AN14093

Fast Boot on i.MX 8M and i.MX 9 using Falcon Mode and Kernel Optimizations

Rev. 1 — 9 October 2023

Application note

COMPANY PUBLIC

Document information

Information	Content
Keywords	AN14093, Falcon Mode, Kernel Optimizations, U-Boot Optimizations, Fast Boot, Falcon Boot, i.MX93, i.MX8M, Linux
Abstract	This document guides how to reduce the boot time for the i.MX 8M and i.MX 9 family.

1 Introduction

This document guides how to reduce the boot time for the:

- i.MX 8M family (i.MX 8M Mini LPDDR4 EVK, i.MX 8M Nano LPDDR4 EVK, and i.MX 8M Plus LPDDR4 EVK)
- i.MX 9 family (i.MX 93 LPDDR4 EVK)

The objectives of this document are as follows:

- Bootloader optimizations
- Linux kernel and user space optimizations
- Comparison between default and improved boot time on all platforms

1.1 Software environment

An Ubuntu 20.04 PC is assumed.

Linux board support package (BSP) release 6.1.22_2.0.0 is used in the optimization process.

The following prebuild images are used:

- **i.MX 8M Mini**: `imx-image-full-imx8mmevk.wic`
- **i.MX 8M Nano**: `imx-image-full-imx8mnevk.wic`
- **i.MX 8M Plus**: `imx-image-full-imx8mpevk.wic`
- **i.MX 93**: `imx-image-full-imx93evk.wic`

Write the prebuild image on the SD card using the below command:

```
$  sudo dd if=<image_name>.wic of=/dev/sd<x> bs=1M status=progress conv=fsync
```

Note: *Check your card reader partition and replace* `sd<x>` *with your corresponding partition.*

1.2 Hardware setup and equipment

- Development kit NXP i.MX 8MM EVK LPDDR4
- Development kit NXP i.MX 8MN EVK LPDDR4
- Development kit NXP i.MX 8MP EVK LPDDR4
- Development kit NXP i.MX 93 EVK for 11x11 mm LPDDR4
- microSD card: SanDisk Ultra 32 GB micro secure digital high capacity (SDHC) I Class 10 was used for the current experiment
- micro-USB (i.MX 8M) or Type-C (i.MX 93) cable for debug port

2 General description

This section describes an overview of the typical modifications required to achieve shorter boot times.

2.1 Reduce the bootloader time

You can opt for either of the following two ways to reduce the bootloader time.

- **Remove the boot delay** — Saves about two seconds compared to default configuration while requiring minimal changes. It leads to U-Boot skipping the wait for the keypress stage during boot.
- **Implement the Falcon mode** — Saves about four seconds compared to the default configuration. It enables the second program loader (SPL), a part of U-Boot to load the kernel directly, skipping the full U-Boot.

2.2 Reduce the Linux kernel boot time

- **Reduce console messages** — Saves about three seconds. Add `quiet` to the kernel command line.
- **Slim down the kernel by removing drivers and filesystems** — By default, the kernel image contains a lot of drivers and filesystems (ex: UBIFS) to enable most of the functionalities supported for the board. The list of included drivers and filesystems can be trimmed according to your use case.

2.3 Reduce the user-space boot time

- **Change the running order in the initialization Systemd scripts** — Saves about 600 ms. Launch the desired process as soon as possible, considering its dependencies.

3 Measurements

The scope of the measurements is between the board POR (Power-On Reset) and the start of the INIT process.

The setup used for the following measurements is described in the Boot Time Measurements Methodology document.

Table 1. Measured intervals

Time point	Interval between pulses	Location of the pulse	Boot stages	
BootROM	nRST -> before ddr_init()	board/freescale/<board>/spl.c/board_init_f()	SPL	Timeline
DDR initialization	before ddr_init() -> after ddr_init()	board/freescale/<board>/spl.c/board_init_f()		
SPL initialization + Load U-Boot image	after ddr_init() -> before image_entry()	common/spl/spl.c/jump_to_image_no_args()		
U-Boot initializations (init_sequence_f)	before image_entry() -> start init_sequence_ r	common/board_r.c/board_init_r()	U-BOOT	
U-Boot initializations (init_sequence_r)	start init_sequence_r -> u-boot main_loop	common/main.c		
Boot sequence	u-boot main_loop -> before load_image	include/configs/<board>.h		
Kernel image load	before load_image -> after load_image	include/configs/<board>.h		
Kernel boot until INIT process	after load_image -> /sbin/init	get the timestamp during Kernel boot	Kernel	

4 Prerequisites

In this section, the software needed to compile the U-Boot and the Linux kernel in a standalone environment is described.

- **Install the required dependencies**

A series of dependencies, including an ARM64 cross-compiler, are required for this guide.

```
$ sudo apt install flex bison libssl-dev gcc-aarch64-linux-gnu u-boot-tools
 libncurses5-dev libncursesw5-dev uuid-dev gnutls-dev
```

Next, download the required sources. Place them all in the same directory.

