBI 启动时间。OpenSBI 只提供必要的启动等功能, 详见“OpenSBI 优化指南”。	OpenSBI	已提供参考实现, 请根据产品要求落实。	~33
OS 选项	Kernel 裁剪	减少 Kernel 镜像尺寸, 裁剪启动流程, 减少 Kernel 启动时间。请根据产品规格对 Kernel 进行裁剪, 详见“Kernel 优化指南”。	Kernel	已提供参考实现, 请根据产品要求落实。	~150
OS 选项	Rootfs 优化	减少 Rootfs 挂载时间, 请根据产品规格对 Rootfs 进行了优化详见“Rootfs 优化指南”。	Rootfs	已提供参考实现, 请根据产品要求落实。	~50
OS 选项	文件系统选项	减少文件系统挂载和读文件时间, 建议采用 ubifs 或 squashfs 文件系统, 加快启动速度。	Rootfs	已提供参考实现, 请根据产品要求落实。	-
OS 选项	不采用压缩	减少读 Flash 时间, AliOS 和 Kernel 镜像、参数文件都不使用硬件解压。	Alios & Kernel	已提供参考实现, 请根据产品要求落实。	~60

表 2.3: Reference Design 优化

分类 1	分类 2	优化说明	优化对象	产品优化工作	收益 (ms)
Reference Design 优化	延后初始化模块	对于非开机业务模块，可以在媒体初始化出图后再进行启动，即可以加快启动速度，同时还可以降低功耗，详见“Reference Design 优化指南”。	Reference Design	已提供参考实现，请根据产品要求落实。	-
Reference Design 优化	提前初始化音视频	-	Reference Design	已提供参考实现，请根据产品要求落实。	-

2.6 时间统计方法

1. C 语言下调用：read_time_ms() 保存时间戳；
2. 系统启动完成后调用 cat /tmp/boottime 一次性打印时间戳。

2.6.1 自定义添加统计时间戳

此处以 CV181x 为例，在 bl2_main 中添加时间戳 fsbl_test_time，步骤如下：

1. 在 fsbl (fsbl/plat/cv181x/include/platform_def.h) 结构体中添加需要打印的时间变量；
2. 在 linux_5.10 (linux_5.10/include/linux/cost_time.h) 结构体中添加需要打印的时间变量；
3. 在需要的位置 (fsbl/plat/cv181x/bl2/bl2_main.c) 获取时间；
4. 在 ramdisk P99boottime (ramdisk/rootfs/overlay/cv181x_musl_riscv64/etc/init.d/P99boottime) 中添加需要写入 /tmp/boottime 的参数；
5. 编译升级后在大核中使用 cat /tmp/boottime 查看添加的时间戳。

2.7 快速启动版本开发建议

鉴于发布版本默认启用了快速启动功能，各组件包为了优化启动速度，进行了功能上的裁剪，包括减少了 busybox 的部分调试命令、uboot 的部分调试命令等。这些调整可能会对开发调试的效率产生一定影响。因此，开发者在项目初期应当仔细阅读本文档，并根据具体的开发需求，对 kernel 和 uboot 等进行适当的重新配置，以满足特殊的调试或业务需求。

```
struct _time_records {  
    uint16_t fsbl_start;  
    uint16_t fsbl_test_time;  
    uint16_t ddr_init_start;  
    uint16_t ddr_init_end;  
    uint16_t release_blcp_2nd;  
    uint16_t load_loader_2nd_end;  
    uint16_t fsbl_decomp_start;  
    uint16_t fsbl_decomp_end;  
    uint16_t fsbl_exit;  
    uint16_t uboot_start;  
    uint16_t bootcmd_start;  
    uint16_t decompress_kernel_start;  
    uint16_t kernel_start;  
    uint16_t kernel_run_init_start;  
} __packed;
```

