Linux GKI
开发指南

版本号: 1.5
发布日期: 2025.03.14

版本历史

版本号	日期	制/修订人	内容描述
1.0	2022.11.25	XAA0193	初版文档
1.1	2022.11.30	XAA0193	根据评审意见，针对 android13+GKI2.0 对内容进行了更新、删减、补充。
1.2	2022.12.07	XAA0193	根据评审意见，GKI 开发章节进行补充，内容的调整。
1.3	2023.3.07	XAA0193	适用范围新增了 A523 芯片。
1.4	2024.11.23	XAA0309	适用范围新增了 A733 芯片。
1.5	2025.03.14	XAA0339	适用范围新增了 A537 与 A333 芯片。

目　录

插　图

1 概述

1.1 编写目的

本文档介绍了 Google GKI2.0 的开发使用，用于 Android13 + Linux-5.15 以及更新版本内核的升级项目中，指导模块驱动进行 GKI 开发，以及指导 GKI 兼容测试及维护。本文档以 Android13-Linux-5.15 下 AW A133 P25 平台为例子进行介绍。

1.2 适用范围

本文档适用于所有需要支持 Google GKI 的平台（Linux-5.15 以及更新版本）。

Android 版本	芯片平台
Android13 Linux-5.15	A133
Android13 Linux-5.15	A523
Android15 Linux-6.6	A733
Android15 Linux-6.6	A537
Android15 Linux-6.6	A333

1.3 适用人员

- Linux 模块驱动开发人员。
- 使用 AW 芯片的下游方案开发厂商。

1.4 术语介绍

术语	解释
GKI	Generic Kernel Image
KMI	Kernel Module Interface
ACK	Android Common Kernels
LTS	Long Term Supported linux kernel

术语	解释
vendor	芯片厂商
kernel fragmentation	内核代码碎片化
GSI	Generic System Image
GKI Modules	通用内核模块，与 Generic Kernel 一起使用，组成 GKI

2 GKI 介绍

2.1 什么是 GKI

GKI，Generic Kernel Image，也就是通用内核镜像，从字面上理解就是一个通用的 kernel 镜像，可以在不同 SoC 厂商的不同设备上运行。

GKI 的目的是为了统一内核代码，把 SoC 板级驱动从内核镜像中剥离出来，并将其都编译为 ko。也就是说，Google GKI 要求 SoC 厂商在新的 android 内核代码中，不要随意去修改内核通用代码，只增加并维护好自己的 SoC 板级驱动 ko，驱动代码要与内核框架层解耦。如果需要修改内核通用代码或者提交 bug-fix，必须先提交到 linux mainline 内核分支中，再 merge 到 google 分支。

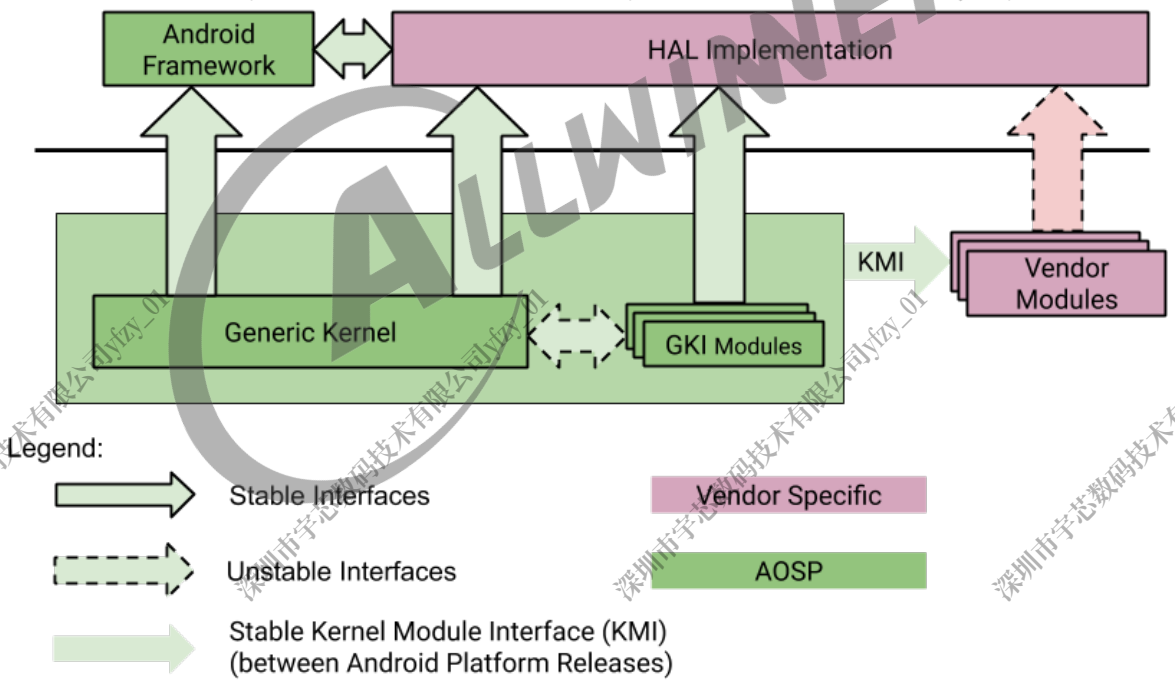

图 2-1: GKI 框架

上图是一个 Android 设备搭载 GKI 的框架图，从图中可以看到，Google 会提供 KMI 接口，用于 vendor modules 和 GKI 的通讯。几个关键词的解析如下：

