Application Note

SITARA AM62Ax 系列处理器快速启动方法

章勇，华腾 *Central FAE*

摘要

在车载和工业应用中，很多情况下都需要上电后快速启动。例如在车载电子后视镜（CMS）/流媒体后视镜中，通常要求在汽车上电后几秒内完成从上电到显示的全流程。在使用默认镜像的情况下，完整功能的启动流程往往耗时 20s 以上。启动时间的优化涉及到启动流程的优化，内核裁减和启动介质的选择等。本文将以 AM62A 的 CMS 应用为例介绍快速启动方法，在本文的测试中，从上电到应用启动耗时 2.5s，能够满足绝大多数需要快速启动的应用场景。

内容

1 简介....2
2 启动流程....2
2.1 SPL 启动流程....4
2.2 SPL Falcon 模式....9
2.3 从 SPL 替换为 SBL....11
3 启动介质....12
4 操作系统启动优化....14
4.1 裁剪操作系统....14
4.2 减少 log 输出....16
5 用户空间优化....17
5.1 使用 systemd 自定义启动服务....17
6 使用 init 脚本最小化启动服务....19
7 总结....20
8 参考文献....20

插图清单

图 2-1. AM62Ax 系统框图....3
图 2-2. SPL 启动流程....4
图 2-3. BOOT 分区文件结构....4
图 2-4. tiboot3 镜像打包....5
图 2-5. tispl 镜像打包....7
图 2-6. SBL 启动流程....11
图 4-1. Menuconfig 界面....15
图 4-2. 在 Menuconfig 界面中配置蓝牙....15
图 4-3. 保存 Menuconfig....16
图 5-1. systemd 启动依赖项....17

商标

1 简介

SITARA AM62Ax 系列的处理器包括了 AM62A3、AM62A7 处理器。TI 为该芯片提供了多种开箱即用的镜像文件，用户在使用烧录工具快速烧录模型镜像文件后即可进行开发调试。然而在使用默认镜像开发完用户应用，将处理器用于实际产品时，却往往会遇到开机后应用启动时间过长的问题，在使用默认镜像以及使用 SD 卡作为启动介质的情况下，单次的启动时间往往高达二十秒以上，漫长的启动等待对于终端用户而言是很难接受的。为了满足快速启动的需求，我们需要修改启动流程，更换启动介质，裁减操作系统等多个步骤来实现。

2 启动流程

在了解启动流程之前需要大致了解一下处理器的大致组成。TI 的处理器多为多核异构的架构，即在一个处理器中会出现多个不同架构的计算单元同时存在且并行处理的场景，而一些计算单元组合在一起构成一个域。以 AM62A7 为例，如下图所示，Application Cores 中有最多 4 颗 A53 核，一般运行 Linux 或者 QNX 等高阶操作系统，MCU Channel with FFI 是用于功能安全的独立 MCU R5F，一般运行 RTOS 或者 AUTOSAR，Device Management 域中有一颗 DM R5F 核，负责启动阶段的部分工作，在启动完成后负责一些片上资源的管理，一般会运行 RTOS 操作系统，Deep Learning Accelerator 中包含一颗 C7xMMA 核，运行 RTOS 操作系统，负责 AI 推理加速。

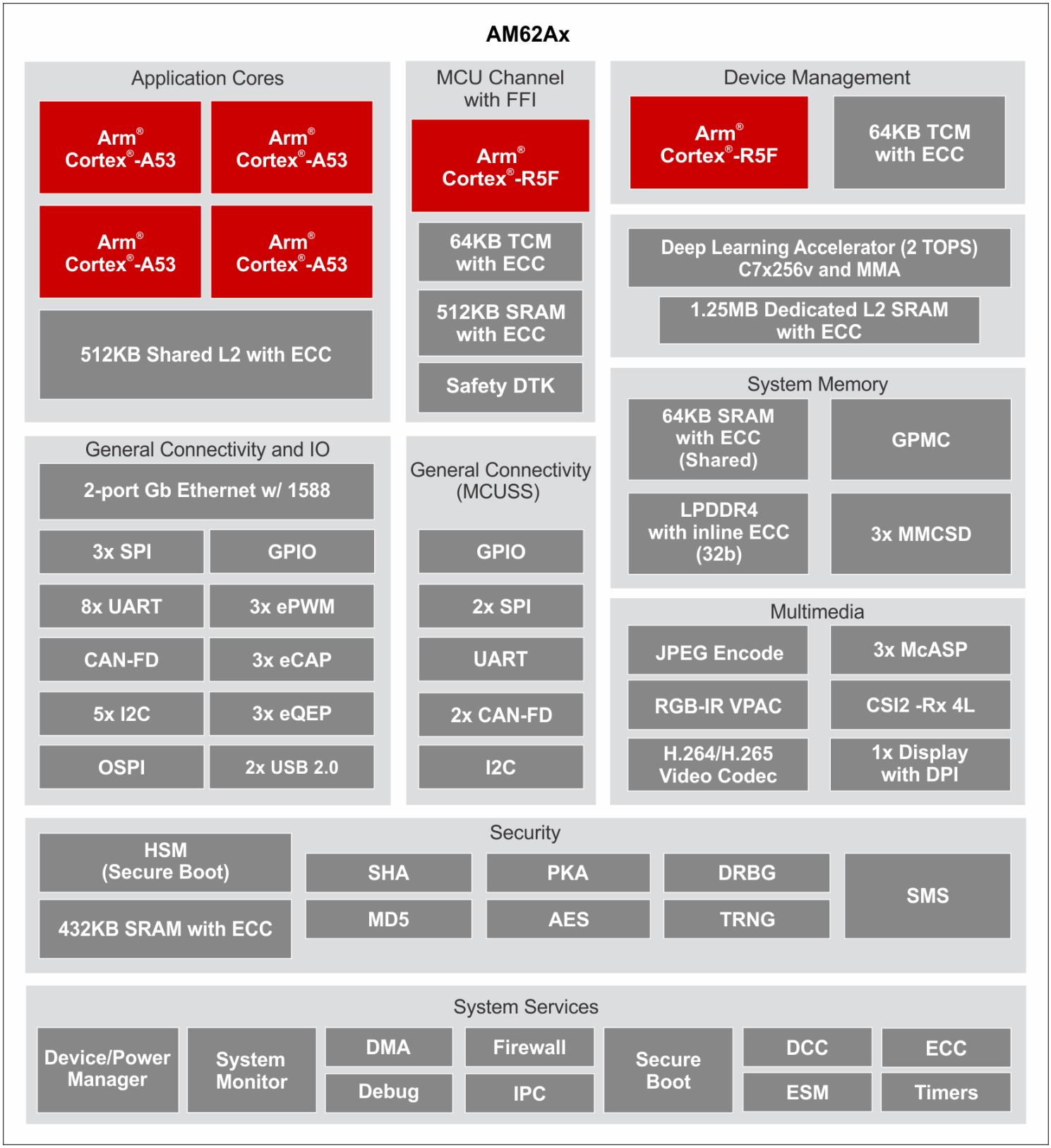

图 2-1. AM62Ax 系统框图

SOC 的完整启动需要给每一个核心都加载各自的代码。TI 提供的启动流程总体上可以分为 SPL 和 SBL 两种，在这两种模式中还有是否经过 U-boot 的细分启动流程。接下来会详细介绍两种启动流程、各种启动过程中可以进行的不同配置以及不同配置带来的影响。

2.1 SPL 启动流程

如图 2 与图 3 所示，SPL 启动流程中，首先 PMIC 上电后 DM R5F 启动 ROM 代码，ROM 代码加载与启动 R5 SPL 代码，R5 SPL 代码编译打包被命名为 tiboot3.bin；紧接着，DM R5F 将加载 tispl.bin，A 核开始执行 tispl.bin 的代码，即 TF-A+OPTEE+A53 SPL 启动过程；接着 A 核加载 u-boot.img，启动 U-Boot 操作系统，进入 U-Boot 启动阶段，U-Boot 操作系统会读取 uEnv.txt 中对于设备树的额外配置；U-Boot 接下来会从文件系统中读取 Linux 内核程序以及 C7x，MCU R5F 等小核的程序，启动 Linux，同时启动小核程序；最后 Linux 内核启动完成，进入用户态服务的启动，最终完成完整的启动。详细的启动流程可以参考 SDK 的 User Guide。