- **Download `imx-mkimage`**

`mkimage` is a tool, which combines the SPL, U-Boot proper, ATF, and DDR firmware into a single image, resulting in the U-Boot image to be flashed on the SD card.

```
$ git clone https://github.com/nxp-imx/imx-mkimage
$ cd imx-mkimage
$ git checkout lf-6.1.22-2.0.0
```

- **Download ATF**

```
$ git clone https://github.com/nxp-imx/imx-atf
$ cd imx-atf
$ git checkout lf-6.1.22-2.0.0
```

- **Download U-Boot**

```
$ git clone https://github.com/nxp-imx/uboot-imx
$ cd uboot-imx
$ git checkout lf-6.1.22-2.0.0
```

- **Download Linux kernel**

```
$ git clone https://github.com/nxp-imx/linux-imx
$ cd linux-imx
$ git checkout lf-6.1.22-2.0.0
```

- **Download the double data rate (DDR) firmware**

```
$ wget https://www.nxp.com/lgfiles/NMG/MAD/YOCTO/firmware-imx-8.20.bin
$ chmod +x firmware-imx-8.20.bin
$ ./firmware-imx-8.20.bin
```

- **[Only for i.MX 93] Download the EdgeLock Secure Enclave (ELE) firmware**

```
$ wget https://www.nxp.com/lgfiles/NMG/MAD/YOCTO/firmware-sentinel-0.9.bin
$ chmod +x firmware-sentinel-0.9.bin
$ ./firmware-sentinel-0.9.bin
```

5 Build the images

Optionally, to check whether the sources and the prerequisites were downloaded correctly, execute the following steps. Otherwise, skip for now and implement Section 7 "Bootloader optimizations" and Section 8 "Kernel space optimizations".

5.1 Build the Arm Trusted Firmware

```
$ CROSS_COMPILE=aarch64-linux-gnu- make PLAT=<plat_name> bl31
```

Where `<plat_name>` can have the following values: `imx8mn`, `imx8mm`, `imx8mp`, or `imx93`.

The generated binary is located in the `build/<plat_name>/release/` directory.

5.2 Build the U-Boot

1. Copy `bl31.bin` from ATF (`build/<plat_name>/release/`) to `imx-mkimage/<platform>/`.

2. Copy all `lpddr4*` files from `firmware/ddr/synopsys/` of the `firmware-imx` package to `imx-mkimage/<platform>/`.
3. **[Only for i.MX 93]** Copy the image of the ELE firmware container `mx93a0-ahab-container.img` of the `firmware-sentinel` to `imx-mkimage/iMX9/`.
4. Compile the U-Boot.

```
$ cd uboot-imx
$ make distclean
$ ARCH=arm CROSS_COMPILE=aarch64-linux-gnu- make <defconfig_file>
$ CROSS_COMPILE=aarch64-linux-gnu- make -j $(nproc --all)
```

To build U-Boot without Falcon support (default boot mode to check that everything compiles), use the following `<defconfig_file>`.

- `imx8mm_evk_defconfig` for i.MX 8MM
- `imx8mn_evk_defconfig` for i.MX 8MN
- `imx8mp_evk_defconfig` for i.MX 8MP
- `imx93_11x11_evk_defconfig` for i.MX 93

For the Falcon mode, the `<defconfig_file>` is (use this `defconfig` file only after Section 7.3 "Falcon mode implementation").

- `imx8mm_evk_falcon_defconfig` for i.MX 8MM
- `imx8mn_evk_falcon_defconfig` for i.MX 8MN
- `imx8mp_evk_falcon_defconfig` for i.MX 8MP
- `imx93_11x11_evk_falcon_defconfig` for i.MX 93

5. Copy `u-boot*.bin` and `spl/u-boot-spl*.bin` into `imx-mkimage/<platform>/`.
6. Copy `imx8mm-evk.dtb` (for i.MX 8M Mini LPDDR4 EVK) or `imx8mn-evk.dtb` (for i.MX 8M Nano LPDDR4 EVK) or `imx8mp-evk.dtb` (for i.MX 8M Plus LPDDR4 EVK) or `imx93-11x11-evk.dtb` for (i.MX 93 11x11 LPDDR4 EVK) from `uboot-imx/arch/arm/dts/` to `imx-mkimage/<platform>/`.
7. Copy `mkimage` from `uboot-imx/tools/` into `imx-mkimage/<platform>/`, renaming into `mkimage_uboot`.

```
$ cp uboot-imx/tools/mkimage imx-mkimage/<platform>/mkimage_uboot
```

8. Generate the complete U-Boot image: `flash.bin`.

```
$ cd imx-mkimage
# for i.MX 8M*
$ make SOC=<plat_name> flash_evk
# for i.MX 93
$ make SOC=iMX9 flash_singleboot
```

Where `<plat_name>` can take the following values: `iMX8MM`, `iMX8MN`, `iMX8MP`.

5.3 Build the Linux kernel

```
$ cd linux-imx
$ ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- make imx_v8_defconfig
$ ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- make -j $(nproc --all) all
```

The resulted binary Image is located in `arch/arm64/boot` directory.

6 Write the binaries on the SD card

To check that the build is correct, write the resulted binaries on the SD card and boot the board.