- Generic Kernel：通用内核代码
- GKI Modules：通用内核模块，与 Generic Kernel 一起组成 GKI；

- KMI：GKI export 给模块 ko 使用的接口；
- Vendor Modules：芯片厂商驱动，比如 sunxi iommu 驱动；

2.2 为什么要用 GKI

先看一张在 2019 以及之前的 Google 内核发展模式图：

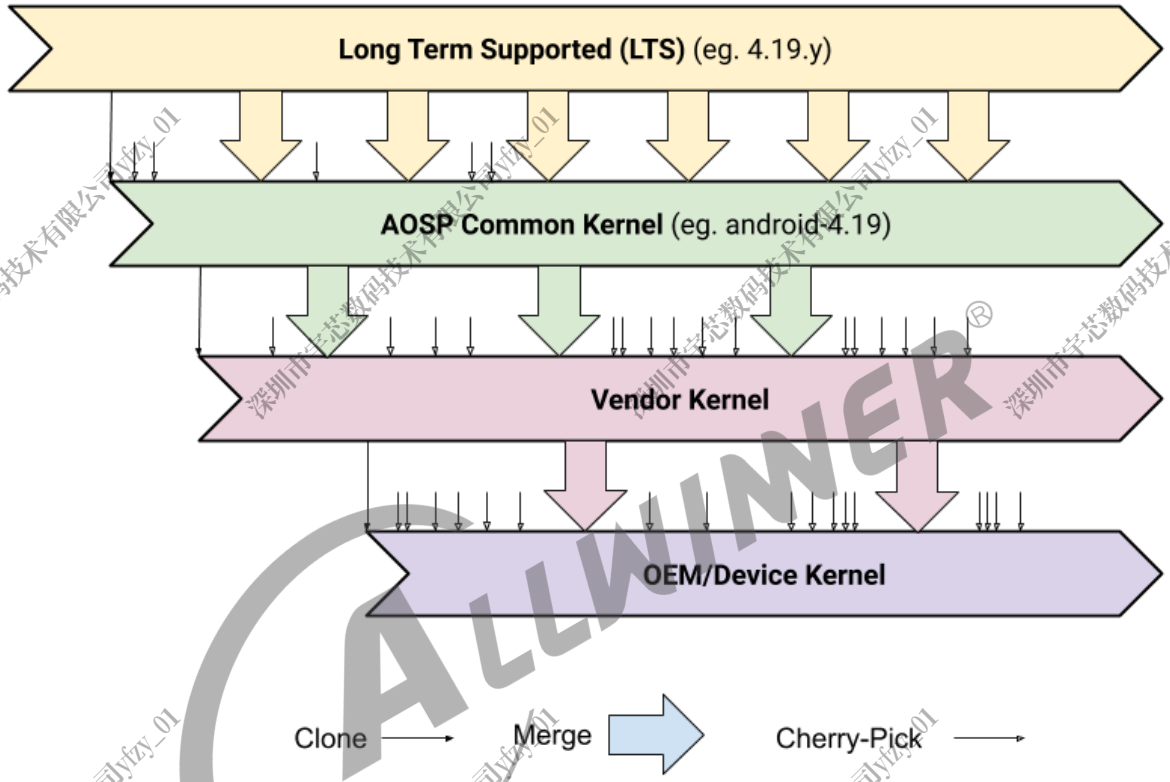

图 2-2: AOSP common kernel 老的发展模式

以 AW Android 10 的开发为例，android 内核分支发展如下：

```
LTS linux-4.9 release
  --> android-4.9
    --> android-4.9-q(Android10 feature complete)
      --> sunxi-dev linux-4.9-q
        --> AW clinet linux-4.9-q
```

在这个过程中会形成 kernel fragmentation，也即是内核代码碎片化，因为几乎每个 Android 设备都有一个定制化的内核代码，使得很多代码都是 out-of-tree 的状态（游离在上游 linux 内核和 andorid 内核之外），很难将这些代码提交到 LTS 中，同时也大大增加了补丁同步的难度，最直观的体现就是我们的内核代码往往没有及时同步上游分支中的 bug-fix 补丁、安全补丁。

为了解决 kernel fragmentation，Google 推出了新的 android 内核发展模式，如下图：

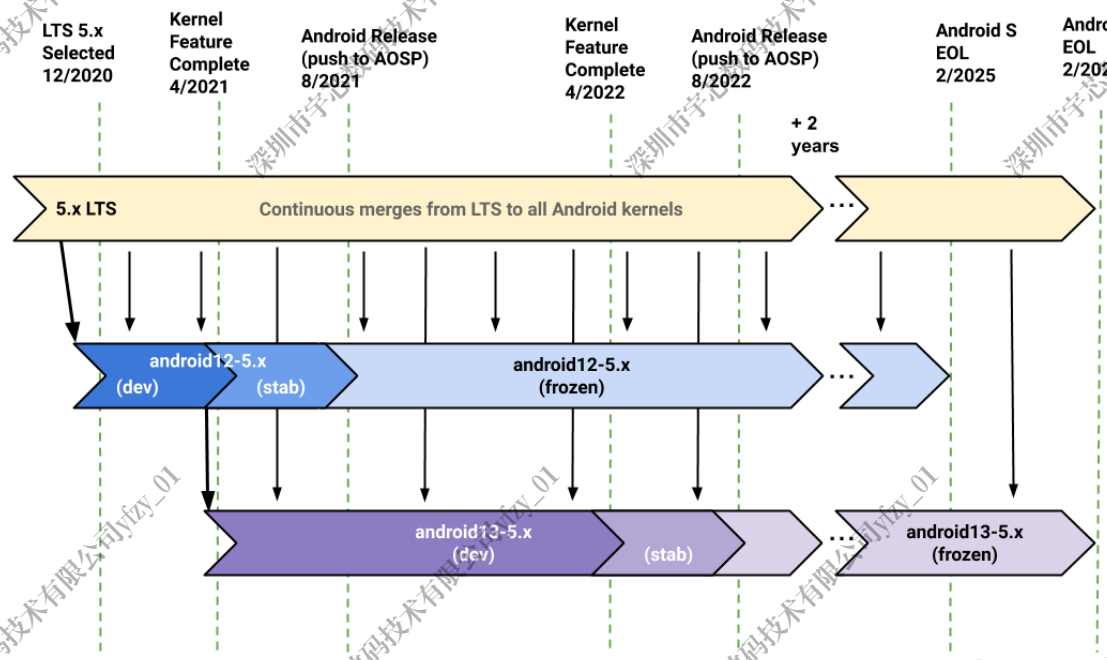

图 2-3: AOSP common kernel 新的发展模式

以目前正在进行 AW Android 13 + linux-5.15 的开发为例，android 内核分支发展如下：

```
LTS linux kernel
  --> android-mainline
    --> android13-5.15(Linux-5.15 feature complete)
      --> android13-5.15(Android13 feature complete)
        --> AW sunxi-dev-aosp linux-5.15
```

这里需要注意几个关键点：

- android-mainline：android 内核主线分支会持续从 LTS merge 补丁，并充分测试；
- android13-5.4：Android 11(R) 对应的内核分支；
- android12-5.x：Android 12(S) 对应的内核分支；
- android13-5.x：Android 13(T) 对应的内核分支；
- 所以在新的模式下，将不会有所谓的 desert-kernel 分支，比如 android-4.9-o、android-4.9-q 等；
- 假如 AW 支持了 GKI，那 sunxi-dev linux-5.15 可以毫不费劲地从 android13-5.15 上同步补丁。

在新的模式下支持 GKI，就可以解决 kernel fragmentation。

2.3 GKI Roadmap

Gogole GKI 分为下面两个阶段推进：

- I. GKI 兼容性：Android 11(R) + linux-5.4 require GKI compatibility test
- II. GKI 产品化：Android 12(S) + linux-5.10 及之后 require GKI kernel

Google 将这两个阶段称为 GKI1.0 和 GKI2.0，详细的区别如下：

2.3.1 GKI1.0

此阶段 Google 要求 SoC 厂商要通过 GKI 兼容性测试，包括：使用 Android 11 和 Linux-5.4 启动的设备必须通过 VTS 和 CTS-on-GSI + GKI 测试。GKI 1.0 目标包括以下内容。

- 用 GKI 内核替换产品内核时，避免在 VTS 或 CTS 中进行回归；
- 减轻合作伙伴的负担，使他们的内核与 AOSP 通用内核保持最新；
- 在内核中包含 Android 的核心更改，以便通过新的 Android 版本升级和启动设备；
- 不要破坏 Android 用户空间；
- 将特定于硬件的组件与核心内核分开，作为可加载模块。