图 2.2: fsbl _time_records 结构体

```
#include <linux/ktime.h>  
struct _time_records {  
    uint16_t fsbl_start;  
    uint16_t fsbl_test_time;  
    uint16_t ddr_init_start;  
    uint16_t ddr_init_end;  
    uint16_t release_blcp_2nd;  
    uint16_t load_loader_2nd_end;  
    uint16_t fsbl_decomp_start;  
    uint16_t fsbl_decomp_end;  
    uint16_t fsbl_exit;  
    uint16_t uboot_start;  
    uint16_t bootcmd_start;  
    uint16_t decompress_kernel_start;  
    uint16_t kernel_start;  
    uint16_t kernel_run_init_start;  
} __packed;  
  
#define TIME_RECORDS_ADDR 0x0E000010  
static struct _time_records *time_records =  
    (struct _time_records *)TIME_RECORDS_ADDR;
```

图 2.3: linux_5.10 _time_records 结构体

```
void bl2_main(void)
{
    enum CHIP_CLK_MODE mode;
    ATF_STATE = ATF_STATE_BL2_MAIN;
    time_records->fsbl_start = read_time_ms();
    NOTICE("\nFSBL %s:%s\n", version string, build message);
    time_records->fsbl_test_time = read_time_ms();

    INFO("sw_info=0x%x\n", get_sw_info()->value);
    INFO("fip_param1: param_cksum=0x%x param2_loadaddr=0x%x\n", fip_param1->param_cksum,
        ... fip_param1->param2_loadaddr);
}
```

图 2.4: 获取时间

```
fields="
fsbl_start
fsbl_test_time
ddr_init_start
ddr_init_end
release_blcp_2nd
load_loader_2nd_end
fsbl_decomp_start
fsbl_decomp_end
fsbl_exit
uboot_start
bootcmd_start
decompress_kernel_start
kernel_start
kernel_run_init_start
user_process_init_start
"
```

图 2.5: ramdisk P99boottime

```
[root@cvitek]~# cat /tmp/boottime
fsbl_start=1057ms
fsbl_test_time=1061ms
ddr_init_start=1079ms
ddr_init_end=1094ms
release_blcp_2nd=1408ms
load_loader_2nd_end=1435ms
fsbl_decomp_start=1435ms
fsbl_decomp_end=1446ms
fsbl_exit=1464ms
uboot_start=1464ms
bootcmd_start=1527ms
decompress_kernel_start=2284ms
kernel_start=0ms
kernel_run_init_start=0ms
Linux cvitek 5.10.4-tag--g3eda0acle3bb-dirty #1 PREEMPT Thu Jan 25 20:30:50 HKT 2024 riscv64 GNU/Linux
ROM elapsed time: 1057 milliseconds
DDR init elapsed time: 15 milliseconds
C906L FreeRTOS start at: 1408 milliseconds
Kernel run init time: 0 milliseconds
user process init time: milliseconds
```

图 2.6: 查看添加的时间戳

本章节只简述硬件中和快速启动相关的优化方案。

3.1 Flash 器件选型

不同 Flash 器件对启动速度有较大影响，本开发包针对 Flash 读速率进行了特别优化，表中的 Flash 器件参考读速率仅供参考。

表 3.1: Flash 器件参考表

Flash 类型	特性	参 考 读 速率	使用限制
SPI Nor	1 线 模 式	8 MB/S	优点：几乎所有的 SPI 兼容设备都支持单线模式，兼容性好，而且更稳定。缺点：数据只通过一个线传输，相比多线模式，传输速度较慢。
	4 线 模 式	32 MB/S	优点：数据速率比单线模式高很多，特别适合大容量数据的快速读写。缺点：由于使用了更多的数据线，硬件设计和软件实现更为复杂。

4.1 FSBL 优化概述

FSBL 启动时间主要受 DDR 初始化时间、所加载的镜像的大小、镜像加载速率等因素影响，快启版本的 FSBL 主要针对以上几点进行优化，使 FSBL 本身的启动时间缩短。

4.2 FSBL 总体优化手段

下文的优化手段均已在 FSBL 版本中实现，供客户进一步优化时参考。

4.2.1 调试打印裁剪

裁剪 FSBL 启动时输出的日志数量

4.2.2 加载镜像大小裁剪

- 裁剪 yoc.bin 的大小
- 裁剪 Linux 镜像的大小

4.2.3 提高镜像加载速率

在 FSBL 阶段提高处理器频率可以加快硬件初始化和系统配置的速度，从而缩短系统的总启动时间。

OpenSBI (Open Supervisor Binary Interface) 是一个开源的 RISC-V SBI 实现，它提供了标准的固件层，使得操作系统可以便捷地与 RISC-V 硬件进行交互。