- **Write the U-Boot image**:

```
$ sudo dd if=flash.bin of=/dev/sd<x> bs=1k seek=<offset> conv=fsync
```

Where `<offset>` is:

- 32 – for i.MX 8M Nano, i.MX 8M Plus and i.MX 93
- 33 – for i.MX 8M Mini

- **Write the Linux Kernel**:

```
$ sudo mount /dev/sd<x>1 /mnt
$ cp Image /mnt
$ umount /mnt
```

7 Bootloader optimizations

This chapter includes the following information.

- Section 7.1 "Default boot mode"
- Section 7.2 "Falcon mode"
- Section 7.3 "Falcon mode implementation"
- Section 7.4 "Memory map"
- Section 7.5 "Function calls during the Falcon mode"

7.1 Default boot mode

Figure 1 describes the default boot sequence. After power on or reset, i.MX 8M executes the **BootROM** (the primary program loader), stored in its read-only memory (ROM).

BootROM configures the system-on-chip (SoC) by performing basic peripheral initializations such as Phase Locked Loops (PLLs), clock configurations, memory initialization (SRAM). Then it finds a boot device from where it loads a bootloader image, which can include the following components: U-Boot SPL, ATF, U-Boot, and so on.

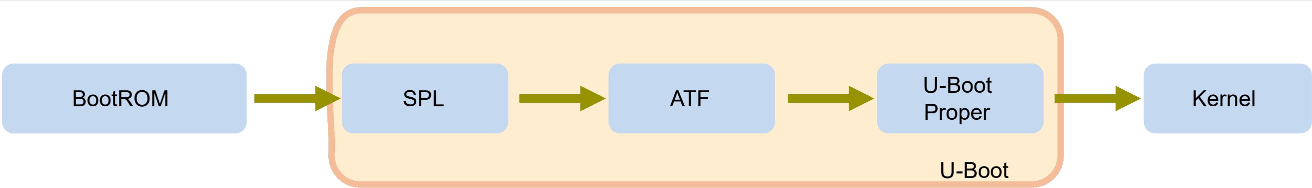


Figure 1. Default boot sequence

A typical U-Boot image does not fit inside the internal SRAM and therefore is split into two parts: **secondary program loader (SPL)** and **U-Boot proper**.

SPL is the first stage of the bootloader, a smaller preloader that shares sources as U-Boot, but with a minimal set of code that fits into SRAM. SPL is loaded into SRAM. It configures and initializes some peripherals and, most importantly, dynamic random-access memory (DRAM). Later, it loads the ATF and U-Boot proper into the DRAM. The final step is to jump to ATF, which will, in turn, jump to U-Boot proper.

Arm Trusted Firmware (ATF), included recently in the i.MX8* family, provides a reference trusted code base for the Armv8 architecture. It implements various Arm interface standards, including the Power State Coordination Interface (PSCI). The binary is typically included in the bootloader binary. It is started in the early stages of U-Boot. Without ATF, the kernel cannot set up the services, which must be executed in the Secure World environment.

U-Boot proper is the second stage bootloader. It offers a flexible way to load and start the Linux kernel. Also, it provides a minimal set of tools to interact with the hardware on the board via a command-line interface. It runs from DRAM and initializes the additional hardware devices. For example, network, USB, and DSI/CSI. Then, it loads and prepares the device tree (FDT). The main task handled by the U-Boot is the loading and starting of the kernel image itself.

Linux kernel runs from DRAM and takes over the system completely. The U-Boot has no longer control over the system from this point onward.

7.2 Falcon mode

Falcon mode is a feature in U-Boot that enables fast booting by allowing SPL to start the Linux kernel. It completely skips the U-Boot loading and initialization, with the effect of reducing the time spent in the bootloader.

Figure 2 illustrates the Falcon mode booting sequence.

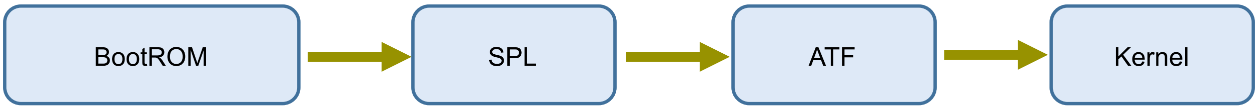


Figure 2. Falcon Mode boot sequence

To implement this mode, perform the following actions:

- Activate some specific configurations for Falcon.
- Prepare the Flattened Device Tree (FDT) in advance.
- Configure ATF to jump to kernel.
- Generate the kernel flattened uImage tree (FIT) image containing the ATF and the kernel image.

7.3 Falcon mode implementation

For ease of implementation, a series of patches has been prepared for enabling the Falcon mode.

Download the associated software `AN14093SW.zip` to get the patches, and perform the following steps.

1. **Apply the U-Boot patch**:

```
$ cd uboot-imx
$ git am 0001-Enable-Fast-Boot-on-i.MX-8M-Family-and-i.MX-93.patch
```

This patch creates the Falcon configuration files for each platform (i.MX 8M and i.MX 93), which can be found in the `uboot-imx/configs/` directory, under the name: `<board>_falcon_defconfig`. The configuration files are based on the default ones `<board>_defconfig`, to which Falcon support is added as follows.