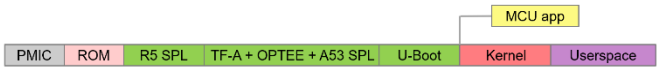

图 2-2. SPL 启动流程

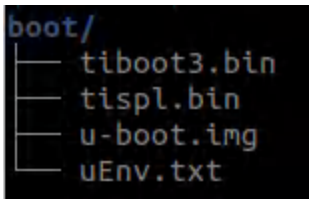

图 2-3. BOOT 分区文件结构

该启动流程是默认状态下的启动过程，可以参考 SD 卡制卡流程制作启动卡，使用 SD 卡启动模式测试。ROM 代码在芯片出厂时就已经固化在芯片里，SDK 不提供 ROM 代码。接下来将结合 SDK 描述各个流程具体的行为。

2.1.1 R5 SPL

R5 SPL 首先从 ROM 中读取启动信息，启动信息是一些基本的芯片属性；接着初始化 R5F 核本身，查找并初始化串口，以便于输出一些启动 log，加载 System firmware，最后初始化 DDR。具体代码详见链接，代码也可以通过安装 LINUX SDK 获得。

在 LINUX SDK 中，通过编译 U-boot 可以编译获得 R5 SPL 对应的镜像文件 tiboot3.bin。TI 的 U-Boot 使用设备树作为启动的最小硬件描述，确保基础外设可用，使用 am62ax_evm_r5_defconfig 进行配置，采用 Binman 作为固件镜像的打包工具，Binman 通过设备树统一配置管理，k3-am62a-sk-binman.dtsi 描述 tiboot3 镜像布局，Binman 按照镜像布局打包镜像，将多个组件集合在一个或少数几个文件中。修改 R5 SPL 时要注意对应的设备树和 defconfig。

Binman 按照对应的设备树打包完的 tiboot3.bin 镜像文件如下图所示，整个包有一个 X.509 证书作为包头，其下有对应的 R5 u-boot-spl.bin 作为实际的 R5 SPL 代码，除此以外，包中还有 sys-firmware（又名 TIFS），TIFS 被加载到片内 SMS 子系统中 M4F_0 小核。包中还包含了最小的设备树，用于描述启动所需的最小设备配置。

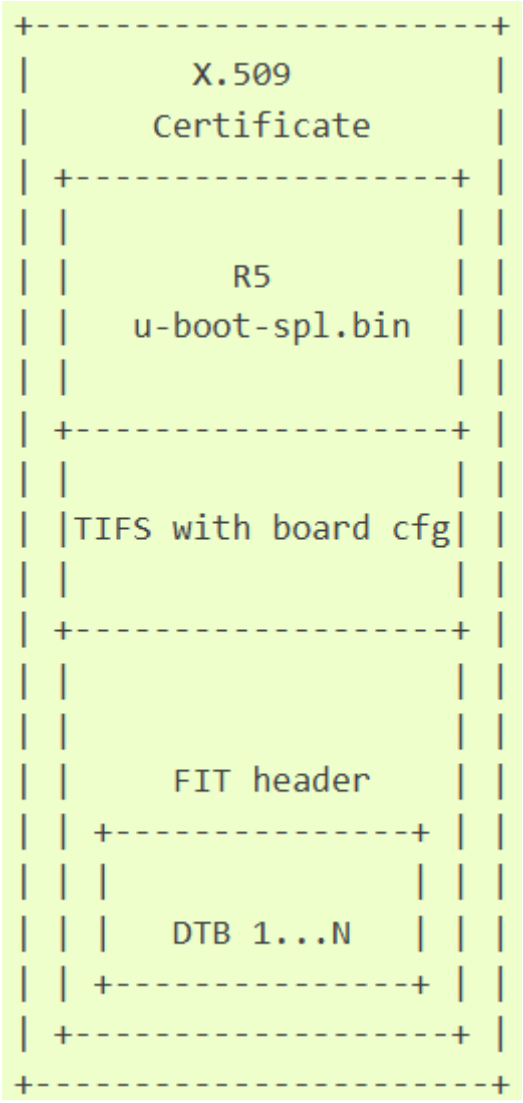

图 2-4. tiboot3 镜像打包

在编译时，设置环境变量 BINMAN_VERBOSE=3 以及 BINMAN_DEBUG=1 可以看到对应 Binman 打包镜像的过程，以下 Binman log 中显示的打包项与设备树中描述的一致。

```
Pack completed after 1 pass(es)
Writing image to './custMpk.pem'
Wrote 0xcab bytes
Pack completed after 1 pass(es)
Writing image to './ti-degenerate-key.pem'
Wrote 0x21a bytes
Pack completed after 1 pass(es)
Writing image to './board-cfg.bin'
Wrote 0x1d bytes
Pack completed after 1 pass(es)
Writing image to './pm-cfg.bin'
Wrote 0x2 bytes
Pack completed after 1 pass(es)
Writing image to './rm-cfg.bin'
Wrote 0x596 bytes
Pack completed after 1 pass(es)
Writing image to './sec-cfg.bin'
Wrote 0x15d bytes
Pack completed after 1 pass(es)
Writing image to './combined-tifs-cfg.bin'
Wrote 0x674 bytes
Pack completed after 1 pass(es)
```

```
Writing image to './combined-dm-cfg.bin'
Wrote 0x5aa bytes
Pack completed after 1 pass(es)
Writing image to './combined-sysfw-cfg.bin'
Wrote 0x734 bytes
Pack completed after 1 pass(es)
Writing image to './tiboot3-am62ax-hs-evm.bin'
Wrote 0x50981 bytes
Pack completed after 1 pass(es)
Writing image to './tiboot3-am62ax-hs-fs-evm.bin'
Wrote 0x50981 bytes
Pack completed after 1 pass(es)
Writing image to './tiboot3-am62ax-gp-evm.bin'
Wrote 0x4ff64 bytes
```

在启动过程中，R5 SPL 对应 log 如下：

```
[2025-08-05 19:53:27.057] U-Boot SPL 2024.04-ti-g29d0c23d67ee (Nov 29 2024 - 11:41:54 +0000)
[2025-08-05 19:53:27.057] SYSFW ABI: 4.0 (firmware rev 0x000a '10.1.8--v10.01.08 (Fiery Fox)')
[2025-08-05 19:53:27.120] SPL initial stack usage: 13568 bytes
[2025-08-05 19:53:27.120] Trying to boot from MMC2
[2025-08-05 19:53:27.265] Authentication passed
[2025-08-05 19:53:27.265] Authentication passed
[2025-08-05 19:53:27.281] Authentication passed
[2025-08-05 19:53:27.409] Authentication passed
[2025-08-05 19:53:27.409] Authentication passed
[2025-08-05 19:53:27.425] Starting ATF on ARM64 core...
```

R5 SPL 流程耗时短，这部分的耗时主要来自于从启动介质往 SOC 内部 RAM 的数据搬运以及一些资源的初始化。资源初始化的部分几乎不可省略，因此提高这部分的启动速度关键在于使用快速的启动介质。

2.1.2 TF-A+OPTEE+A53 SPL

TF-A+OPTEE+A53 SPL 以下简称 A53 SPL。如名称所示，这个流程会加载运行 ARM Trusted Firmware, OPTEE 和 A53 SPL。这部分镜像都打包在 tislp.bin 中，包中镜像的加载位置等配置描述位于设备树文件 k3-binman.dtsi 中，tispl.bin 的结构如下图所示，包括了 ATF，OPTEE，DM 镜像，SPL 镜像以及一些设备树文件。