5.1 OpenSBI 优化概述

OpenSBI 可以配置为包含不同的功能和服务，包含的功能越多，意味着更长的初始化时间。快速启动版本的 OpenSBI 主要移除不必要的初始化步骤和设备支持，使得 OpenSBI 初始化的时间缩短。

优化后的 OpenSBI 裁剪了命令行等调试功能，只具备快速启动等功能，若需要其他功能，请自行在 OpenSBI 中进行配置。

5.2 OpenSBI 总体优化手段

下文的优化手段均已在 OpenSBI 版本中实现，供客户进一步优化时参考。

5.2.1 调试打印裁剪

裁剪 OpenSBI 启动时输出的日志数量

5.2.2 OpenSBI 初始化时间优化

裁剪 OpenSBI 初始化未用到的驱动代码，优化部分模块代码

- 裁剪 `sbi_platform_early_init` 和 `sbi_platform_final_init`，它们提供了在特定时刻执行硬件或平台特定初始化代码。
- 裁剪 `sbi_console_init`，用于在系统启动过程中初始化控制台。
- 裁剪 `sbi_platform_irqchip_init`，用于初始化 OpenSBI 阶段的中断控制器。
- 裁剪 `sbi_ipi_init`，用于设置和初始化处理器间通信的硬件机制。
- 优化 `fdt_timer_cold_init`，将动态解析设备树节点改成全局静态数据的初始值。

6.1 Kernel 快启动耗时统计

对于 kernel build-in 的模块，我们主要采用 `initcall_debug` + `bootgraph` 工具进行耗时统计。

1. ubuntu 安装 `bootgraph` 工具：<<https://github.com/intel/pm-graph>>
2. 在设备树 `chosen` 节点中加入 `bootarg: initcall_debug`。

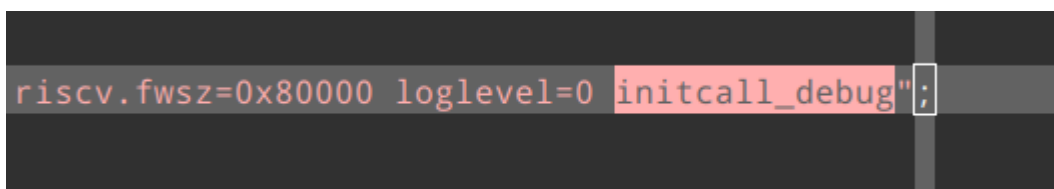


图 6.1: 设备树加入 `bootarg: initcall_debug` 节点

1. 重新编译并烧录 kernel 固件，将 kernel 启动阶段的 `dmesg log` 保存到文件 `dmesg.txt`
2. 执行以下命令生成 `html` 数据文件：

```
bootgraph -dmesg dmesg.txt -addlogs
```

3. 打开生成的 `bootgraph.html` 文件。

图 6.2: bootgraph.html 文件

为满足快速启动需求，我们对 kernel 进行了一些裁剪，主要是通过 kernel 的 CONFIG 配置进行裁剪，对于少量 kernel 代码的优化修改，本章后续会进行详细说明。

6.3 Kernel 配置说明

快启模式下 kernel 使用的配置文件是独立的，位于 `build/boards/default/linux/cvitek_{chip_name}_fastboot_riscv_defconfig`，目前该配置文件只适用于 spinor flash。

6.3.1 SDK 配置

SDK 为快启提供了一个总开关，开启后会打开所有快启相关的功能，用户可以根据自己的需求决定是否开启，下方是大核相关的 SDK 级配置：

- **CONFIG_FASTBOOT** 快启总开关，开启后会自动打开大小核除了 kernel 非压缩的所有快启相关选项。
- **CONFIG_FLASH_SIZE_SHRINK** 开启后将不会打包一些 sample 和 self test 应用程序。
- **CONFIG_NO_FP** 不编译 frame buffer 驱动。
- **CONFIG_NO_TP** 不编译触摸屏驱动。
- **CONFIG_ROOTFS_FORMAT_OPTIMIZATION** 使用 gzip 作为文件系统的压缩算法。
- **CONFIG_SKIP_UBOOT** 开启 OpenSBI 跳 kernel。
- **CONFIG_OSDRV_BUILD_IN** 开启后一些 osdrv 下的驱动将会以 build-in 的形式编译。