Enabled parameters [=y]

- `CONFIG_SPL_SERIAL`
- `CONFIG_CMD_SPL` — Enables spl export command in U-Boot; required for step 15.
- `CONFIG_SPL_MMC` — Enables SPL to read from MMC using the SPL MMC API.
- `CONFIG_MMC_BROKEN_CD` **[only for i.MX 93]**
- `CONFIG_SPL_FS_FAT` — Enables SPL to read from FAT partition.
- `CONFIG_SPL_LOAD_FIT`
- `CONFIG_FIT`
- `CONFIG_SPL_OS_BOOT` — Activates the Falcon mode.

- `CONFIG_ SPL_MMC_IO_VOLTAGE` and `CONFIG_SPL_MMC_UHS_SUPPORT` — Enable MMC high speed transfer for SPL, used to reduce the loading time for the Kernel image (***not functional for i.MX 8MM** since SPL DM is not supported due to OCRAM size limitation).
- `CONFIG_LTO` **[only for i.MX 8MN]** — Reduces the binary size by adding link-time optimizations. Required on i.MX 8M Nano to ensure the SPL image with FAT support fits.

Disabled parameter [=n]

`CONFIG_SPL_BOOTROM_SUPPORT`

Set parameters

`CONFIG_SYS_SPL_ARGS_ADDR`

With:

- `0x43000000` for i.MX 8MN, i.MX 8MM and i.MX 8MP
- `0x83000000` for i.MX 93

`CONFIG_SPL_FS_LOAD_PAYLOAD_NAME` with `u-boot.itb`

`CONFIG_SPL_FS_LOAD_KERNEL_NAME` with `Image.itb`

`CONFIG_SPL_FS_LOAD_ARGS_NAME` with:

- `imx8mm-evk-falcon.dtb` for i.MX 8MM
- `imx8mn-evk-falcon.dtb` for i.MX 8MN
- `imx8mp-evk-falcon.dtb` for i.MX 8MP
- `imx93-11x11-evk-falcon.dtb` for i.MX 93

`CONFIG_CMD_SPL_WRITE_SIZE` with `0xC000`

`CONFIG_FIT_EXTERNAL_OFFSET=0x3000` **[only for i.MX 93]**

In addition, the patch implements the `spl_start_uboot()` function, located in `uboot-imx/board/freescale/<board>/spl.c`, where `<board>` is: `imx8mm_evk`, `imx8mn_evk`, `imx8mp_evk` or `imx93_evk`. This function checks if SPL should start the kernel or U-Boot. If the ‘c’ key is pressed during boot, the function returns 1, meaning that U-Boot must be started. Otherwise, SPL should start the kernel. To bring it up in the operational state in which Ethernet MAC can interact with PHY, the PHY must be reset from SPL for i.MX 8M Family. This is also added when applying the U-Boot patch. The PHY is reset in the `board_init_r()` function, located in the `uboot-imx/common/spl/spl.c` file.

2. **Apply the ATF patch**:

```
$ cd imx-atf
$ git am 0001-Add-support-to-jump-to-Kernel-directly-from-ATF.patch
```

The patch adds support for jumping directly to the kernel. Since ATF does not support to jump directly to kernel on NXP platforms, the FDT address must be passed as an argument, in `bl31_early_platform_setup2()` function, located in `imx-atf/plat/imx/imx8m/<board>/<board>_bl31_setup.c` for i.MX8M Family and `imx-atf/plat/imx/imx93/imx93_bl31_setup.c` for i.MX93.

3. **Apply the mkimage patch**:

```
$ cd imx-mkimage
$ git am 0001-Add-scripts-for-Fast-Boot-implementation-for-i.MX8M-.patch
```

This patch adds the `“os”` property to `uboot-1` node of the U-Boot FIT image source (`u-boot.its`) . This property is required when loading U-Boot (the case when `spl_start_uboot()` returns 1) while Falcon Mode is enabled. Otherwise, the U-Boot fails to boot.

In addition, the patch adds the script, which generates the U-Boot FIT image for i.MX 93 (since in this version does not exist): `imx-mkimage/iMX9/mkimage_fit_atf.sh`.

The second script added by this patch is the one used to generate the kernel FIT image (ATF + kernel) - needed for the Falcon mode implementation. This script is used for both i.MX 8M Family and i.MX 93.

4. Build the ATF as stated in Section 5.1. Copy the modified ATF binary into `imx-mkimage/<platform>/`.

5. Build the bootloader image as stated in Section 5.2 – Falcon Mode. Write the resulted U-Boot binary according to Section 6.
6. **[Only for i.MX93]** Generate the U-Boot FIT image. When building the `flash.bin` image, the `u-boot.itb` FIT image is not built automatically for i.MX93, since there is no script that generates it.

```
$ cd imx-mkimage/iMX9
$ DEK_BLOB_LOAD_ADDR=0x80400000 TEE_LOAD_ADDR=0x96000000
ATF_LOAD_ADDR=0x204e0000 ./mkimage_fit_atf.sh imx93-11x11-evk.dtb > u-boot.its
$ ./mkimage_uboot -E -p 0x3000 -f u-boot.its u-boot.itb
```

7. Copy the `u-boot.itb` binary located in `imx-mkimage/<platform>` on the first (FAT) partition of the SD card.
8. Before building the Linux kernel, you may want to optimize it according to Section 8.2 "Remove the unnecessary drivers and file systems". Build the Linux kernel according to Section 5.3 "Build the Linux kernel".
9. Generate the kernel FIT Image.
   The FIT image contains the ATF and the kernel Image. This is loaded during Falcon mode by SPL. Since SPL does not load the ATF image in the Falcon mode, the ATF must be included into the FIT image.