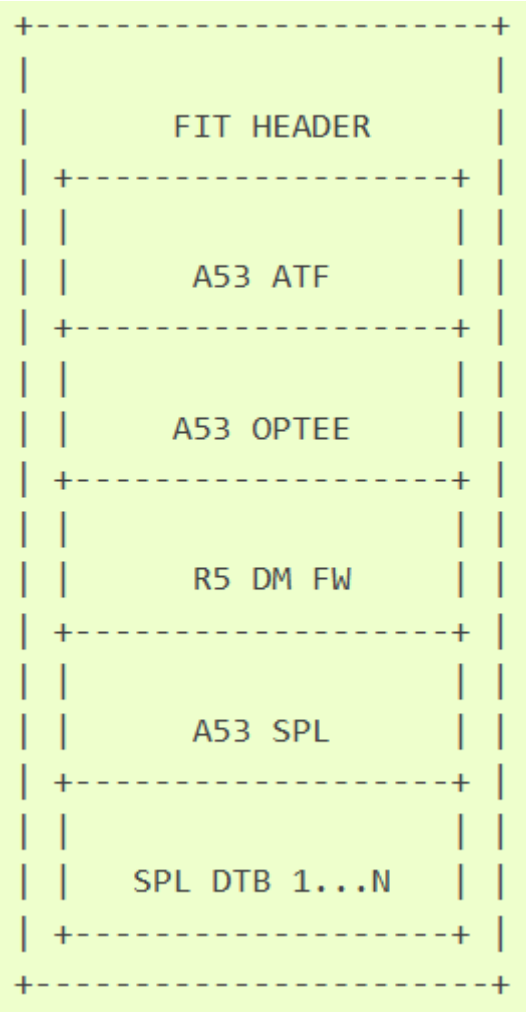

图 2-5. tispl 镜像打包

以下是对应的打包 log，打包的过程中会将 U-Boot.img 一起进行打包。

```
Pack completed after 1 pass(es)
Writing image to './custMpk.pem'
Wrote 0xcab bytes
Pack completed after 1 pass(es)
Writing image to './ti-degenerate-key.pem'
Wrote 0x21a bytes
Pack completed after 1 pass(es)
Writing image to './ti-dm.bin'
Wrote 0x8edf0 bytes
Pack completed after 1 pass(es)
Writing image to './tifsstub.bin_hs'
Wrote 0x6d1 bytes
Pack completed after 1 pass(es)
Writing image to './tifsstub.bin_fs'
Wrote 0x0 bytes
Pack completed after 1 pass(es)
Writing image to './tifsstub.bin_gp'
Wrote 0x3ca bytes
Pack completed after 1 pass(es)
Writing image to './tispl.bin'
Wrote 0x165bab bytes
Pack completed after 1 pass(es)
Writing image to './u-boot.img'
Wrote 0x11ff63 bytes
Pack completed after 1 pass(es)
Writing image to './tispl.bin_unsigned'
Wrote 0x163a77 bytes
Pack completed after 1 pass(es)
```

```
Writing image to './u-boot.img_unsigned'
Wrote 0x11f21b bytes
```

A53 SPL 对应的启动 log 如下，这部分的启动耗时约 1.4s，主要的耗时仍然在于搬运加载对应的镜像文件，与启动介质息息相关。

```
 [2025-08-05 19:53:27.425]
[2025-08-05 19:53:27.425] NOTICE: BL31: v2.11.0(release):v2.11.0-906-g58b25570c9-dirty
[2025-08-05 19:53:27.425] NOTICE: BL31: Built : 04:20:32, Nov 1 2024
[2025-08-05 19:53:27.681]
[2025-08-05 19:53:27.681] U-Boot SPL 2024.04-ti-g29d0c23d67ee (Nov 29 2024 - 11:41:54 +0000)
[2025-08-05 19:53:27.681] SYSFW ABI: 4.0 (firmware rev 0x000a '10.1.8--v10.01.08 (Fiery Fox)')
[2025-08-05 19:53:27.696] Trying to boot from MMC2
 [2025-08-05 19:53:28.193] Authentication passed

 [2025-08-05 19:53:28.833] Authentication passed
```

在这部分的启动过程中，还会进行两个镜像的验签，对应启动 log 中"Authentication passed"，对应代码 ti_secure_image_post_process()，验签只针对 DM 镜像和设备树文件，ATF 和 OPTEE 并不验签，验签耗时较短，对整体启动速度影响不大。

2.1.3 U-Boot

U-Boot 阶段运行 U-Boot 操作系统，U-Boot 操作系统 log 如下。U-Boot 启动阶段会等待 3s 用户在串口上的输入，决定是否进行 U-Boot 的命令行，如果进入命令行，可以输入 U-Boot 相关命令做一些早期操作，比如修改设备树，给 EMMC 进行分区和镜像载入等。U-Boot 阶段还会更改启动介质的速率，会根据启动介质支持的传输速率重新配置电压和相关时钟，也只有在这个阶段才能将 SD 卡等介质设置为高速模式，相应的镜像加载会在这个阶段进行加速。在修改为高速模式之前，启动介质通常运行在低速模式下，镜像加载速度较慢，相关可以参考 R5 SPL 与 A53 SPL 阶段的镜像加载。

```
[2025-08-05 19:53:30.032]
[2025-08-05 19:53:30.032]
[2025-08-05 19:53:30.032] U-Boot 2024.04-ti-g29d0c23d67ee (Nov 29 2024 - 11:41:54 +0000)
[2025-08-05 19:53:30.048]
[2025-08-05 19:53:30.048] SoC: AM62AX SR1.0 HS-FS
[2025-08-05 19:53:30.048] Model: Texas Instruments AM62A7 SK
[2025-08-05 19:53:30.048] DRAM: 2 GiB (effective 4 GiB)
[2025-08-05 19:53:30.544] Core: 87 devices, 30 uclasses, devicetree: separate
[2025-08-05 19:53:30.545] MMC: mmc@fa10000: 0, mmc@fa00000: 1
[2025-08-05 19:53:30.560] Loading Environment from nowhere... OK
[2025-08-05 19:53:30.576] In: serial@2800000
[2025-08-05 19:53:30.576] Out: serial@2800000
[2025-08-05 19:53:30.576] Err: serial@2800000
[2025-08-05 19:53:30.576] Net: eth0: ethernet@8000000port@1
 [2025-08-05 19:53:30.608] Hit any key to stop autoboot: 0
 [2025-08-05 19:53:32.784] switch to partitions #0, OK
[2025-08-05 19:53:32.784] mmc1 is current device
[2025-08-05 19:53:33.056] SD/MMC found on device 1
[2025-08-05 19:53:33.104] 653 bytes read in 35 ms (17.6 KiB/s)
[2025-08-05 19:53:33.104] Loaded env from uEnv.txt
[2025-08-05 19:53:33.104] Importing environment from mmc1 ...
[2025-08-05 19:53:33.168] 53172 bytes read in 51 ms (1017.6 KiB/s)
[2025-08-05 19:53:33.168] Warning: Did not detect image signing certificate. Skipping
authentication to prevent boot failure. This wills
[2025-08-05 19:53:33.184] Load Remote Processor 0 with data@addr=0x82000000 53172 bytes: Success!
[2025-08-05 19:53:33.360] 10719312 bytes read in 159 ms (64.3 MiB/s)
[2025-08-05 19:53:33.360] Warning: Did not detect image signing certificate. Skipping
authentication to prevent boot failure. This wills
[2025-08-05 19:53:33.376] Load Remote Processor 2 with data@addr=0x82000000 10719312 bytes: Success!
[2025-08-05 19:53:33.648] 21029376 bytes read in 256 ms (78.3 MiB/s)
[2025-08-05 19:53:33.696] 61175 bytes read in 37 ms (1.6 MiB/s)
[2025-08-05 19:53:33.696] Working FDT set to 88000000
[2025-08-05 19:53:33.696] ## Flattened Device Tree blob at 88000000
[2025-08-05 19:53:33.712] Booting using the fdt blob at 0x88000000
[2025-08-05 19:53:33.712] Working FDT set to 88000000
[2025-08-05 19:53:33.744] Loading Device Tree to 000000008feee000, end 000000008fffffff ... OK
[2025-08-05 19:53:33.744] Working FDT set to 8feee000
```

U-Boot 阶段还会读取相应设备树，加载小核镜像，启动 C7 等其他小核。U-Boot 阶段耗时较长，通常在 3s 以上，这个阶段的主要耗时在于 u-boot.img 的镜像加载和等待用户键入的过程，在载入 u-boot.img 时，启动介质还

没有切换为高速模式，耗时较长。U-Boot 阶段主要能加速的部分在于跳过 3s 的等待，然而即使跳过了 3s 等待，U-boot 阶段耗时仍然需要 1s 以上。

U-Boot 阶段结束后，开始 Linux 的启动。

2.1.4 Linux 以及用户态

Linux 启动流程较为常规，Linux 内核初步启动后，会启动第一个进程，在默认的 SDK 中我们使用 systemd 进行资源管理和一些服务的启动，第一个进程通常 sbin/目录下的 init 程序，这个程序会加载 systemd 进行文件系统的挂载，外设初始化，内核模块加载，网络服务启动等一系列任务。Linux 内核的启动 log 如下：

```
[2025-08-05 19:53:33.760]
[2025-08-05 19:53:33.760] Starting kernel ...
[2025-08-05 19:53:33.760]
[2025-08-05 19:53:33.792] [ 0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd034]
[2025-08-05 19:53:33.792] [ 0.000000] Linux version 6.6.58-ti-01497-ga7758da17c28-dirty (oe-user@oe-
host) (aarch64-oe-linux-gcc (GCC4
 ..........
 ..........
[2025-08-05 19:53:37.040] [ 2.356567] VFS: Mounted root (ext4 filesystem) on device 179:98.
[2025-08-05 19:53:37.057] [ 2.367111] devtmpfs: mounted
[2025-08-05 19:53:37.057] [ 2.370726] Freeing unused kernel memory: 2496K
```