2.3.2 GKI2.0

GKI2.0 支持 Android 12 和 Linux-5.10 及其后续的更高的 Android 版本和 Linux 内核版本。在此阶段中，Google 要求 SOC 厂商使用 Google 提供的 GKI 镜像进行出货，这个变化带来最大的优势就是从 Linux-5.15 开始，芯片原厂无需再为 Android 产品做内核升级了，下游的设备厂商也无需自己升级，而这部分工作将会被 Google 承担。

GKI 2.0 目标包括以下内容。

- 用 GKI 内核替换产品内核时，不要引入明显的性能或功耗下降。
- 使合作伙伴无需供应商参与即可提供内核安全修复程序和错误修复程序。
- 降低将设备的主要内核版本更新的成本（例如，从 v5.x 到 v5.y）。
- 通过使用清晰的升级过程更新内核版本，可以为每个体系结构维护一个 GKI 内核二进制文件。

这里插讲一下 GKI 兼容性测试：GKI 兼容性测试中，Google 会提供 boot.img（包含了 GKI 镜像），替换掉我们固件中的 boot.img，并将该固件烧到机器中，如果机器能成功启动，vendor-boot 中的模块 ko 能顺利加载并运行，系统功能正常，并且最终要能通过 VTS 和 CTS-on-GSI + GKI 测试。所以为了能通过测试，我们需要做到以下三点：

- build：模块驱动 (ko 文件) 的编译环境要跟 Google 编译 GKI 的环境保持一致；
- bootloader：bootloader 需要成功加载并整合 ramdisk；
- device driver：模块驱动代码需要与框架层解耦。

3 GKI 开发

如上一章节所述，andriod13-linux-5.15 中 AW 已经支持了 GKI2.0 方案，所以后文中将以 GKI2.0 下的方案开发指导作为重点，同时由于 GKI2.0 的很多功能 future 的实现来源于 GKI1.0，所以也会对 GKI1.0 的部分内容展开介绍。

3.1 GKI 启动流程

在 GKI 方案下，我们的 android 启动流程大致如下所示：

- **boot0**
- **uboot**：（1）加载 boot.img 中的 GKI 镜像；（2）加载 init_boot 分区中 google 的 ramdisk-1；（3）加载 vendor-boot.img 中的 vendor 私有的 ramdisk-2；（4）整合 ramdisk-1 和 ramdisk_2；（5）jump to GKI
- **GKI**: core kernel init
- **init**：（1）加载 ramdisk-1，启动一些必要的必要的文件系统功能；（2）加载 ramdisk-2，启动一些厂商自己的文件系统功能，其中最重要的就是加载启动第一阶段需要的 ko 文件

3.2 镜像分区和 bootloader 开发

Andriod13 较与 Andriod12 在固件的组织方面发生了很大的变化，Android 固件组织结构的变化详情如下所示

- boot.img 变为 boot.img + vendor-boot.img，其中 boot.img 中放的是 GKI 镜像（boot header + kernel 数据），vendor-boot.img 中放的是需要启动加载的 vendor 模块 ko，这样使得 boot 分区的缩小，镜像的内容更加的单一。
- 增加了 vendor_boot，将开发者的 KO 放在 vendor_ramdiskz 中，vendor_boot.img 包括 vendor_boot_header，vendor_ramdisk，dtb
- ramdisk 分为两部分：init_boot 分区中的 AOSP 原生 ramsik 和 vendor_boot 分区中的产商自己的 ramdisk，启动过程中，需要 bootloader 加载并整合 init_boot.img 和 vendor-boot.img 中的两个 ramdisk；
- 与 GKI 对应的，Google 还推出 GSI，也就是通用 system 镜像。

同时在 ramdisk 方面，ramdisk 需要 bootloader 进行拼接，才能得到完整的 ramdisk，具体的流程如下所示：

- Android 13 中，ramdisk 分为两份，一份为 boot_ramdisk，存放在 init_boot.img，另一份为 vendor_boot_ramdisk 存放在 vendor_boot.img 中；
- 在 bootloader 启动时需要先后加载 boot_ramdisk，vendor_boot_ramdisk 并进行前后拼接；
- ramdisk 用的是 cpio.lz4 格式，可以进行简单的首尾拼接，但是 bootloader 需要注意两个 ramdisk 中间必须紧密拼接，不能对齐再拼接，否则会导致内核解压时失败，同时拼接后改变 ramdisk 的大小。

关于这两部分 ramdisk 的整合，google 官方给出的推荐方式（同时也是 AW 目前采用的方式）如下所示：

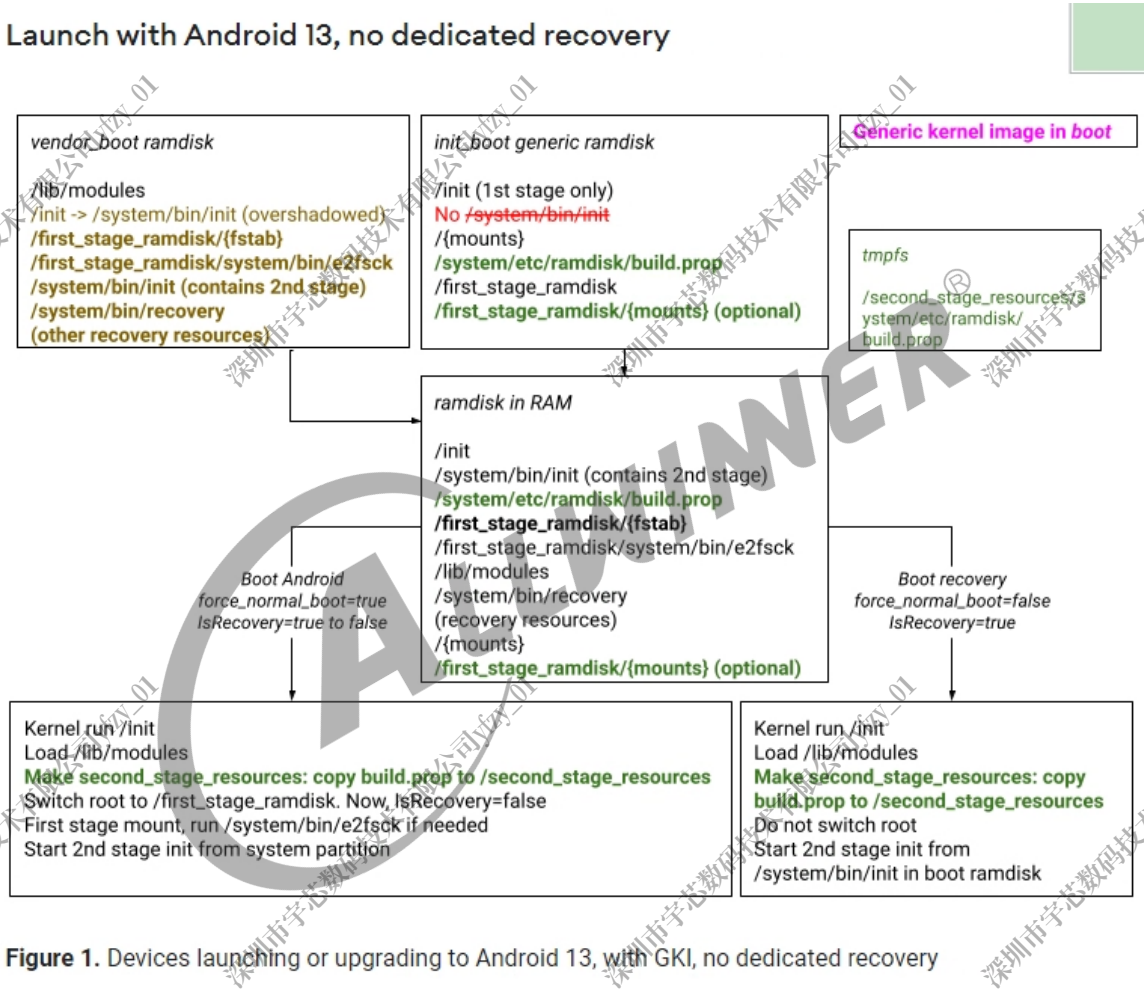

Figure 1. Devices launching or upgrading to Android 13, with GKI, no dedicated recovery