   To prepare the FIT image (`Image.itb`), the `mkimage_fit_atf_kernel.sh` script is used.
   Copy the kernel Image to the `imx-mkimage/<platform>/` directory:

```
$ cp linux-imx/arch/arm64/boot/Image imx-mkimage/<platform>
```

   Generate the FIT image:

   - For i.MX8M

```
$ cd imx-mkimage/iMX8M
# for i.MX8MM
$ ATF_LOAD_ADDR=0x00920000 KERNEL_LOAD_ADDR=0x40200000 ../mkimage_fit_atf_kernel.sh
 > Image.its
# for i.MX8MN
$ ATF_LOAD_ADDR=0x00960000 KERNEL_LOAD_ADDR=0x40200000 ../mkimage_fit_atf_kernel.sh
 > Image.its
# for i.MX8MP
$ ATF_LOAD_ADDR=0x00970000 KERNEL_LOAD_ADDR=0x40200000 ../mkimage_fit_atf_kernel.sh
 > Image.its

# To generate the FIT binary run:
$ ./mkimage_uboot -E -p 0x3000 -f Image.its Image.itb
```

   - For i.MX93

```
$ cd imx-mkimage/iMX9
$ ATF_LOAD_ADDR=0x204e0000 KERNEL_LOAD_ADDR=0x80200000 ../
mkimage_fit_atf_kernel.sh > Image.its

# To generate the FIT binary run:
$ ./mkimage_uboot -E -p 0x3000 -f Image.its Image.itb
```

10. Copy the resulted `Image.itb` file to the first (FAT) partition of the SD card.
11. Prepare the Flattened Device Tree and write it on the SD card.
    When booting in Falcon Mode, a key step is to prepare the device tree. Usually, U-Boot does FDT fixups when booting Linux. It means that to the initial device tree, U-Boot adds the kernel arguments and the memory node, among other modifications.

    These arguments can be found in one of the configuration files: `uboot-imx/configs/<board>_evk.h`, under the name `bootargs`. They specify console parameters and tell the kernel where to find the root file system. Where `<board>` is: `imx8mm`, `imx8mn`, `imx8mp` or `imx93`.

    There are two methods of generating the Flattened Device Tree:

    - **Method 1:** By manually adding the required fixups to the device tree

- **Method 2:** By letting U-Boot to do the fixups and save the resulted device tree

Method 2 is more general and requires less knowledge, but it is also lengthier and with several other steps.

Method 1: You can try generating the FDT manually, by adding the bootargs and the memory node to the kernel device tree. For example, for i.MX 93 create the `imx93-11x11-evk-falcon.dts` file in `linux-imx/arch/arm64/boot/dts/freescale` and add the code lines from below. The memory node is copied from the U-Boot's DTS. The included device tree can be changed according to your use case. In this case, we are using the default kernel device tree.

```
#include "imx93-11x11-evk.dts"

 / {
     memory {
         reg = <0x00 0x80000000 0x00 0x80000000>;
         device_type = "memory";
     };

     chosen {
         bootargs = "console=ttyLP0,115200 earlycon root=/dev/mmcblk1p2
 rootwait rw";
     };
 };
```

Recompile the kernel to generate the associated device tree binary and copy the resulted `imx93-11x11-evk-falcon.dtb` file to the first partition of the SD card.
If you chose the first method, the next step is to boot the board and the Falcon mode should be functional.

Method 2: FDT can be prepared by using a `spl export` command in the U-Boot stage. To enter in U-Boot, keep the **C** key pressed. The command is equivalent to running `bootm` until the device tree fixup is done. The device tree in memory is the one needed for the Falcon mode. This image has to be saved to the SD card boot partition.

[Prerequisites]

a. Build the kernel legacy `uImage` file from Image.
   `uImage` is a special image file that adds a 64-byte header before the actual kernel Image, where loader information is specified (load address, entry point, OS type, and so on).
   This type of image is needed by the `spl` command, to generate the Flattened Device Tree.

   - **For i.MX 8M**

```
$ cd linux-imx/arch/arm64/boot
$ mkimage -A arm -O linux -T kernel -C none -a 0x43FFFFC0
-e 0x44000000 -n "Linux kernel" -d Image uImage
Image Name: Linux kernel
Created: Wed Jul 26 14:12:09 2023
Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 31072768 Bytes = 30344.50 KiB = 29.63 MiB
Load Address: 43ffffc0
Entry Point: 44000000
```

   - **For i.MX 93**

```
$ cd linux-imx/arch/arm64/boot
$ mkimage -A arm -O linux -T kernel -C none -a 0x83FFFFC0
-e 0x84000000 -n "Linux kernel" -d Image uImage
Image Name: Linux kernel
Created: Wed Jul 26 14:14:09 2023
Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 31072768 Bytes = 30344.50 KiB = 29.63 MiB
Load Address: 83ffffc0
Entry Point: 84000000
```