Linux 内核的启动会占用大约 5s，这部分的启动加速主要在于裁减 Linux 内核和减少 log 的输出。裁减 Linux 内核以取消一些对目标应用没有作用的内核服务，减小内核镜像的体积，能有效地加速启动服务。Log 的输出会占用很大一部分的启动时间，取消不必要的输出 log 能有效减少启动时间。关于如何裁减操作系统，减少 log 输出将在第四节详细描述。

在内核启动结束后，用户态的启动主要由第一个进程 init 开启，由 systemd 进行后续的启动和配置。其启动 log 如下：

```
[2025-08-05 19:53:37.072] [ 2.375331] Run /sbin/init as init process
[2025-08-05 19:53:37.264] [ 2.575462] systemd[1]: System time before build time, advancing clock.
[2025-08-05 19:53:37.312] [ 2.617899] NET: Registered PF_INET6 protocol family
[2025-08-05 19:53:37.312] [ 2.623708] Segment Routing with IPv6
 ..........
 ..........
[2025-08-05 19:53:48.479] Arago Project am62axx-evm ttyS2
[2025-08-05 19:53:48.479]
[2025-08-05 19:53:48.479] Arago 2023.10 am62axx-evm ttyS2
[2025-08-05 19:53:48.479]
[2025-08-05 19:53:48.479] am62axx-evm login:
```

在 systemd 接管后，这一段的启动时间约 10s，因此裁减 systemd 或者使用 sysVinit 以优先启动必要的服务能有效减少启动时间，对启动速度优化影响极大。这部分的优化将在第四节进一步讨论。

2.2 SPL Falcon 模式

Falcon 模式指的是将 Linux 镜像打包进 tispl.bin 中，在 A53 SPL 启动阶段直接加载启动以跳过 U-Boot 阶段。Falcon 有多种实现方式，对于 AM62A 而言，在 SDK11.01 及以后，可以在 yocto 编译阶段指定使用 falcon 模式快速实现，使用 SDK11.01 时可以参考相应的用户手册，对于后续的 SDK 版本可以参考对应用户手册的相应章节；对于其他 SDK 版本，可以在 Linux SDK 中简单配置后启用，本文将描述在 SDK 中进行配置的方式，这种方式也适用于 SDK11.01 以后的版本。

2.2.1 实现步骤

2.2.1.1 安装 Linux SDK

在 ti.com 中搜索对应芯片的 SDK，如 AM62A，其 SDK 可以在 https://www.ti.com/tool/PROCESSOR-SDK-AM62A 中找到。按照 PC 电脑操作系统类型（Windows，Linux）选择适合的 SDK 下载安装即可。

2.2.1.2 添加 falcon 脚本

在默认的 SDK 中没有 falcon 脚本，请参考 git patch 中的信息自行添加。在这个脚本中主要定义了一些镜像的加载地址和入口。如下列字段中主要定义了 Linux 内核代码镜像加载目标地址。

```
kernel {
 description = "Linux Kernel (64-bit)";
 data = /incbin/("$1");
 type = "standalone";
 os = "linux";
 arch = "arm64";
 compression = "none";
 load = <0x80080000>; /* Needs to match PRELOADED_BL33_BASE in ATF */
 entry = <0x80080000>;
};
```

2.2.1.3 编译 Linux 设备树

因为这种启动方式的核心在于跳过 U-boot，因此需要将一些原本用 Overlay 方式添加的设备树描述文件加入主设备树描述中，此外还需要将文件系统位置告知操作系统。本文在实现快速启动时对主设备树源文件做了如下改动。

```
--- a/arch/arm64/boot/dts/ti/k3-am62a7-sk.dts
+++ b/arch/arm64/boot/dts/ti/k3-am62a7-sk.dts
@@ -30,6 +30,7 @@ aliases {

 chosen {
 stdout-path = "serial2:115200n8";
+ bootargs = "console=ttyS2,115200n8 earlycon=ns16550a,mmio32,0x02800000 loglevel=4 root=/dev/
mmcblk1p2 rw rootfstype=ext4 rootwait";
 };
```

改动后需要重新编译设备树二进制文件 dtb。在 SDK 的安装目录下，执行如下命令重新编译设备树。

```
make linux_dtbs
```

如果进一步修改了 Linux kernel，需要重新编译 Linux 镜像。

2.2.1.4 制作 tispl.bin

运行如下命令生成.its 文件。

```
ATF=$IMAGES_DIR/bl31.bin TEE=$IMAGES_DIR/bl32.bin \
 DM=$IMAGES_DIR/ti-dm/am62axx/ipc_echo_testb_mcu1_0_release_strip.xer5f \
 /home/ht/ti-processor-sdk-linux-edgeai-am62a-evm-10_01_00_05/board-support/ti-u-boot-2024.04+git/
scripts/tispl-linux-falcon.sh \
 $IMAGES_DIR/../ti-linux-kernel-6.6.58+git-ti/arch/arm64/boot/Image $IMAGES_DIR/../ti-linux-
kernel-6.6.58+git-ti/arch/arm64/boot/dts/ti/k3-am62a7-sk.dtb > tispl-falcon.its
```

在该命令中，主要需要指定 falcon 二进制文件中各个组件的位置如 ATF，TEE，DM 等，使用时需要按照自己的实际情况进行修改。在实例命令中，IMAGES_DIR 为 SDK 安装目录中 board-support/prebuilt-images 文件夹。

接着运行如下命令生成 tispl-falcon.bin

```
mkimage -E -f tispl-falcon.its tispl-falcon.bin
```

该命令会生成 tispl-falcon.bin。

2.2.1.5 制卡启动

参考快速评估指南制作 SD 启动卡。在启动卡中，默认的 Boot 分区会有 tiboot3.bin，tispl.bin，u-boot.image 文件，分别对应了 R5SPL，A53SPL 和 U-boot。使用上一步骤中生成的 tispl-falcon.bin 替换 tispl.bin 即可。此时插卡启动，可以看到 log 中会跳过 U-boot 相关内容而直接启动 Linux 内核。

2.3 从 SPL 替换为 SBL

SBL 是 TI 提供的另一种启动方式，这种启动方式抛弃了传统的 SPL 启动，其启动流程如下图。在 ROM 代码启动后，R5 核的启动分为两步，SBL1 和 SBL2，在 SBL2 启动过程中会将其他小核的镜像也加载启动。这种启动方式也可以结合 falcon 方式，跳过 U-boot 直接启动 trusted firmware，optee 和内核，最后由内核启动文件系统。在文件系统启动的过程中，还能进一步将文件系统分为两部分，第一部分仅启动需要快速启动的服务，第二部分启动剩余的完整文件系统。

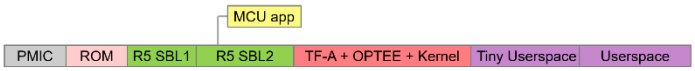

图 2-6. SBL 启动流程

使用这种启动方式还需要下载额外的 MCU SDK，在该 SDK 的文档中，描述了不同启动介质下的 SBL 启动方法。本文主要以 EMMC 启动为例，介绍 EMMC 启动情况下的实现步骤。启动介质的影响不仅仅在于不同启动介质的传输速率，还在于 SBL 启动模式下对高速模式的配置。这是由于一些启动介质的配置较为复杂，需要 Linux 操作系统启动时才能完成高速模式的配置，在启动的前期只能使用低速模式进行镜像的搬运。

SBL 的实现可以完全参考实现指南。在指南中说明了需要以下核的镜像：

```
A Linux appimage containing the Linux binaries (ATF, OPTEE, A53 SPL).
tiboot3.bin with SBL stage1, TIFS, BoardConfig
Appimage for SBL stage2
Appimage for MCU R5
Appimage for HSM M4
Appimage for C75
DM firmware appimage for DM R5
```

其中，Linux appimage，tiboot3，SBL stage2， HSM 镜像均有详细描述。C7 以及 R5 核的 appimage 缺少相应教程指导如何打包，接下来的内容将介绍简单的镜像打包方式。