- `CONFIG_KERNEL_UNCOMPRESSED` 开启后关闭 kernel 压缩

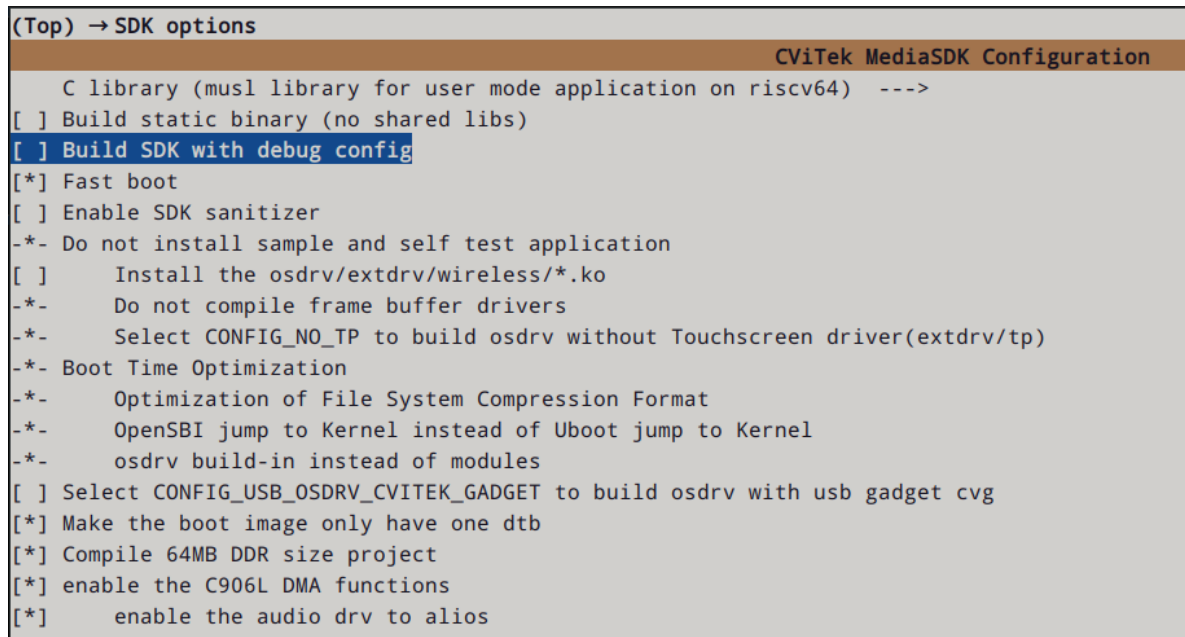


图 6.3: 大核快起 SDK 相关配置

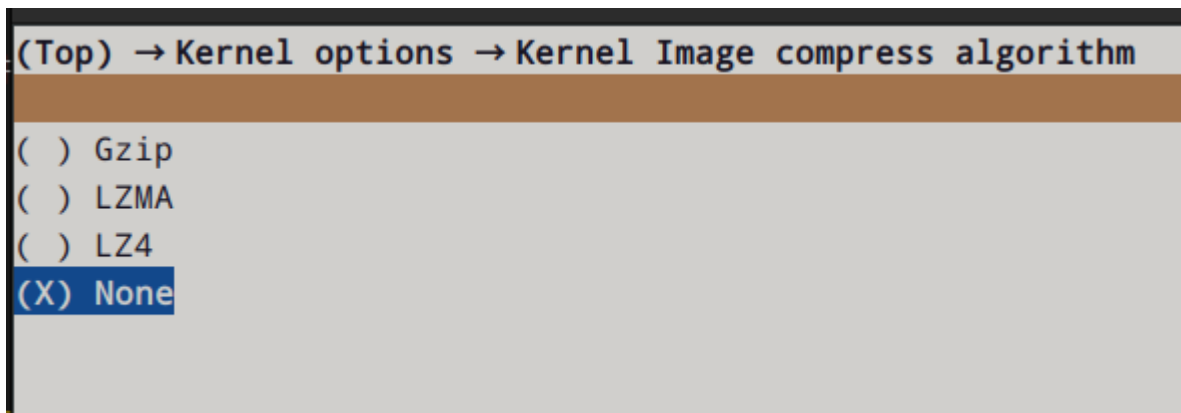


图 6.4: 大核快起 kernel 相关配置

`CONFIG_FASTBOOT` 开启后除了会自动打开所有 SDK 级的快启 `CONFIG` 外，还会自动打开 Linux 的 `CONFIG_CVITEK_FASTBOOT` 配置项 (kernel 相关快启 feature 支持)，若关闭 `CONFIG_FASTBOOT`，要使用 kernel 的相关 feature 要记得在 Linux 级的 config 配置中打开 `CONFIG_CVITEK_FASTBOOT`。

6.4 Kernel 快启策略

Linux 下所有快启相关的特性都通过一个宏 `CONFIG_CVITEK_FASTBOOT` 进行控制，用户若想自定义功能请参考下节说明。kernel 相关的快启优化主要分为两个部分：

1. 内核配置裁剪：

- net 相关 config:

```
CONFIG_NET=y
CONFIG_STMMAC_ETH=m
CONFIG_CVITEK_PHY=m
CONFIG_PHYLIB=m
CONFIG_STMMAC_PLATFORM=m
CONFIG_DWMAC_CVITEK=m
CONFIG_NET_VENDOR_GOOGLE=n
CONFIG_NET_VENDOR_MELLANOX=n
CONFIG_NET_VENDOR_PENSANDO=n
CONFIG_NET_VENDOR_XILINX=n
```

- mmc/sd 相关 config:

```
CONFIG_MMC=m
CONFIG_MMC_SDHCI=m
CONFIG_MMC_SDHCI_CVI=m
```

- usb 相关 config:

```
CONFIG_USB=m
```

- sysfs 相关 config:

```
CONFIG_CONFIGFS_FS=n
CONFIG_SYSFS=y
```

- procfs 相关 config:

```
CONFIG_PROC_FS=y