Where:

- A [Architecture]: To set architecture.
- O [os]: To set the operating system.
- T [image type]: To set the image type.
- C [compression type]: To set the compression type.
- n [image name]: To set image name to **image name**.
- d [image data file]: To use image data from an **image data file**.
- a [load address]: To set the load address with a hex number.
- e [entry point]: To set the entry point with a hex number.

b. Copy the kernel uImage to the EXT2 partition of the SD card.

```
$ sudo mount /dev/sd<x>2 /mnt
$ sudo mkdir -p /mnt/home/root/.falcon
$ sudo cp uImage /mnt/home/root/.falcon
$ sudo umount /mnt
```

To prepare the FDT using `spl export` command, perform the following steps.

i. Boot the board into U-Boot and stop it right before entering in the autoboot sequence. To enter in U-Boot, the 'c' key must be pressed during boot. At this point, Falcon Mode fails since there is no prepared FDT for Linux kernel on the SD card.

ii. **[Optional]** If you need a different FDT from the default one, run the following command first. The file must be on the FAT partition on the SD.

```
u-boot=> setenv fdtfile <file_name>.dtb
```

iii. Load the FDT into RAM.

```
u-boot=> run loadfdt
43801 bytes read in 15 ms (2.8 MiB/s)
```

iv. Load the kernel `uImage` into RAM.

```
u-boot=> ext2load mmc 1:2 ${loadaddr} /home/root/.falcon/uImage
31072832 bytes read in 387 ms (76.6 MiB/s)
```

v. If you require the kernel boot-time optimizations as well, run the commands from Section 8.1 "Add quiet", step 2, before the next step.

vi. Load the kernel boot arguments.

```
u-boot=> run mmcargs
```

vii. Prepare FDT (example for i.MX 93).

```
u-boot=> spl export fdt ${loadaddr} - ${fdt_addr_r}
## Booting kernel from Legacy Image at 80400000 ...
   Image Name:   Linux kernel
   Created:      2023-07-19   6:57:40 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    31072768 Bytes = 29.6 MiB
   Load Address: 83ffffc0
   Entry Point:  84000000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 83000000
   Booting using the fdt blob at 0x83000000
Working FDT set to 83000000
   Loading Kernel Image
   Using Device Tree in place at 0000000083000000, end 000000008300db18
Working FDT set to 83000000
subcommand failed (err=-1)
```

```
subcommand failed (err=-1)
   Using Device Tree in place at 0000000083000000, end 0000000083010b18
Working FDT set to 83000000
Argument image is now in RAM: 0x0000000083000000
```

Note: *The difference between the start and the end addresses in **bold** above is the size of the patched FDT in memory. Copy the resulted FDT from RAM to the FAT partition of the SD card specifying the right copy size as the last parameter. In the example output above, that would be* `0x83010b18 - 0x83000000 = 0x10b18`.

```
# for i.MX 93
u-boot=> fatwrite mmc ${mmcdev}:${mmcpart} ${fdt_addr_r}  imx93-11x11-
evk-falcon.dtb 0x10b18
```

Note: *The name of the saved FDT must match the name set in the CONFIG_SPL_FS_LOAD_ARGS_NAME variable in Step 1 from Section 7.3 "Falcon mode implementation". Otherwise, the SPL does not load the device tree in the DRAM and the board fails to boot.*

12. After reboot, by default, the board will boot in Falcon Mode.

7.4 Memory map

Figure 3 is the memory map during Falcon Mode for i.MX93.

BootROM loads SPL and the SPL runs from the on-chip RAM (OCRAM - the internal processor memory). SPL initializes the dynamic RAM (DDR), loads the ATF into OCRAM, then loads the kernel device tree and the kernel image into DDR. SPL has a reserved memory space in DDR, for malloc. This area must not be overwritten while in SPL.

```
0x2051FFFF +--------------------------------+ End of OCRAM NS (640KB)
           |                                |
           |                                |
           +--------------------------------+
   31KB    |          ARM Trusted           |
           |           Firmware             |
0x204E0000 +--------------------------------+ ATF
           |                                |
           :       Secondary Program        :
  ~231KB   |            Loader              |
           |                                |
0x2049A000 +--------------------------------+ SPL
           |                                |
0x20480000 +--------------------------------+ Start of OCRAM NS

0x8FFFFFFF +--------------------------------+ End of DDR (256MB)
           |                                |
           +--------------------------------+
           |                                |
  512KB    :        SPL malloc space        :
           |                                |
0x83200000 +--------------------------------+ CONFIG_CUSTOM_SYS_SPL_MALLOC_ADDR
           :                                :
   67KB    |          Kernel DTB            |
           |                                |
0x83000000 +--------------------------------+ FDT
           |                                |
           |                                |
  ~31MB    :         Kernel Image           :
           |                                |
0x80200000 +--------------------------------+ Kernel
           |                                |
0x80000000 +--------------------------------+ Start of DDR
```

Figure 3. Memory map during Falcon Mode for i.MX93

Table 2 lists the addresses for the i.MX 8M family.