图 3-1: google-ramdisk 拼接流程

鉴于如上的新变动，android13 在启动、bootloader 的开发中带来了新的变化：

- boot header 必须使用最新的 boot Head V4.0 代替 V3.0；
- DTB 的存放位置从 boot.img 改到了 vendor-boot.img；
- 启动过程中 boot 需要加载 init_boot.img 和 vendor_boot.img 中的 ramdisk 并做整合，而且 init_boot.img 的 ramdisk 优先级高。

下面我们将对这些变化和开发、使用详情进行介绍。

3.3 HeadV4

在 Android13 中使用了对镜像统一使用 HeadV4 格式，HeadV4 格式镜像的定义如下：

```
/* When the boot image header has a version of 4, the structure of the boot
 * image is as follows:
 *
 * +---------------------+
 * | boot header       | 4096 bytes
 * +---------------------+
 * | kernel            | m pages
 * +---------------------+
 * | ramdisk           | n pages
 * +---------------------+
 * | boot signature    | g pages
 * +---------------------+
 *
 * m = (kernel_size + 4096 - 1) / 4096
 * n = (ramdisk_size + 4096 - 1) / 4096
 * g = (signature_size + 4096 - 1) / 4096
 *
 * Note that in version 4 of the boot image header, page size is fixed at 4096
 * bytes.
 *
```

google 官方对于这个 Head 头部信息定义如下位置中：

```
aosp/tools/mkbootimg/include/bootimg/bootimg.h
```

HeadV4 较与 HeadV3 的特点如下所示：

- 支持多个 vendor ramdisk
- 支持 bootconfig（对于内核的传参可以使用该方式），下文将继续介绍次功能。

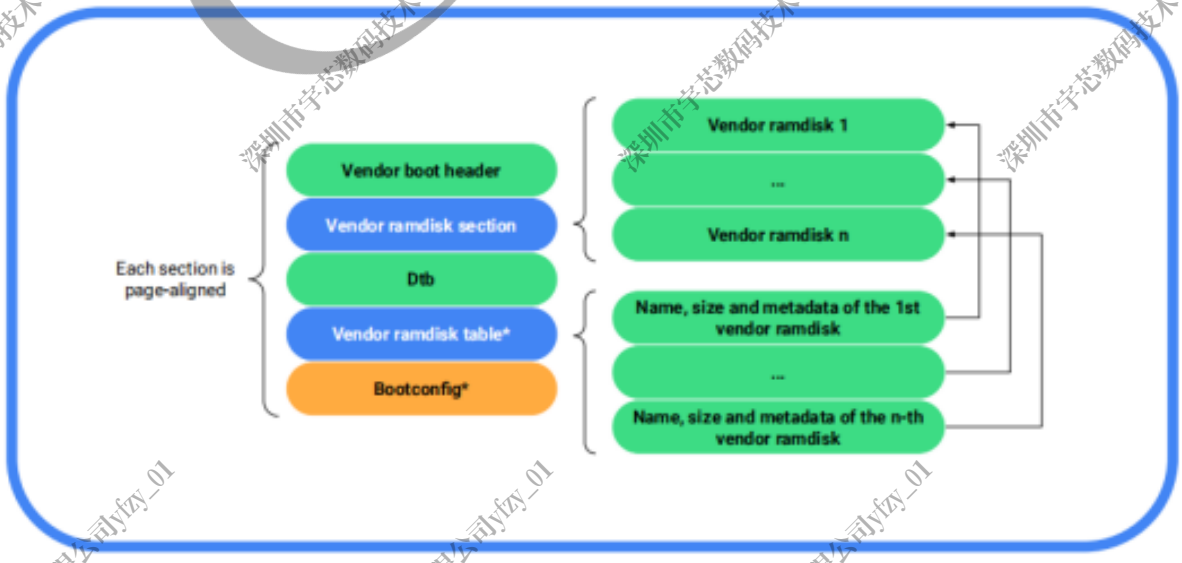

图 3-2: HeadV4 结构图

3.4 Bootconfig

3.4.1 Bootconfig 介绍

Bootconfig 是代替 cmdline 形式通过 androidboot.xxx=yyy 键值对格式传递给安卓的一种传参方式。它从 linux-5.10 开始引入，在 GKI2.0 中被强制使用。

Bootconfig 的特点如下所示：

- 为启动设备传递 key-value 格式的键值对
- 与 cmdline 相似，都是以键值对传递过去，但是与 cmdline 区别开来,cmdline 是通过在 fdt 添加 bootargs 节点传递给内核，bootconfig 通过地址形式传递给内核
- Bootconfig 在打包时候默认将数据放在 ramdisk 的最后
- 会在/proc/bootconfig 中呈现