CONFIG_PROC_SYSCTL=n
CONFIG_PROC_PAGE_MONITOR=n
```

- thermal 相关 config:

```
CONFIG_THERMAL=n
```

- pwm 相关 config:

```
CONFIG_PWM=n
```

- 其他 config:


```
CONFIG_XZ_DEC_ARM=n
CONFIG_EFI_PARTITION=n
CONFIG_MSDOS_PARTITION=n
```

以上将 net/mmc/usb 相关模块以 ko 的形式进行编译，减小 kernel 固件大小并节省启动时间（主要手段，可以减少约 200ms 的启动时间），去除一些非必要的模块，如 thermal/pwm 等，用户若需要可以自行打开，大约会增加 10ms 左右的最终启动时间

2. 采用 deferred initcall 机制将一些不那么急需的模块延后到第一个应用程序之后加载，目前 defer 的模块如下：

- gpio-dwapb

gpio 驱动，若有 build-in 的驱动需要操作 gpio，则该驱动不能 defer；

- gpiolib-sysfs

gpio 的 sysfs 接口

- i2c-designware-platdrv

i2c 驱动

- mm sysfs

内存相关的 sysfs 接口

- slub sysfs

slub 分配器的 sysfs 接口

- net sysfs

网络协议栈相关 sysfs 接口

- pwm sysfs

pwm 子系统 sysfs 接口

- fdt sysfs

设备树相关 sysfs 接口（裁剪）

- 8250_dw

串口驱动

- af_net

网络协议栈

- params

module 的 sysfs 接口注册

注：若由于将自己的模块 defer 或打开某些 kernel config 后导致 kernel 启动时 panic 了，可以尝试将此模块恢复（即不 defer），或将所打开模块的 sysfs 接口也 defer 掉即可解决问题

defer 能把非必要的内核模块放到应用层去初始化，保证快启应用程序优先执行，用户可以将与应用程序初始化无关的模块 **defer** 到应用程序启动之后，具体优化时间由实际情况决定。