Table 2. i.MX 8M family addresses

Platform	SPL	ATF	Kernel Image	Kernel DTB
i.MX 8M Mini	0x007e1000	0x00920000	0x40200000	0x43000000
i.MX 8M Nano	0x00912000	0x00960000	0x40200000	0x43000000
i.MX 8M Plus	0x00920000	0x00970000	0x40200000	0x43000000

7.5 Function calls during the Falcon mode

Figure 4 lists the important functions called during the SPL Falcon mode.

Called from `arch/arm/cpu/armv8/start.S` Found in `arch/arm/lib/crt0_64.S` Initializations (stack + place to store GD in SRAM) for calling `board_init_f()`	**`_main`**

`board/freescale/imx8mm_evk/spl.c`
Prepares the hardware for execution from DRAM
	`board_init_f()``{`
	`...`
	`arch_cpu_init()`
	`preloader_console_init()`
	`spl_dram_init()`
	`...}`

`common/spl/spl.c`	**`board_init_r()`**`{`
	`... initializations ...`
	`boot_from_devices``(...)`
	`spl_load_image``(...)`
	`...`
	`spl_board_prepare_for_boot``()`
	`jump_to_image_no_args``(...)`
	`}`

`common/spl/spl_mmc.c`	**`spl_mmc_load_image`**`(...)` **`spl_mmc_load`**`(...){` `case MMCSD_MODE_FS:` **`spl_mmc_do_fs_boot`**`(...)`
`common/spl/spl_fat.c`	**`spl_load_image_fat_os`**`(...)` `defaults: load FDT, load Kernel FIT` **`spl_load_image_fat`**`(...)`
`common/spl/spl_fit.c`	**`spl_load_simple_fit`**`(...)`

Figure 4. Functions called during the SPL Falcon mode

8 Kernel space optimizations

This section lists the steps to Section 8.1 "Add quiet" and Section 8.2 "Remove the unnecessary drivers and file systems".

8.1 Add quiet

To reduce the kernel time by about a half, add the `quiet` argument in the kernel `bootargs`. It suppresses the debug messages during the Linux startup sequence.

Note: *The device tree must be regenerated with the new `bootargs`, using the `spl export` command.*

1. To enter in U-Boot, keep the **C** key pressed while booting.

2. Edit the `mmcargs` parameter by adding `quiet`.

```
u-boot=> edit mmcargs
edit: setenv bootargs ${jh_clk} console=${console} root=${mmcroot} quiet
u-boot=> saveenv
Saving Environment to MMC... Writing to MMC(1)... OK
```

3. Regenerate and save the device tree to the SD card as in Section 7.3 "Falcon mode implementation", step 14.

8.2 Remove the unnecessary drivers and file systems

Depending on your use case, you can slim down the kernel by removing unnecessary drivers and file systems. You can analyze kernel functions during boot with `bootgraph`, a kernel feature that allows you to graph what happens in the kernel during initialization.

To create `bootgraph`, perform the following steps:

1. Add `initcall_debug` to the kernel `bootargs`.
   a. To enter in U-Boot, keep the **C** key pressed while booting,
   b. Edit the `mmcargs` parameter by adding `initcall_debug`.

```
u-boot=> edit mmcargs
edit: setenv bootargs ${jh_clk} console=${console} root=${mmcroot} quiet
 initcall_debug
u-boot=> saveenv
Saving Environment to MMC... Writing to MMC(1)... OK
```

2. Regenerate and save the device tree to the SD card as in Section 7.3 "Falcon mode implementation", step 14.
3. Boot the board and get the kernel log.

```
root@imx8mn-lpddr4-evk:~# dmesg > boot.log
```

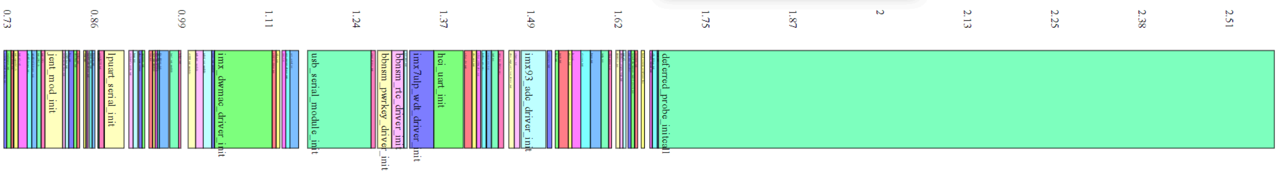
The `boot.log` file contains data like the following log. The data can be analyzed on how much time each function spend during kernel boot.

```
[2.583922] initcall deferred_probe_initcall+0x0/0xb8 returned 0 after 895357
[2.583955] calling  genpd_power_off_unused+0x0/0x98 @ 1
[2.583977] initcall genpd_power_off_unused+0x0/0x98 returned 0 after 12 usec
[2.583984] calling  genpd_debug_init+0x0/0x90 @ 1
[2.584312] initcall genpd_debug_init+0x0/0x90 returned 0 after 321 usecs
[2.584333] calling  ubi_init+0x0/0x23c @ 1
[2.584627] initcall ubi_init+0x0/0x23c returned 0 after 286 usecs
```

4. Copy the resulted `boot.log` file on the host PC. Go back on the host PC and create the graph using the following commands.

```
$ cd linux-imx/scripts
$ ./bootgraph.pl boot.log > boot.svg
```

You can obtain something like this and can analyze how the kernel boot time is used.