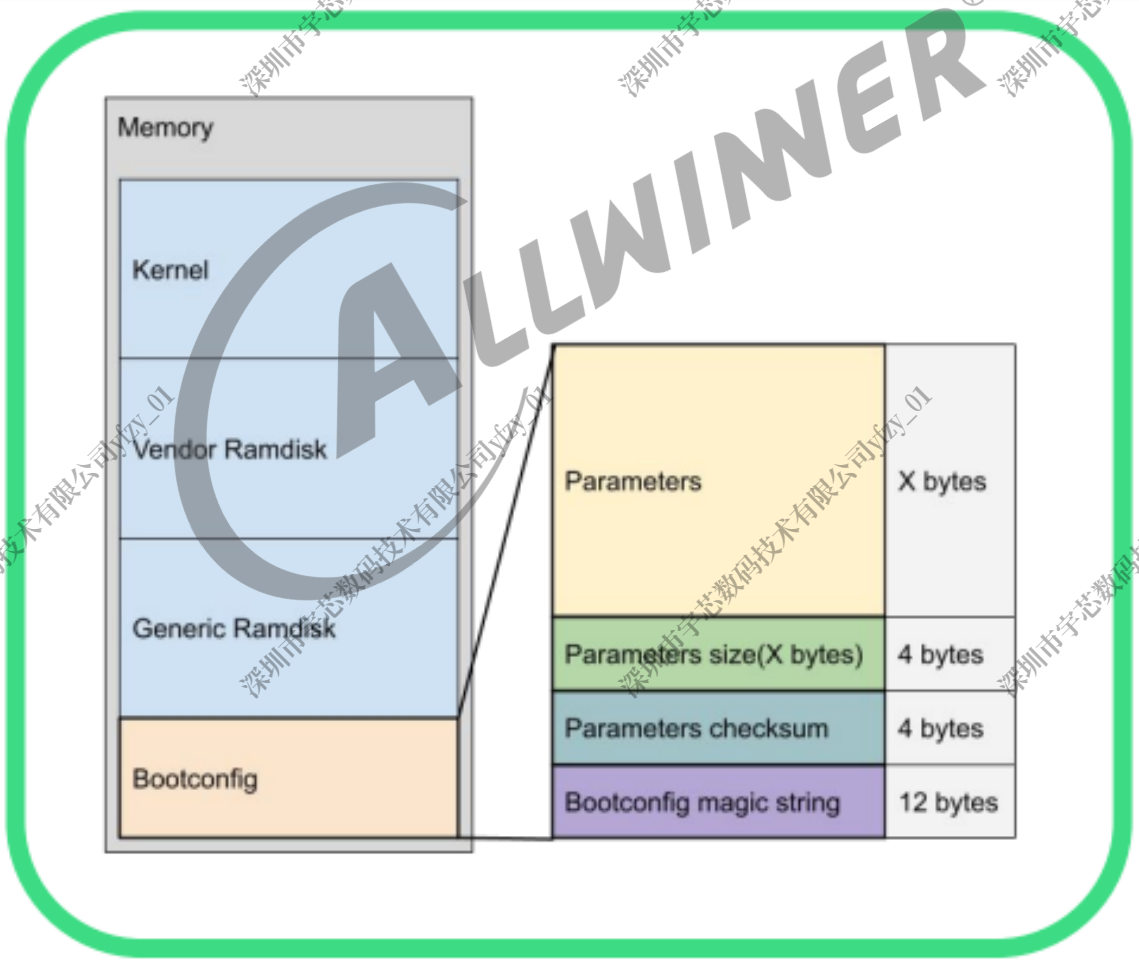

图 3-3: Bootconfig 组织结构

上图中的参数解释如下所示：

- Parameters Size: 传参个数
- Parameters Checksum：bootconfig 分区 checksum 和
- Magic:bootconfig 魔数

3.4.2 Bootconfig 的使用

1. 参数设置。Bootconfig 在编译打包阶段根据实际的情况，将需要传递给 kernel 的参数，打包进 vendor_boot 分区。在 Android13-linux5.15 + A133-P25 的环境下，传递的参数示例如下：

```
androidboot.dtbo_idx=0,1,2
androidboot.load_modules_parallel=true
androidboot.dynamic_partitions=true
androidboot.dynamic_partitions_retrofit=true
androidboot.selinux=enforcing
androidboot.dtbo_idx=0,1,2
```

如果想改变这些传递的参数，可以通过在构建脚本中进行修改：

```
# android13/longan/build/hook/pack/android3/pack_hook.sh

  INTERNAL_VENDOR_BOOTIMAGE_ARGS+=" --vendor_bootconfig $prebuiltpath/vendor-bootconfig.img"

  echo "androidboot.dynamic_partitions=true"        > $prebuiltpath/vendor-bootconfig.img
  echo "androidboot.dynamic_partitions_retrofit=true" >> $prebuiltpath/vendor-bootconfig.img
  echo "androidboot.selinux=enforcing"              >> $prebuiltpath/vendor-bootconfig.img
  echo "androidboot.dtbo_idx=0,1,2"                 >> $prebuiltpath/vendor-bootconfig.img

  rm -rf $prebuiltpath/vendor_boot.img
  ${MKBOOTIMG} \
    ${INTERNAL_VENDOR_BOOTIMAGE_ARGS} \
    --vendor_cmdline "${VENDOR_CMDLINE}" \
    ${BOARD_MKBOOTIMG_ARGS} \
    --vendor_ramdisk ${INTERNAL_VENDOR_RAMDISK_TARGET} \
    ${INTERNAL_VENDOR_RAMDISK_FRAGMENT_ARGS} \
    --vendor_boot $prebuiltpath/vendor_boot.img
```

同样你可以通过直接调用 mkbootimg 打包工具，再次修改传参的方式完成同样的功能。

2. 启动加载。然后在启动阶段，uboot 从 verdor 分区和 cmdline 中获取数据，最终放到 ramdisk 的后面。

如果你想确认你的固件中是否真正包含了此部分内容，则可以通过 uppack_bootimg 工具查看 vendor_boot.img 镜像中 vendor bootconfig 的大小，示例如下：

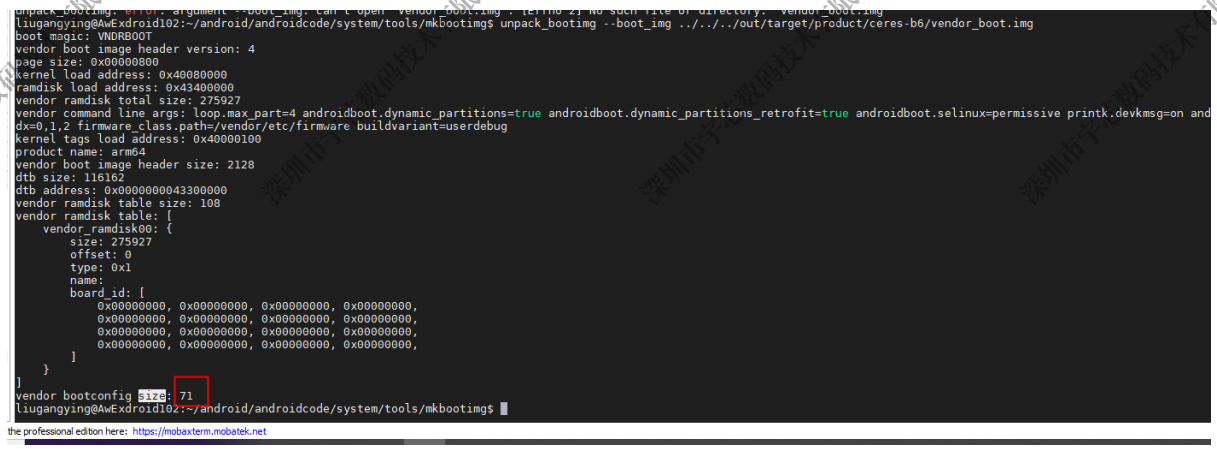

图 3-4: 查看 bootconfig 是否打包进去

3.5 模块驱动开发

Google GKI 要求 SoC 板级驱动做到一下几点：

- 模块 ko 化；
- 与框架层解耦，能成功在 Google 通用内核中运行。
- 提交初版驱动时，原则上只添加源文件，不修改内核原生代码；

模块 ko 化的方式就不在这里讲了，遇到问题请自行 google 解决。下面介绍驱动开发中需要注意的地方。

3.5.1 增加驱动版本号管理

Linux-5.4 及其之后的版本内核要求版本号管理，各个模块都要有自己的版本号记录，最好能相应有的 version 节点。

3.5.2 不使用 sys_config

Linux-5.4 及其之后的版本内核需要完全剥离 sys_config.fex，设备驱动配置一律放在 dts 中。

3.5.3 可使用 MODULE 变量

当模块被编译为 ko 时，MODULE 变量为 true，built-in 时为 false。所以可以使用 #ifdef MODULE 和 #ifndef MODULE 来进行 ko 和 built-in 的区分。