3. 通过对设备树一些不必要的节点进行裁剪，减小 dtb 大小和 kernel 启动时设备树解析时间，裁剪的节点主要包括：

- 多媒体相关节点，如 vcodec、dwa 等；
- 不常用的外设节点，如 keyscan、irrx 等；

注：可以通过宏 ‘**__FASTBOOT__**’ 来增减设备树节点，例如通过以下方式去掉 trng 节点。设备树的裁剪可以优化大约 10ms 的启动时间：

```
158 #ifndef __FASTBOOT__
159     trng: dw_trng@0x02070000 {
160         reg = <0x0 0x02070000 0x0 0x1000>;
161         compatible = "snps,dw-trng";
162     };
163 #endif
```

图 6.5: 通过宏 **__FASTBOOT__** 增减设备树节点

6.5 deferred initcall 开发使用说明

6.5.1 概述

Linux kernel 提供了多种 initcall 机制，其基本原理是为不同阶段需要执行的函数提供一个专门的代码段（.init 段），在启动的不同阶段进行调用。deferred initcall 机制在 Linux 的 initcall 机制基础上增加了一种 initcall 类型——deferred_initcall，使用 deferred_initcall 初始化的模块将会在用户态由用户手动触发加载。

通过 deferred initcall 机制将一些非必要的模块延迟到用户态触发加载，有以下优势：

1. 不需要像 ko 一样打包到文件系统，增加文件系统大小；
2. 由用户随时控制加载（加载之前需要占用一定的内存）；

```
cat /proc/deferred_initcalls
```

手动触发加载的操作默认放在了 S11defer_init 启动脚本中，可以根据需要自行修改，但要注意加载顺序要满足依赖关系

6.5.2 使用说明

deferred_initcall 使用主要分为两种场景：

1. 内核模块，如一些驱动模块会使用 platform 框架注册驱动：

```
module_platform_driver(dw8250_platform_driver);
```

要将其设置成 deferred_initcall，只需要在之前加上下面几行代码：

```
#ifndef CONFIG_CVITEK_FASTBOOT
#define module_init(x) deferred_initcall(x)
#endif
module_platform_driver(dw8250_platform_driver);
```

2. 内核 subsys/fs 等模块，如网络协议栈：

```
fs_initcall(inet_init);
```

要将其设置成 deferred_initcall，只需要做如下改动：

```
#ifndef CONFIG_CVITEK_FASTBOOT
fs_initcall(inet_init);
#else
deferred_initcall(inet_init);
#endif
```

通过以上修改，即可在开启 CONFIG_CVITEK_FASTBOOT 时将要配置的模块作为 deferred_initcall 延迟到用户态加载。

7.1 Rootfs 概述

SDK 使用的 rootfs 为 busybox 编译得到的根文件系统，为了满足快启需求，我们对 busybox 的一些功能进行了优化和裁剪。主要分为三个方面：

1. init 进程执行顺序优化；
2. busybox 功能裁剪；
3. ko 打包到 data 分区。

下面将详细介绍如何自己添加功能并替换到 SDK 中。

7.2 init 进程执行顺序优化

busybox 中 init 进程的代码主要位于 init/init.c 文件中，init 进程执行顺序为：

1. 注册一些必要的信号；
2. 初始化默认环境变量，如 HOME, SHELL 等；
3. 解析 inittab 文件；
4. 执行 sysinit 命令，即 inittab 中启动脚本中的命令；
5. 进入 idle 循环。

为了满足启动时间的需求，对 init 进程中 sysinit 执行的顺序进行了调整并去除了 inittab 解析功能，调整后的执行顺序为：

1. 注册一些必要的信号；

2. 初始化默认环境变量，如 HOME，SHELL 等；

3. 执行 sysinit 命令：

```
new_init_action(SYSINIT, "/etc/init.d/fastboot start", "");
new_init_action(SYSINIT, "/bin/mount -t proc proc /proc", "");
new_init_action(SYSINIT, "/bin/mount -t tmpfs -o mode=1777 none /tmp", "");
new_init_action(SYSINIT, "/sbin/mdev -s", "");
new_init_action(SYSINIT, "/etc/init.d/rcS", "");
new_init_action(SYSINIT, "/bin/hostname -F /etc/hostname", "/dev/null");
new_init_action(SYSINIT, "/etc/init.d/rcP", "");

new_init_action(RESPAWN, "/sbin/getty -L console 115200 vt100 -n -l /usr/local/bin/autologin
→", "/dev/console");
/* Reboot on Ctrl-Alt-Del */
new_init_action(CTRLALTDDEL, "reboot", "");

new_init_action(SHUTDOWN, "/etc/init.d/rcK", "");
/* Swapoff on halt/reboot */
new_init_action(SHUTDOWN, "swapoff -a", "");
/* Umount all filesystems on halt/reboot */
new_init_action(SHUTDOWN, "umount -a -r", "");
```