以 C7 核镜像为例，在编译完 MCU Plus SDK 后，在 mcu_plus_sdk_am62ax_xxxxxx/examples/drivers/ipc/ipc_rpmsg_echo_linux/am62ax-sk/c75ss0-0_freertos/ti-c7000 目录下，将生成的 appimage，out，rprc 文件删除，将 Linux SDK 中 ti-processor-sdk-linux-edgeai-am62a-evm-xxxxx/filesystem/rootfs/lib/firmware/vision_apps_eaik 目录下的 out 文件复制到 ti-c7000 目录下替换掉删除的 out 文件，需要注意的是文件名需要与被删除的 out 文件一致，重新编译 MCU SDK 中对应 example。这种方式是借用 MCU SDK 编译指令将 DSP 的镜像打包为需要的 appimage。

R5 相关的镜像打包也可以参考这个方式生成。

在所需镜像都打包完成后，按照 demo 运行指南将镜像全部烧入 EMMC，随后启动即可。

2.3.1 在 SBL 中进行 Falcon 启动

SBL 模式下的 Falcon 启动主要参考 Linux 镜像打包用户手册配置即可。

3 启动介质

启动介质对启动速度的影响非常重要，在硬件层面，启动介质的速度会影响镜像文件的搬运，在软件层面，软件是否尽早将介质配置为高速模式也极大地影响启动速度。在 SDK 文档中，列出了主要的启动介质的读写速度测试结果。

在 SPL 中，高速模式的配置位于 U-boot 中，因此使用 Falcon 模式跳过 U-boot 也会造成高速模式需要在后续 Linux 启动过程中调用相关驱动程序配置，高速模式仅能加速文件系统相关镜像的加载，Linux Image 以及其他 SPL 镜像的搬运会运行在低速模式下，加载耗时将大于 1s，不利于快速启动。以下 Log 是 falcon 模式下的 SPL 启动，其启动耗时与跳过 U-boot 等待的常规 SPL 流程几乎相同。并且在这种启动模式下，无论何种启动介质，在进入 Linux 之前都运行在低速模式下，然而需要注意的是，后续的 SDK 可能会修改这一点，可能会在未来的版本中会将快速模式的使能提前以加速启动，因此需要根据 SDK 版本进行判断。

```
[2025-07-04 17:26:03.270] U-Boot SPL 2024.04-ti-g29d0c23d67ee (Nov 29 2024 - 11:41:54 +0000)
[2025-07-04 17:26:03.270] SYSFW ABI: 4.0 (firmware rev 0x000a '10.1.8--v10.01.08 (Fiery Fox)')
[2025-07-04 17:26:03.334] SPL initial stack usage: 13568 bytes
[2025-07-04 17:26:03.334] Trying to boot from MMC2
[2025-07-04 17:26:03.461] Warning: Did not detect image signing certificate. Skipping
authentication to prevent boot failure. This will fail on Security Enforcing(HS-SE) devices
[2025-07-04 17:26:04.229] Warning: Did not detect image signing certificate. Skipping
authentication to prevent boot failure. This will fail on Security Enforcing(HS-SE) devices
[2025-07-04 17:26:04.517] Warning: Did not detect image signing certificate. Skipping
authentication to prevent boot failure. This will fail on Security Enforcing(HS-SE) devices
[2025-07-04 17:26:04.566] Warning: Did not detect image signing certificate. Skipping
authentication to prevent boot failure. This will fail on Security Enforcing(HS-SE) devices
[2025-07-04 17:26:04.566] Warning: Did not detect image signing certificate. Skipping
authentication to prevent boot failure. This will fail on Security Enforcing(HS-SE) devices
[2025-07-04 17:26:04.597] Starting ATF on ARM64 core...
```

在 SBL 模式下，其启动代码更加定制化，用户可以更加方便地修改定制启动代码。在使用 EMMC 模式下结合 falcon 的 SBL 启动 log 如下。在启动开始时，第一阶段的 SBL 就将启动介质设置为高速模式，在后续的镜像加载过程中，高速模式下的镜像搬运极大缩减了启动时间。在 Linux 启动之前，总耗时仅 600ms，请参考以下启动 log。

```
 [2025-07-08 10:03:51.246] SYSFW Firmware revision 0xa
[2025-07-08 10:03:51.246] SYSFW ABI revision 4.0
[2025-07-08 10:03:51.246]
[2025-07-08 10:03:51.262] [BOOTLOADER_PROFILE] Boot Media : eMMC
[2025-07-08 10:03:51.278] [BOOTLOADER_PROFILE] Boot Media Clock : 200.000 MHz
[2025-07-08 10:03:51.278] [BOOTLOADER_PROFILE] Boot Image Size : 163 KB
[2025-07-08 10:03:51.278] [BOOTLOADER_PROFILE] Cores present :
[2025-07-08 10:03:51.294] mcu-r5f0-0
[2025-07-08 10:03:51.294] r5f0-0
[2025-07-08 10:03:51.294] [BOOTLOADER PROFILE] System_init : 36549us
[2025-07-08 10:03:51.294] [BOOTLOADER PROFILE] Board_init : 0us
[2025-07-08 10:03:51.294] [BOOTLOADER PROFILE] Drivers_open : 20614us
[2025-07-08 10:03:51.310] [BOOTLOADER PROFILE] Board_driversOpen : 0us
[2025-07-08 10:03:51.310] [BOOTLOADER PROFILE] Sciclient Get Version : 10174us
[2025-07-08 10:03:51.326] [BOOTLOADER PROFILE] MCU R5 Image Load : 7232us
[2025-07-08 10:03:51.326] [BOOTLOADER PROFILE] DM R5 Image Load : 8510us
[2025-07-08 10:03:51.326] [BOOTLOADER_PROFILE] SBL Total Time Taken : 83081us
[2025-07-08 10:03:51.342]
[2025-07-08 10:03:51.342] Image loading done, switching to application ...
[2025-07-08 10:03:51.342] Starting MCU-r5f and 2nd stage bootloader
[2025-07-08 10:03:51.374]
[2025-07-08 10:03:51.374] SYSFW Firmware Version 10.1.8--v10.01.08 (Fiery Fox)
[2025-07-08 10:03:51.390] SYSFW Firmware revision 0xa
[2025-07-08 10:03:51.390] SYSFW ABI revision 4.0
[2025-07-08 10:03:51.390]
[2025-07-08 10:03:51.758] ERROR: Bootloader_verifyMulticoreImage:684: Failed to authenticate Image
[2025-07-08 10:03:51.758] [BOOTLOADER_PROFILE] Boot Media : eMMC
[2025-07-08 10:03:51.758] [BOOTLOADER_PROFILE] Boot Media Clock : 200.000 MHz
[2025-07-08 10:03:51.774] [BOOTLOADER_PROFILE] Boot Image Size : 16589 KB
[2025-07-08 10:03:51.774] [BOOTLOADER_PROFILE] Cores present :
[2025-07-08 10:03:51.774] hsm-m4f0-0
[2025-07-08 10:03:51.774] r5f0-0
[2025-07-08 10:03:51.774] a530-0
[2025-07-08 10:03:51.774] [BOOTLOADER PROFILE] System_init : 1645us
[2025-07-08 10:03:51.790] [BOOTLOADER PROFILE] Board_init : 0us
[2025-07-08 10:03:51.790] [BOOTLOADER PROFILE] Drivers_open : 20540us
```

```
[2025-07-08 10:03:51.806] [BOOTLOADER PROFILE] Board_driversOpen : 0us
[2025-07-08 10:03:51.806] [BOOTLOADER PROFILE] Sciclient Get Version : 10234us
[2025-07-08 10:03:51.806] [BOOTLOADER PROFILE] HSM Image Load : 5857us
[2025-07-08 10:03:51.822] [BOOTLOADER PROFILE] DM R5 Image Load : 15695us
[2025-07-08 10:03:51.822] [BOOTLOADER PROFILE] A53 Image Load : 291147us
 [2025-07-08 10:03:51.838] [BOOTLOADER_PROFILE] SBL Total Time Taken : 345123us
```

4 操作系统启动优化

4.1 裁剪操作系统

操作系统对启动的影响有两个方面，一方面是镜像大小会影响搬运速度，另一方面是一些内核模块的初始化代码运行需要耗费相当多的时间。因此裁减操作系统既能加快镜像的搬运，又能加速内核启动。