3.5.4 不使用 #ifdef 来判断配置是否打开

Linux-5.4 及其之后的版本内核驱动代码中不能使用 #ifdef 或者 #if define 来判断 menuconfig 配置是否打开，**必须使用：#if IS_ENABLED()。**

比如模块 AAA 编译为 ko，对应的 CONFIG_AAA=m，驱动代码中不能用 #ifdef CONFIG_AAA 来判断，而必须使用 #if IS_ENABLED(CONFIG_AAA)，因为用 #ifdef 无法判断编译为 ko 的配置，会导致判断出错。

所以，在驱动代码中，原则上编译为 KO 的模块必须使用：#if IS_ENABLED()，而 built-in 的模块可以使用 #ifdef。但是，为了避免疏忽，建议统一使用 #if IS_ENABLED()。

3.5.5 不使用解析 cmdline 参数的接口

模块驱动编译为 ko，驱动代码中不能使用 early_param__setup__setup_param 这类接口来解析 cmdline 参数，不然要么编译出错，要么得到的结果为 0。这类驱动通常是需要从 bootloader 中获取参数，建议把该参数放在 dts 中，由 uboot 更新 dts 对应的参数，最后在驱动中直接解析 dts。请模块负责人兼顾好 uboot 和内核驱动的适配，如果需要，请找 bootloader 同事支持。

3.5.6 不使用 OF_DECLARE 接口

当驱动被编译为 ko 时，OF_DECLARE 就相当于一个空操作，所以 TIMER_OF_DECLARE、CLK_OF_DECLARE、IRQCHIP_DECLARE 等均不允许在驱动为 ko 时使用。

google 官方提供的适配示例如下，使用 MODULE 变量来区分：

```
#ifdef MODULE
static int xxx_timer_probe(struct platform_device *pdev)
{
    struct device_node *np = pdev->dev.of_node;
    return xxx_timer_init(np);
}
static const struct of_device_id xxx_timer_match_table[] = {
    { .compatible = "XXXX,xxx-timer" },
    {}
};
MODULE_DEVICE_TABLE(of, xxx_timer_match_table);
```

```
static struct platform_driver xxx_timer_driver = {
    .probe      = xxx_timer_probe,
    .driver     = {
        .name   = "xxx-timer",
        .of_match_table = xxx_timer_match_table,
    },
};
module_platform_driver(xxx_timer_driver);
#else
TIMER_OF_DECLARE(xxx_timer, "XXX,xxx-timer", xxx_timer_init);
#endif
```

图 3-5: of declare

3.5.7 不使用 debugfs 接口

从 Android 11 开始不允许打开 CONFIG_DEBUG_FS，debugfs 类接口都不能调用，为了满足调试需求，可以用 CONFIG_DEBUG_FS 宏定义来做兼容。

3.5.8 不使用 vfs_read/vfs_write/kernel_read/kernel_write 接口

5.4 及其之后的版本中移除了 sdcardfs 驱动，连带 vfs_read/vfs_write/kernel_read/kernel_write 接口也取消了 export，所以在驱动代码中，不能调用这几个接口去操作文件节点。

3.5.9 模块 ko 加载过程中不允许失败

启动过程中模块 ko 加载失败，会导致设备会重启，所以要做好模块启动顺序及依赖关系的梳理，制定好启动加载流程，同时要求模块驱动不要有太强的模块依赖性。

3.5.10 配置 Android ko 加载列表

A133 B3 的 ko 加载列表文件为如下：

```
android/device/softwinner/{board}/configs/vendor_ramdisk.modules
```

只需在该文件中增加 ko 的名称，列表的先后顺序等于 ko 加载顺序。出现在该列表上的模块会在 android 启动的 frist init stage 阶段被加载，如果想要把某些模块加载延后，比如放在 late-fs 阶段加载，则可以在 init.xxx.rc 文件中配置。

3.6 GKI whitelist

3.6.1 whitelist 介绍

Google GKI 镜像中不可能包含所有 SoC 设备驱动所需要的代码，肯定会有很多 SoC 驱动 ko 没法直接在 GKI 上运行，那怎么办呢？为此，Google 提供了一份 gki_defconfig 和 GKI whitelist，让 SoC 厂商根据自身的需求，往 gki_defconfig 中增加模块配置，并在 whitelist 中添加自己需要的 symbol(函数、数据成员等)，Google 会把 gki_defconfig 中的配置项全部编译到 GKI 镜像中，并将这份 whitelist 对应的 symbol 全部 export 出来，供模块 ko 使用。同时，Google 也会利用这份 whitelist 来监测及维护 KMI 的 ABI 稳定性，ABI 稳定性是 GKI 得以实现的基础。

当然，Google 会有一份原始 GKI whitelist，有个别芯片厂商目前也提交了自己的 whitelist。如果这些 whitelist 加起来可以满足我们驱动的开发，那是再好不过了。如果不满足，我们就需要增加自己的 whitelist，并将其提交到 Google 内核分支中。

对于 arm64 来说，这份基础白名单为 abi_gki_aarch64_whitelist，同时各个厂商会增加属于自己白名单文件。Google 编译 GKI 内核时，会把所有内核顶层目录下的所有 whitelist 合并为一份，并且将其 export 出来。linux-5.15 与 linux-6.6 内核的组织形式稍有不同：

1. 现阶段 linux-5.15 内核的 whitelist 如下：

```
~/google/common/android$
├── abi_gki_aarch64
├── abi_gki_aarch64_amlogic
├── abi_gki_aarch64_db845c
├── abi_gki_aarch64_exynos
├── abi_gki_aarch64_exynosauto
├── abi_gki_aarch64_fips140
├── abi_gki_aarch64_galaxy
├── abi_gki_aarch64_honor
├── abi_gki_aarch64_imx
├── abi_gki_aarch64_mtk
├── abi_gki_aarch64_mtktv
├── abi_gki_aarch64_oplus
├── abi_gki_aarch64_pasa
├── abi_gki_aarch64_pixel
├── abi_gki_aarch64_qcom
├── abi_gki_aarch64_rtktv
├── abi_gki_aarch64_sony
├── abi_gki_aarch64_sunxi
├── abi_gki_aarch64_type_visibility
├── abi_gki_aarch64_unisoc
├── abi_gki_aarch64_virtual_device
├── abi_gki_aarch64_virtual_device_removed
├── abi_gki_aarch64_vivo
├── abi_gki_aarch64_xiaomi
├── abi_gki_aarch64.xml
├── abi_gki_modules_exports
├── abi_gki_modules_protected
├── gki_aarch64_fips140_modules
├── gki_aarch64_modules
└── gki_system_dlkm_modules
```