4. 进入 idle 循环；

可以看到，最先执行的脚本为 fastboot 脚本，在该脚本中启动了急需执行的应用程序。挂载 proc/tmpfs 等文件系统以单独的命令执行，并放在 fastboot 之后。

7.3 busybox 功能裁剪与添加

为了提升 init 进程二进制文件的解析速度，对 busybox 进行了功能裁剪并以静态编译的方式进行编译（第一个应用程序也建议静态编译），下面介绍 busybox 的编译过程（SDK 已经整合了 busybox，可以供用户自定义功能）：

1. 配置 busybox：

```
menuconfig_busybox
```

若开启了快启选项 CONFIG_FASTBOOT，busybox 的默认配置使用 cvitek_musl_simplify_defconfig，即使用最为精简的配置（该配置下 busybox 为静态编译）。

2. 添加功能：

如图，每个菜单栏都有对应的功能，如 Init Utilities 则包含了 init 进程的一些配置。

例如要添加 vi 的功能，则去到 Editors 菜单，选中 vi，并保存退出。

1. 编译 busybox：

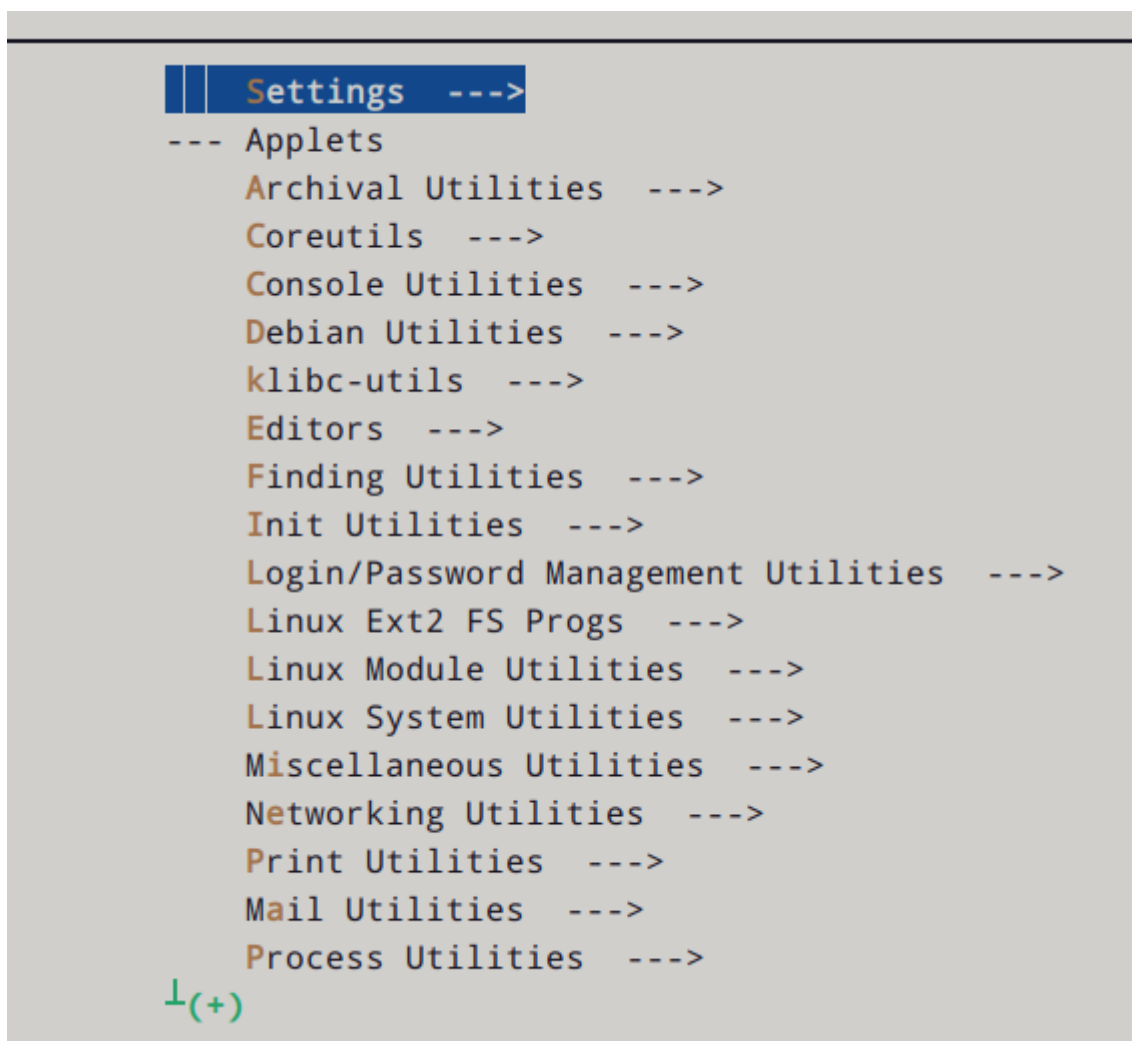


图 7.1: Init Utilities 菜单

```
[*] awk (23 kb)
[*]   Enable math functions (requires libm)
[*]   Enable a few GNU extensions
[ ] cmp (4.9 kb)
[ ] diff (13 kb)
[ ] ed (21 kb)
[ ] patch (9.4 kb)
[ ] sed (12 kb)
[*] vi (23 kb)
(4096) Maximum screen width (NEW)
[ ]   Allow to display 8-bit chars (otherwise shows dots) (NEW)
[*]   Enable ":" colon commands (no "ex" mode) (NEW)
[*]   Enable yank/put commands and mark cmds (NEW)
[*]   Enable search and replace cmds (NEW)
[ ]   Enable regex in search and replace (NEW)