5. To disable a driver or a feature, update the kernel configuration.

For example, we disabled the debug from the kernel (that reduce the size of the image) and the UBI file system.

a. Run the following commands to enter the kernel `menuconfig`.

```
$ cd linux-imx
$ ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- make imx_v8_defconfig
$ ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- make menuconfig
```

From the `menuconfig`, disable `CONFIG_UBIFS_FS` and `CONFIG_DEBUG_KERNEL`, similar to Section 7.3 "Falcon mode implementation". The resulting `.config` file contains the following lines.

```
# CONFIG_UBIFS_FS is not set
# CONFIG_DEBUG_KERNEL is not set
```

b. Build the new kernel image.

```
$ ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- make all
```

c. Regenerate the kernel FIT image as in Section 7.3 "Falcon mode implementation", step 13 and copy it to an SD card boot (FAT) partition.

d. *[Optional]* If you want to use the modified kernel during normal boot, copy the new Image binary to the first SD card boot partition.

9 User space optimizations

This section lists the steps to Section "Start an application before systemd" and Section "Change the dependencies of a systemd unit".

9.1 Start an application before systemd

If required, a program can be started before `systemd`.

1. Create a script `/home/root/newinit.sh`, which starts your program before `systemd`. Below is a simple example of how to start your program before `systemd`. Replace the `echo` line with your desired application.

```
#!/bin/sh
echo “Early start” > /dev/kmsg
exec /lib/systemd/systemd
```

2. Make the script executable.

```
$ chmod +x newinit.sh
```

3. Link `/sbin/init` to your `newinit.sh` script.

```
$ ln -sf /home/root/newinit.sh /sbin/init
```

Note: *To return to the initial configuration, use the following command.*

```
$ ln -sf /lib/systemd/systemd /sbin/init
```

4. Reboot the board and check the kernel log. Searching the "`Early start`" string in `dmesg`, shows that the `newinit.sh` script is executed before the `init` process.

9.2 Change the dependencies of a systemd unit

The easiest way to reduce the time spent in user space is to reorder the sequence in which applications are run. To start the service earlier, change the dependencies with which `Systemd` operates.

On the board, open a `/lib/systemd/system/<service_name>.service` file and change the unit dependencies. For example, starting `<service_name>` before `local-fs-pre.target`.

```
[Unit]
...
Before=local-fs-pre.target
DefaultDependencies=no
```

If the command system-analyze is called with the `blame` argument `Systemd` also provides a utility called systemd-analyze, which prints the services and their starting time.

```
$ systemd-analyze blame
```

To disable a service, you can use the `systemctl disable` command. To disable some services (especially the ones `systemd` provides), use the `systemctl mask` command. However, take care when disabling services since the system can depend on them to operate properly.

10 Results

Table 3. Initial boot time measurements

	SPL			U-Boot				Kernel	
Board	BootROM	DDR initialization	SPL initializations + Load U-Boot image	U-Boot initializations (init_ sequence_ f)	U-Boot initializations (init_ sequence_ r)	Boot sequence	Kernel image load	ATF + Kernel boot until INIT process	Total time
	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
i.MX 8MN	161	241	162	363	790	2894	333	3506	8450
i.MX 8MP	162	301	175	373	1726	4181	345	3627	10890
i.MX 8MM	142	265	117	412	812	2970	396	5002	10116
i.MX 93	369	111	117	628	1172	3271	412	3090	9170

Table 4. Optimized boot time measurements

	SPL				Kernel	
Board	BootROM	DDR initialization	SPL initializations	Kernel Image Load[1]	ATF + Kernel Boot until INIT process[2]	Total time
	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
i.MX 8MN	203	240	86	376	1185	2090
i.MX 8MP	187	301	97	382	1237	2204
i.MX 8MM[3]	139	265	63	1336	2956[3]	4759
i.MX 93	374	111	89	366	1391	2330

[1] CONFIG_DEBUG_KERNEL disabled, resulting in a smaller kernel image size => decreases kernel image loading.
[2] Kernel log messages are suppressed using quiet.
[3] i.MX 8M Mini EVK does not come with an integrated Wi-Fi Module connected to the PCIe port (unlike i.MX 8M Plus). Therefore, the PCIe PHY initialization consumes time waiting for an active link. If a Wi-Fi module is attached to the PCIe interface, the kernel boot time decreases to 1215 ms, so the total boot time is 3018 ms.

11 Note about the source code in the document

The example code shown in this document has the following copyright and BSD-3-Clause license:

12 Revision history

Table 5. Revision history

Revision number	Revision Date	Description
1	09 October 2023	Initial release

13 Legal information

13.1 Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

13.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at http://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

13.3 Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

EdgeLock — is a trademark of NXP B.V.

i.MX — is a trademark of NXP B.V.

Microsoft, Azure, and ThreadX — are trademarks of the Microsoft group of companies.

Tables

Figures

Contents

Date of release: 9 October 2023
Document identifier: AN14093