2. 现阶段 linux-6.6 内核的 whitelist 如下：

```
~/google/common/android$
├── abi_gki_aarch64
├── abi_gki_aarch64_fips140
├── abi_gki_aarch64_nothing
├── abi_gki_aarch64_tcl
├── abi_gki_aarch64_xiaomi
├── abi_gki_aarch64_amlogic
├── abi_gki_aarch64_galaxy
├── abi_gki_aarch64_oplus
├── abi_gki_aarch64_tuxera
└── abi_gki_protected_exports_aarch64
```

```
├── abi_gki_aarch64_asus
├── abi_gki_aarch64_honor
├── abi_gki_aarch64_pixel
├── abi_gki_aarch64_type_visibility
├── gki_aarch64_protected_modules
├── abi_gki_aarch64_db845c
├── abi_gki_aarch64_imx
├── abi_gki_aarch64_qcom
├── abi_gki_aarch64_unisoc
├── abi_gki_aarch64_exynos
├── abi_gki_aarch64_lenovo
├── abi_gki_aarch64.stg
├── abi_gki_aarch64_virtual_device
├── abi_gki_aarch64_exynosauto
├── abi_gki_aarch64_mtk
├── abi_gki_aarch64_sunxi
└── abi_gki_aarch64_vivo
```

3.6.2 AW 模块 whitelist 需求收集

3.6.2.1 白名单检查

在模块完成 ko 之后，可以利用 google 提供的工具及 GKI 对应的 vmlinux 文件，检测自己的模块 ko 所需要的接口是否有在 GKI 中 export 出来，具体步骤如下。

- 编译 google 的源码，具体步骤详见 “GKI 环境自编译” 章节
- 从 aosp 源码中拷贝用于白名单测试的文件：

```
cp google/build/abi/extract_symbols android13/longan/out/kernel/staging/lib/modules/5.15.41

cp google/out/android13-5.15/common/vmlinux android13/longan/out/kernel/staging/lib/modules/5.15.41
```

- 对所有编译得到的 ko 进行白名单检查（我们的 andriod 在编译完成之后，ko 文件均放在如下路径中）

```
# a.whitelist 为我们指定的一个存放已经导出的白名单的文件
android13/longan/out/kernel/staging/lib/modules/5.15.41$ ./extract_symbols --whitelist a.whitelist
```

- 如果你行对单个 ko 进行白名单的检测，则只需要将 ko 和白名单检测工具放置在同一个目录下，然后执行如上命令即可。

下面我们以一个实际的例子，阐述如何为我们的驱动 ko 文件新增所需的白名单。

```
Hss:~/gki/google$ ls
extract_symbols  sunxi-iommu.ko  vmlinux
Hss:~/gki/google$
Hss:~/gki/google$ ./extract_symbols --whitelist out.whitelist
Symbol __dynamic_dev_dbg required by sunxi-iommu.ko but not provided
Symbol __dynamic_pr_debug required by sunxi-iommu.ko but not provided
Symbol iommu_device_sysfs_add required by sunxi-iommu.ko but not provided
Symbol iommu_device_sysfs_remove required by sunxi-iommu.ko but not provided
Symbol iommu_group_alloc required by sunxi-iommu.ko but not provided
Symbol iommu_group_register_notifier required by sunxi-iommu.ko but not provided
Hss:~/gki/google$
Hss:~/gki/google$ ls
extract_symbols  out.whitelist  sunxi-iommu.ko  vmlinux
```

图 3-6: gki whitelsit detect

可以看到 google GKI 没有 export 出 sunxi-iommu.ko 所需要的所有接口，“not provided” 表示 GKI 没有 export 出该接口，但是 sunxi-iommu.ko 需要用到。

最直接的解决方法就是将其添加到 gki whitelist 中，让 google 编译 GKI 时把这些接口 export 出来。

当然我们也要先考虑是否一定需要调用这些接口，看是否能够优化。

其中 __dynamic_dev_dbg 和 __dynamic_pr_debug 可以忽略，因为我们打开了 CONFIG_DYNAMIC_DEBUG，而 google GKI 是没打开的，后续 GKI 测试时，我们也需要关闭该选项。类似的接口还有：mutex_destroy 等。

- 确认接口能否加入到 whitelist

当发现 google GKI 没有 export 出自己模块所需接口时，首先要看 google GKI 是否有编译这些接口所对应的代码，如果有，那我们可以考虑将其加入 whitelist；如果没有，我们优先考虑将其编译为 ko。如果发现无法将其编为 ko，则只能考虑给 google 提需求，但需慎重考虑，因为 google 对与 gki defconfig 的提交很严格，毕竟涉及到 ABI 稳定性问题。

可以通过 google 编译 GKI 所用到的.config，也就是上面提供的 gki_defconfig，来确认是否能直接加入 whitelist。

还是以 sunxi_iommu.ko 为例子，上面看到它还需要 export 的接口有：iommu_device_sysfs_add、iommu_device_sysfs_remove 等，找到这些接口对应源文件的编译选项：CONFIG_IOMMU_API，在 google gki_defconfig 中找：

```
Hss:~/gki/google$ cat gki_defconfig | grep CONFIG_IOMMU_API
CONFIG_IOMMU_API=y
```

图 3-7: gki whitelsit detect

能找到并且为 y，说明可以直接加入 whitelist，

之所以推荐用 gki_defconfig 方式进行确认，主要想让大家弄清楚模块 CONFIG 的依赖关系，虽然可能会稍微费点时间。当然，也可以通过直接解析 vmlinux 进行快速确认：

```
Hss:~/gki/google$ nm --defined-only vmlinux | grep  iommu_device_sysfs_add
ffffffc01075b210 T iommu_device_sysfs_add
ffffffc01162ed44 d __ksym_marker_iommu_device_sysfs_add
```

图 3-8: gki whitelsit detect

上图可以看出 iommu_device_sysfs_add 已经被编译进 GKI 镜像，但是没有被 export 出来，因为编译时被优化了.

下面是我把 iommu_device_sysfs_add 加入 gki whitelist 之后，在 google 内核环境下编译出 vmlinux 的运行结果：

```
Hss:~/gki/google$ nm --defined-only vmlinux | grep  iommu_device_sysfs_add
0000000006607bd95 A __crc_iommu_device_sysfs_add
ffffffc01075b25c T iommu_device_sysfs_add
ffffffc010d75838 t iommu_device_sysfs_add.cfi_jt
ffffffc0116510f7 r __kstrtab_iommu_device_sysfs_add
ffffffc01162f044 d __ksym_marker_iommu_device_sysfs_add
ffffffc011641038 r __ksymtab_iommu_device_sysfs_add
```

图 3-9: gki whitelsit detect

kstrtab、ksymtab 等字段表示已经 export 出来了。

3.6.2.2 白名单添加

whitelist 添加格式：

```
[abi_whitelist]

# required by sunxi_iommu.ko
  iommu_device_sysfs_add
  iommu_device_sysfs_remove
  iommu_group_alloc
  iommu_group_register_notifier
```

模块负责人可以自己新建一个 whitelist，可以命名为：sunxi_xxx 模块 _whitelist，按照上面的格式添加，并发给 AW GKI 负责人。

4 GKI 常见问题

4.1 使用 Google 官方发布的 GKI 镜像

当下游厂商需要做内核升级或者同步最新 Google 发布的 GKI 镜像时，可以直接在 Google 官方发布网站上获取：

Android13-5.15：

```
https://source.android.com/docs/core/architecture/kernel/gki-android13-5_15-release-builds#launch-releases
```