裁减操作系统镜像的主要方式是将一些内核模块禁用或者改为动态加载的模块。这个可以直接修改 arch/arm64/configs/defconfig 文件中的定义来控制或者通过 Linux 配置工具。

4.1.1 使用预设的裁剪

为了快速实现快速启动功能，TI 提供了额外的 overlay config 来帮助裁减内核。在内核代码中，kernel/configs/ti_early_display.config 文件提供了额外的内核模块配置。该配置中有如下的描述，详细说明了关闭该模块能够带来的启动优化。

```
#This reduces kernel image size by 0KB and boot time by 144ms
CONFIG_RODATA_FULL_DEFAULT_ENABLED=n
```

为了应用预设的快速启动配置，需要修改相应的 makerule。首先进入 Linux SDK 安装目录下的 makerules 文件夹，修改 Makefile_linux 相应部分，将 ti_early_display.config 加入编译命令。

```
linux: linux-dtbs
@echo ================================
@echo Building the Linux Kernel
@echo ================================
$(MAKE) -C $(LINUXKERNEL_INSTALL_DIR) ARCH=arm64 CROSS_COMPILE=$(CROSS_COMPILE) defconfig
ti_arm64_prune.config ti_early_display.config
$(MAKE) -j $(MAKE_JOBS) -C $(LINUXKERNEL_INSTALL_DIR) ARCH=arm64 CROSS_COMPILE=$(CROSS_COMPILE)
Image
$(MAKE) -j $(MAKE_JOBS) -C $(LINUXKERNEL_INSTALL_DIR) ARCH=arm64 CROSS_COMPILE=$(CROSS_COMPILE)
modules
ifeq ($(SOC_TYPE),hs)
$(MAKE) linux-fitImage
endif
```

由于 Linux 依赖于设备树，因此设备树的编译命令中也需要加入该配置文件，在 Makefile_linux-dtbs 中，将 ti_early_display.config 加入编译命令。

```
linux-dtbs:
@echo ================================
@echo Building the Linux Kernel DTBs
@echo ================================
$(MAKE) -C $(LINUXKERNEL_INSTALL_DIR) ARCH=arm64 CROSS_COMPILE=$(CROSS_COMPILE) defconfig
ti_arm64_prune.config ti_early_display.config
@for DTB in $(KERNEL_DEVICETREE); do \
$(MAKE) -j $(MAKE_JOBS) -C $(LINUXKERNEL_INSTALL_DIR) DTC_FLAGS=-@ ARCH=arm64 CROSS_COMPILE=$
(CROSS_COMPILE) $$DTB; \
done
```

在修改完编译指令后，重新编译 Linux，重新编译出的镜像文件大小从 20MB 下降到 15MB 左右，同时 Linux 启动时间降低约 1s。

4.1.2 自定义裁剪

自定义裁减需要修改 arch/arm64/configs/defconfig 中的配置，比如将 CONFIG_PPS 设置为 y 时将该配置的相关代码编入内核中并在内核启动时启动，设置为 m 时即为动态加载的模块，在内核启动完成后文件系统启动时加载，设置为 n 时则禁用。然而需要注意的是，很多模块具有相互依赖的关系，因此简单修改 defconfig 可能并不生效，具体是否生效可以在编译后查看 Linux SDK 目录下 board-support/ti-linux-xxxx 目录下.config 确认最终生效的配置。例如，B 模块依赖于 A，将 A 配置为禁用，但是 B 配置为使能，编译完会发现在.config 中 A 仍然能被使能。