[*]   Catch signals (NEW)
[*]   Remember previous cmd and "." cmd (NEW)
[*]   Enable -R option and "view" mode (NEW)
[*]   Enable settable options, ai ic showmatch (NEW)
↓(+)
```

图 7.2: Editors 菜单

```
build_busybox
```

编译完成后会自动将 busybox bin 文件以及相应的软连接复制到相应的根文件系统中。

4. 打包文件系统

```
:: pack_rootfs
```

此时重新配置的 busybox 已经打包到了根文件系统中，重新烧录即可。

7.4 ko 打包到 data 分区

为了节省 rootfs 的挂载时间，将 kernel 中需要编译成 ko 的模块打包到 data 分区，在第一个应用程序之后挂载。在编译完 kernel 后通过 pack_data 命令打包，要删除 data 中文件则通过 clean_rootfs 命令将 rootfs 和 data 一起 clean。

7.5 文件系统注意事项

1. 统一通过 CONFIG_FASTBOOT 宏进行控制：

- a. 使用裁剪的文件系统；
- b. 将 kernel 编译生成的 ko 打包到 data 分区；

当前只有在 SDK 打开 CONFIG_FASTBOOT 才会使用优化后的文件系统，若想使用优化后的文件系统，必须开启所有快启的特性（由 CONFIG_FASTBOOT 控制），但 SDK 提供了用户自由配置 busybox 的功能。

2. 开启 CONFIG_FASTBOOT 宏后，fastboot 启动脚本将会作为 kernel 启动后的第一个脚本执行；
3. fastboot 脚本作为第一个脚本执行时，data 分区还未挂载（在 S10 脚本中挂载），所以在 fastboot 中不能有对 data 分区中的数据的操作；
4. deferred_initcalls 和 data 分区中 ko 的加载在 S11defer_init 脚本（只在开启 CONFIG_FASTBOOT 后才会打包）中触发加载，所以网络 /SD/USB 等被 defer 的驱动，要在该脚本执行完才可使用；
5. 自定义快启应用程序请加在 fastboot 脚本中。