Android15-6.6：

```
https://source.android.com/docs/core/architecture/kernel/gki-android15-6_6-release-builds#launch-releases
```

需要注意的是：

1. 不要使用 boot-debug 镜像，仅适用于 GKI 之前的设备；
2. boot-5.15.img 为未压缩的 boot.img（AW 只支持未压缩的 boot.img）；
3. boot-5.15-gz、boot-5.15-lz4 分别为 gz、lz4 压缩的 boot.img（AW 不支持压缩 boot.img，不能使用）；

4.2 使用自编译 GKI 镜像 (仅供调试用)

4.2.1 GKI 环境自编译

注：对于 Linux 内核版本高于 5.15 的系统，无需自行编译，因此下述编译方法仅适用于内核版本为 5.15 或更低的系统。

- 下载 Google 源码环境
- 在 google 源码顶层目录下运行以下命令：

```
google$ BUILD_BOOT_IMG=1 SKIP_VENDOR_BOOT=1 KERNEL_BINARY=Image GKI_RAMDISK_PREBUILT_BINARY=gki-ramdisk.img BUILD_CONFIG=common/build.config.gki.aarch64 build/build.sh
```

- 将 ramdisk 添加进 boot.img 文件：

```
google$ ./tools/mkbootimg/mkbootimg.py --header_version 3 --kernel out/android13-5.15/dist/Image --ramdisk gk-
    ramdisk.img --output out/android13-5.15/dist/boot.img
```

- 注意：编译过程中在LTO vmlinux.o阶段可能会卡主一段时间（大概十分钟），属于正常现场，可以通过将 gki_defconfig 去掉如下配置的方式进行临时解决。

```
CONFIG_SHADOW_CALL_STACK
CONFIG_LTO_CLANG_FULL
CONFIG_CFI_CLANG
```

- 编译过程需要 ramdisk 文件，这个文件的获取见如下的网站：

```
https://ci.android.com/builds/branches/aosp-master-throttled/grid?
```

```
https://ci.android.com/builds/submitted/8549645/aosp_arm64-eng/latest
```

- 编译完成后，boot.img 存放在下面的目录中，这里说的 boot.img 其实就是我们说的 GKI 镜像

```
google/out/android13-5.15/dist/boot.img
```

- AOSP 的内核编译详细介绍请见：https://source.android.com/setup/build/building-kernels

4.2.2 GKI 替换测试

编译得到 GKI 镜像后，通过如下步骤替换镜像中已有的 GKI 镜像：

- 在设备中打开 OEM unlocking 以解锁设备：Developer options -> OEM unlocking，或者通过命令的方式进行解锁：
- 使用 fastboot 烧写 GKI boot.img 到我们的机器，在机器上执行如下命令：

```
adb reboot bootloader
fastboot oem unlock
fastboot flash boot boot.img
fastboot reboot
```

注意：自编译的 GKI 镜像仅用于调试使用，不可用于实际的发货。

4.3 GKI 补丁提交

GKI 补丁提交包括如下文件，如果有相关需求请联系 AW。

- gki whitelist：android13/longan/kernel/linux-5.15/android/abi_gki_aarch64_sunxi
- gki_defconfig：android13/longan/kernel/linux-5.15/arch/arm64/configs/gki_defconfig
- SoC 驱动代码的提交：android13-a133/longan/bsp/*

4.4 GKI 兼容性问题 1：symbol 校验失败

- 问题现象：第一个 ko 就加载失败了，系统不断重启。log 如下所示：

```
[   2.553495] ccu_sunxi_ng: disagrees about version of symbol module_layout
[   2.560111] init: Failed to insmod '/lib/modules/ccu-sunxi-ng.ko' with args ''
```

- 问题原因：本地环境编译 ko 时生成的 symbol magic 与 GKI 环境编译的不一致，导致启动校验失败。

```
android13/longan/out/kernel/build/Module.symvers # longan下的文件路径
androidT-google/out/android13-5.15/common/Module.symvers # aosp环境下的文件路径
```

- 解决办法：检查 defconfig 配置文件是否打开了内核公共模块，除了编译成 m 的 ko 模块以外，其余的内核配置需要和 GKI defconfig 保持一致

```
# GKI defconfig 文件路径
android13/longan/kernel/linux-5.15/arch/arm64/configs/gki_defconfig

# AW android defconfig 文件路径
android13/longan/device/config/chips/a133/configs/default/linux-5.15/android13_arm64_defconfig
```

4.5 GKI 兼容性问题 2-whitelist 缺失导致 ko 加载失败

- 问题现象：ko 加载过程中，某个非第一个 ko 文件加载失败了，系统不断重启；

```
[   5.336277] lan78xx: Unknown symbol device_set_wakeup_enable (err -2)
[   5.343984] lan78xx: Unknown symbol phy_ethtool_ksettings_get (err -2)
[   5.351439] lan78xx: Unknown symbol genphy_read_status (err -2)
[   5.358199] lan78xx: Unknown symbol mdiobus_read (err -2)
[   5.364532] lan78xx: Unknown symbol phy_ethtool_set_wol (err -2)
```

- 问题原因：模块 ko 在加载过程中需要的一些接口未被导出，导致 ko 加载时没法使用这些 symbol，从而加载失败。
- 解决办法：请联系 AW，根据实际的进行解决。

著作权声明

本文档及内容受著作权法保护，其著作权由珠海全志科技股份有限公司（“全志”）拥有并保留一切权利。

本文档是全志的原创作品和版权财产，未经全志书面许可，任何单位和个人不得擅自摘抄、复制、修改、发表或传播本文档内容的部分或全部，且不得以任何形式传播。

商标声明





（不完全列举）均为珠海全志科技股份有限公司的商标或者注册商标。在本文档描述的产品中出现的其它商标，产品名称，和服务名称，均由其各自所有人拥有。

免责声明

您购买的产品、服务或特性应受您与珠海全志科技股份有限公司（“全志”）之间签署的商业合同和条款的约束。本文档中描述的全部或部分产品、服务或特性可能不在您所购买或使用的范围内。使用前请认真阅读合同条款和相关说明，并严格遵循本文档的使用说明。您将自行承担任何不当使用行为（包括但不限于如超压，超频，超温使用）造成的不利后果，全志概不负责。

本文档作为使用指导仅供参考。由于产品版本升级或其他原因，本文档内容有可能修改，如有变更，恕不另行通知。全志尽全力在本文档中提供准确的信息，但并不确保内容完全没有错误，因使用本文档而发生损害（包括但不限于间接的、偶然的、特殊的损失）或发生侵犯第三方权利事件，全志概不负责。本文档中的所有陈述、信息和建议并不构成任何明示或暗示的保证或承诺。

本文档未以明示或暗示或其他方式授予全志的任何专利或知识产权。在您实施方案或使用产品的过程中，可能需要获得第三方的权利许可。请您自行向第三方权利人获取相关的许可。全志不承担也不代为支付任何关于获取第三方许可的许可费或版税（专利税）。全志不对您所使用的第三方许可技术做出任何保证、赔偿或承担其他义务。