自定义裁减也可以使用 Linux 配置工具 menuconfig 进行配置，该配置工具在配置时会检查依赖关系，帮助有效配置。

举个例子，如下图所示，在目标应用中可能不需要使用蓝牙以及相关协议栈，使用 menuconfig 时会打开图形界面，使用方向键进行选择控制。

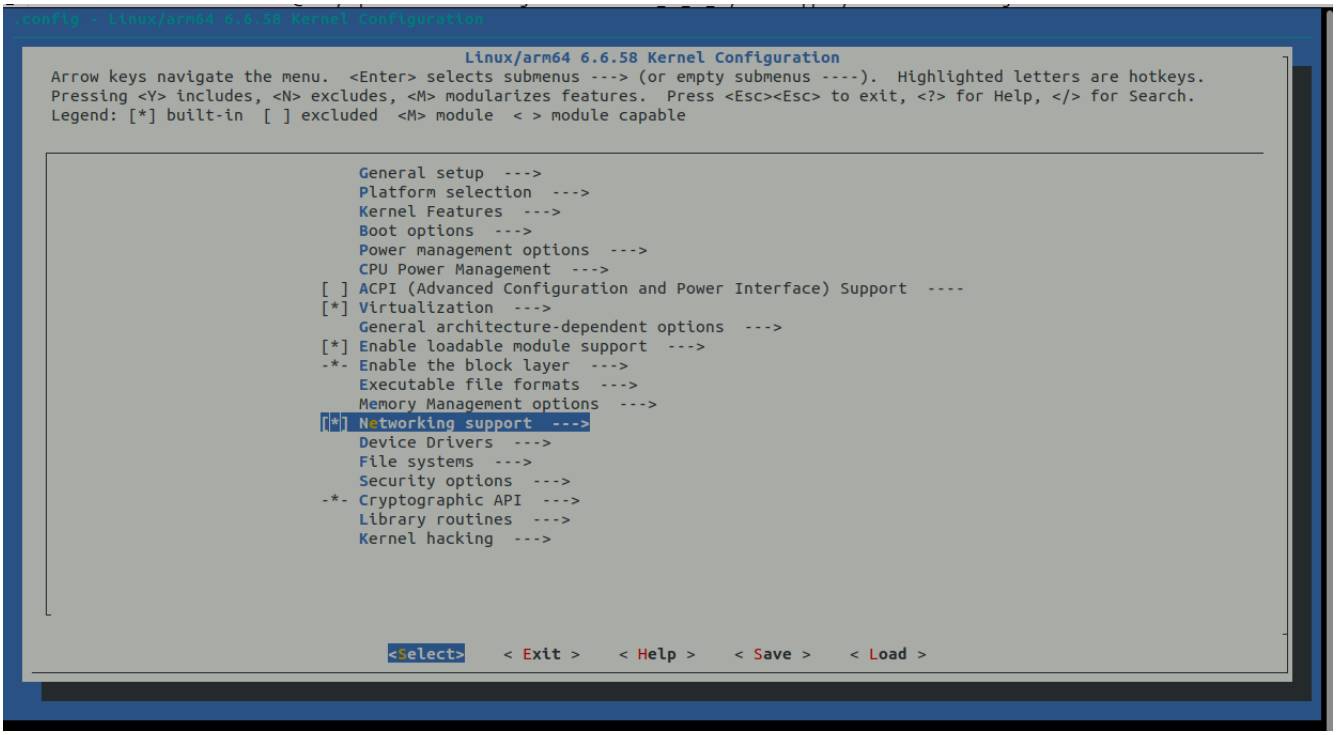

图 4-1. Menuconfig 界面

选择网络支持项后，可以看到蓝牙相关选项。在蓝牙选项上按 m 则配置为动态加载的模块，按 n 配置为禁用，按 y 配置为内核模块，会与内核一起启动。

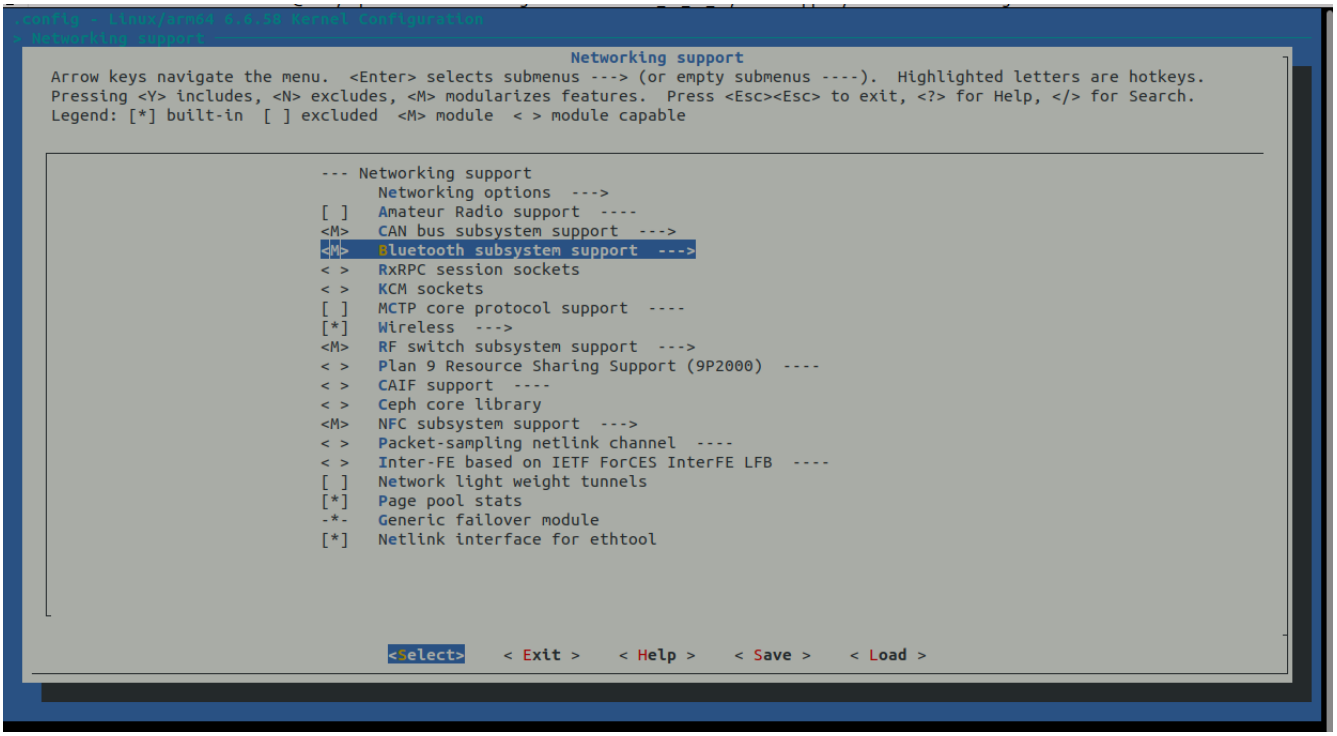

图 4-2. 在 Menuconfig 界面中配置蓝牙

修改完成后，选择保存。

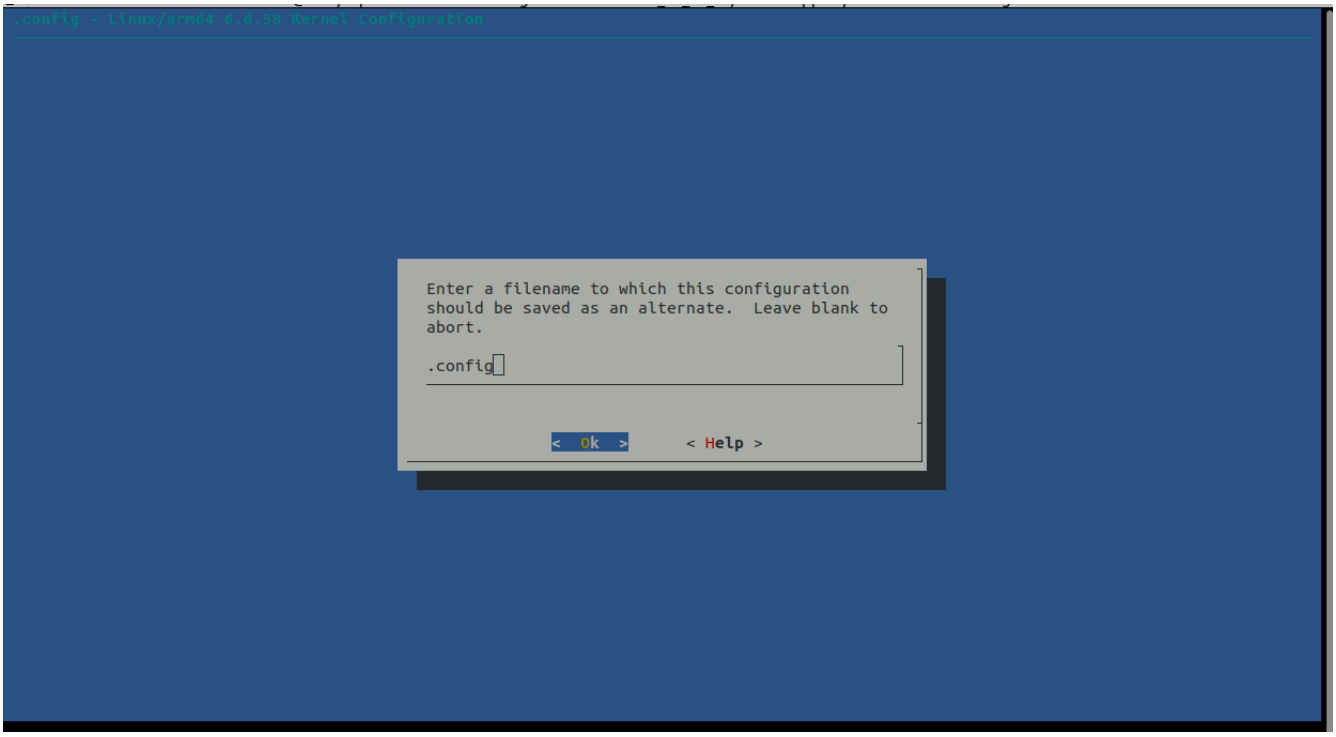

图 4-3. 保存 Menuconfig

4.2 减少 log 输出

在默认的启动流程中，启动 log 会在启动时通过串口打印。打印 log 会极大增加启动时间，减少 log 会带来数秒的启动加速。相关步骤请参考在设备树中添加 quiet。在完成减少 log 输出以及裁减内核后，内核启动阶段耗时减少到 1s 左右。

5 用户空间优化

5.1 使用 systemd 自定义启动服务

默认 SDK 文件系统/用户空间服务的启动使用 systemd。Linux 内核启动后启动的第一个进程是/sbin/init 程序，该程序紧接着就会启动 systemd 程序完成剩余的配置。TI 使用的 systemd 符合 Linux 标准情况下的启动流程，请参考 Linux systemd bootup 文档了解启动的相关服务。

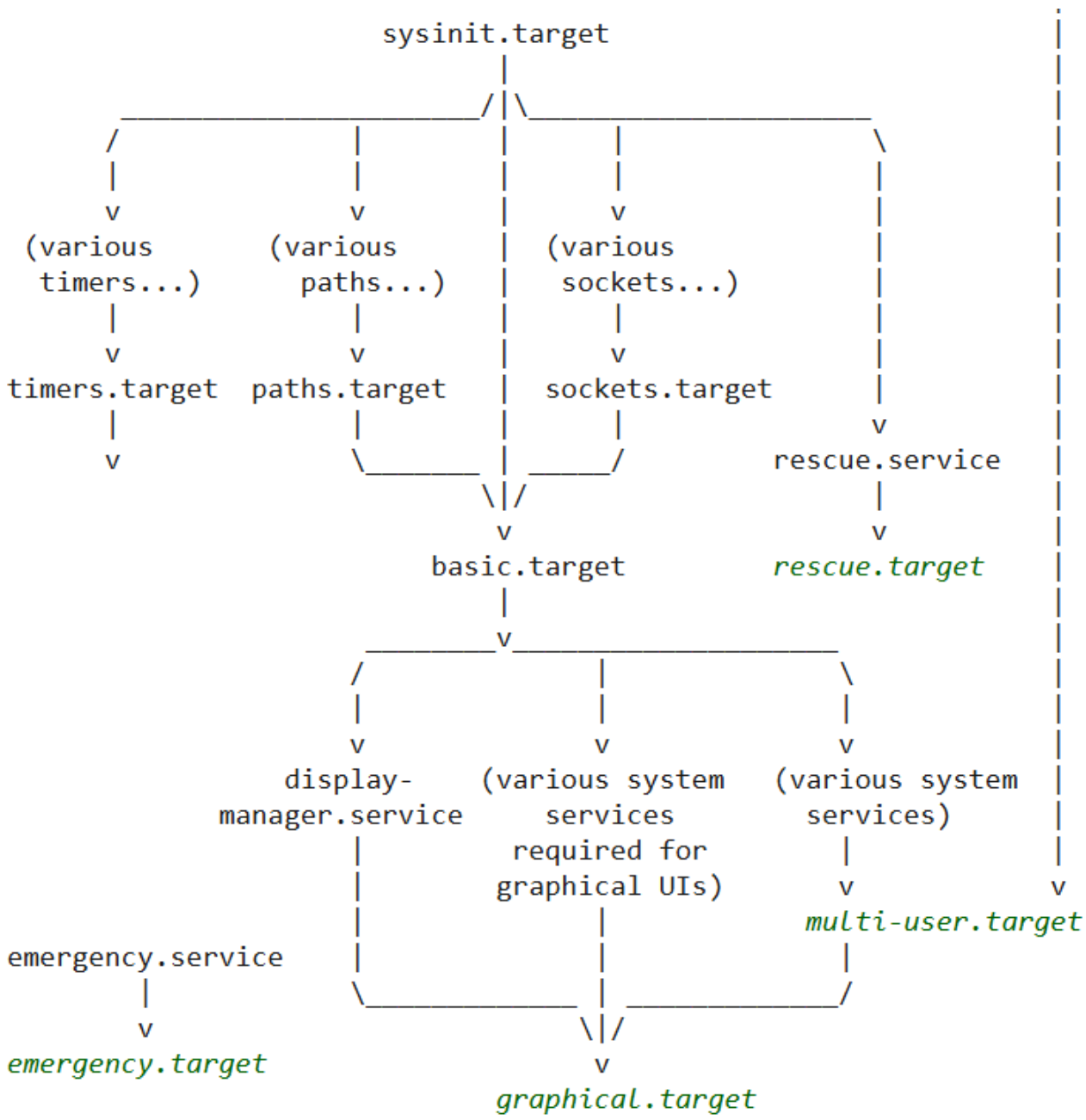

图 5-1. systemd 启动依赖项

Systemd 使用 target 进行启动服务的管理，当运行到最终 target 即 graphical.target 时 systemd 启动完成，标志着用户态启动结束。默认 systemd 的最终目标为 graphical.target.

在常规启动后，可以通过 dmesg | grep target 看到 systemd 启动时确认的目标被设置为 graphical。

```
systemd[1]: Queued start job for default target Graphical Interface.
```

使用如下命令可以查看该 target 的状态：

```
systemctl status graphical.target
```

在启动完成后，其状态为如 log 所示，已经完成了该 target 的启动。

```
 am62axx-evm systemd[1]: Reached target Graphical Interface.
```

状态中还包括了如下的描述，指明了目标这个 target 对应的配置来源：

```
Loaded: loaded (8;;file://am62axx-evm/usr/lib/systemd/system/graphical.target/usr/lib/systemd/
system/graphical.target
```

根据这个提示，可以查看这个 target 的详细配置。

```
# SPDX-License-Identifier: LGPL-2.1-or-later
#
# This file is part of systemd.
#
# systemd is free software; you can redistribute it and/or modify it
# under the terms of the GNU Lesser General Public License as published by
# the Free Software Foundation; either version 2.1 of the License, or
# (at your option) any later version.
[Unit]
Description=Graphical Interface
Documentation=man:systemd.special(7)
Requires=multi-user.target
Wants=display-manager.service
Conflicts=rescue.service rescue.target
After=multi-user.target rescue.service rescue.target display-manager.service
AllowIsolate=yes
```

这个 target 会依赖于 multi-user.target 等 units。通过这种方式，能够梳理出当前 systemd 执行的所有 units。在梳理完所有的服务后可以根据自己的需要，关闭一些服务。使用以下命令关闭服务：

```
systemctl mask <some unit>
```

6 使用 init 脚本最小化启动服务

Linux 启动后第一个启动的是/sbin/init 程序，可以使用自定义的启动脚本替换该程序。替换时需要先创建脚本，接着将/sbin/init 删除，最后使用 ln -sf <your script> /sbin/init 命令将自定义脚本软连接为启动程序。

以下是一个 CMS 快速启动的参考脚本。该脚本最终启动 edgeai-tiovx-apps-main 应用程序。该脚本首先执行一些 mount 命令以挂载必要的分区；接着使用 modprobe 命令加载一些必要的模组，包括相机驱动，csi-rx，v4l2 等用于摄像头数据的接入，rpmsg 等用于与 AI DSP 和执行 ISP 的 R 核通信，tidss 以及 display_connecter 用于显示；然后 export 一些环境变量为最后的运行服务做准备；最终运行 edgeai-tiovx-apps-main 可执行程序。

```
#!/bin/sh
set -x
export PATH=/usr/bin:/sbin:/bin
export LD_LIBRARY_PATH=/lib:/usr/lib:$LD_LIBRARY_PATH
mount -t proc proc /proc
mount -n -t sysfs sysfs /sys
mount -t debugfs debugfs /sys/kernel/debug
mount -t configfs configfs /sys/kernel/config
mount -t efivarfs efivarfs /sys/firmware/efi/efivars
mount -n -t tmpfs none /run
mount -n -t devtmpfs devtmpfs /dev
mkdir -p /dev/shm
chmod 1777 /dev/shm
mount -t tmpfs /dev/shm
export SOC=am62a
export DEVICE_NAME=AM62A
# Load essential kernel modules
insmod /lib/modules/6.6.58-ti-01497-ga7758da17c28-dirty/kernel/drivers/rpmsg/rpmsg_char.ko
insmod /lib/modules/6.6.58-ti-01497-ga7758da17c28-dirty/kernel/drivers/rpmsg/rpmsg_ctrl.ko
modprobe sii902x
modprobe tidss
modprobe display_connector
modprobe ti_k3_r5_remoteproc
modprobe ti_k3_dsp_remoteproc
modprobe omap_mailbox
modprobe cdns_csi2rx
modprobe j721e_csi2rx
modprobe v4l2_cci
modprobe videobuf2_dma_contig
modprobe v4l2_fwnode
modprobe videobuf2_memops
modprobe videobuf2_v4l2
modprobe videobuf2_common
modprobe v4l2_async
modprobe videodev
modprobe cdns_dphy_rx
modprobe mc
modprobe imx219
ulimit -c unlimited
echo "/tmp/core.%e.%p" > /proc/sys/kernel/core_pattern
export EDGEAI_GST_APPS_PATH=/opt/edgeai-gst-apps
export EDGEAI_DATA_PATH=/opt/edgeai-test-data
export OOB_DEMO_ASSETS_PATH=/opt/oob-demo-assets
export MODEL_ZOO_PATH=/opt/model_zoo
export EDGEAI_VERSION=10.1
export EDGEAI_SDK_VERSION=10_01_00
bash /opt/edgeai-gst-apps/scripts/setup_cameras.sh
./opt/edgeai-tiovx-apps/bin/Release/edgeai-tiovx-apps-main ./opt/edgeai-tiovx-apps/configs/linux/
imx219_cam_example.yaml &
echo "System initialized: Hello world!!!!!!"
/bin/sh
```

7 总结

本文描述了 SITARA AM62Ax 系列处理器的快速启动方法。在结合 falcon sbl，emmc 启动，内核启动优化，用户态使用最小启动脚本的方式后，能够实现 AM62A 在 2.5s 内启动 CMS 应用，实现了 CMS 等应用的快起需求。本文详细讲述了各个环节的实现方法以便于快速复现。

8 参考文献

1. https://www.ti.com/product/AM62A7
2. https://www.ti.com/tool/PROCESSOR-SDK-AM62A
3. https://software-dl.ti.com/processor-sdk-linux/esd/AM62AX/11_01_07_05/exports/docs/devices/AM62AX/linux/Linux_Performance_Guide.html
4. https://e2e.ti.com/support/processors-group/processors/f/processors-forum/1510026/am62a7-soc-system-startup-time-is-more-than-20s
5. https://e2e.ti.com/support/processors-group/processors/f/processors-forum/1274738/am62a7-based-on-sdk-9-version-implement-a-cms-project-car-rearview-mirror-how-to-import-ai-tag-function-through-gstreamer
6. https://e2e.ti.com/support/processors-group/processors/f/processors-forum/1479388/am62a7-how-to-set-the-parameter-for-cms-product-what-is-using-for-models-in-pipiline
7. https://e2e.ti.com/support/processors-group/processors/f/processors-forum/1479107/am62a7-cms-resize

重要通知和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、与某特定用途的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他安全、安保法规或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的相关应用。严禁以其他方式对这些资源进行复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。对于因您对这些资源的使用而对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，您将全额赔偿，TI 对此概不负责。

TI 提供的产品受 TI 销售条款)、TI 通用质量指南 或 ti.com 上其他适用条款或 TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。 除非德州仪器 (TI) 明确将某产品指定为定制产品或客户特定产品，否则其产品均为按确定价格收入目录的标准通用器件。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

最后更新日期：2025 年 10 月