2. KernelBoot

NXKR\_YoungSikYang

Exported on 2024-02-14 13:18:00

Table of Contents

1 Difference between kernel images 3

1.1 Image 3

1.2 zImage 3

1.3 uImage 3

2 How to compile the linux kernel to make Image and zImage 4

2.1 Install dependencies 4

2.2 Build (for ARM32) 4

2.3 Build (for ARM64) 4

3 How to make uImage 5

3.1 Copy the compiled kernel image to wrap around the uImage into uboot-2016.01/tools 5

3.2 Create uImage 5

4 How to boot the created image (uImage in this example) 6

4.1 Check the boot command with $ printenv 6

4.2 Load the image into the RAM of the board 6

4.3 Run the uImage in the target board 7

5 booti, bootz 8

6 How to upload data into the eMMC using fastboot 9

6.1 File system 9

6.1.1 How to check if boot.img containts the kernel Image 9

6.2 partmap\_emmc.txt 9

6.3 Upload 10

# Difference between kernel images

## Image

* **Image** refers to the uncompressed, raw binary image of the Linux kernel. This is the most basic form of the kernel image, directly produced by the kernel compilation process. However, this raw Image format is not directly used in most embedded systems due to its size and lack of metadata for bootloaders.

## zImage

* **zImage** is a compressed version of the Linux kernel(ARM32), built directly from the kernel source. The kernel is compressed to reduce its size, which is crucial for embedded systems with limited storage space. The zImage includes a small decompression stub at the beginning of the image, which is executed by the bootloader and unpacks the kernel into memory before it starts.

## uImage

* **uImage** is a format specific to the U-Boot bootloader, widely used in embedded systems. It wraps around the kernel image (which can be an Image, zImage, or even an initramfs image) with a header that includes metadata like the image's size, load address, entry point, and compression type. This metadata is used for the bootloader to correctly load and execute the kernel image. The uImage format is designed to be versatile, supporting different types of payloads and compression, making it suitable for a wide range of embedded devices.

# How to compile the linux kernel to make Image and zImage

## Install dependencies

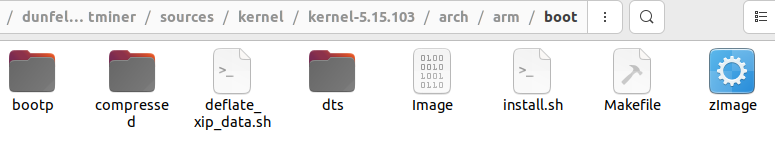
|  |
| --- |
| sudo apt-get install make build-essential libncurses-dev bison flex libssl-dev libelf-dev |

## Build (for ARM32)

|  |
| --- |
| sudo apt-get install gcc-arm-linux-gnueabihf # Cross compiler  make menuconfig # Save menuconfig and create .config  export ARCH=arm  export CROSS\_COMPILE=arm-linux-gnueabihf-  make  # make ARCH=arm CROSS\_COMPILE=aarch64-linux-gnu- |

## Build (for ARM64)

|  |
| --- |
| sudo apt-get install gcc-aarch64-linux-gnu # Cross compiler  make menuconfig # Save menuconfig and create .config  export ARCH=arm64  export CROSS\_COMPILE=aarch64-linux-gnu-  make  # make ARCH=arm64 CROSS\_COMPILE=aarch64-linux-gnu- |



# How to make uImage

## Copy the compiled kernel image to wrap around the uImage into uboot-2016.01/tools



## Create uImage



|  |
| --- |
| ./mkimage -A arm64 -O linux -T kernel -C none -a 0x40080000 -e 0x40080000 -d Image uImage |

# How to boot the created image (uImage in this example)

## Check the boot command with $ printenv

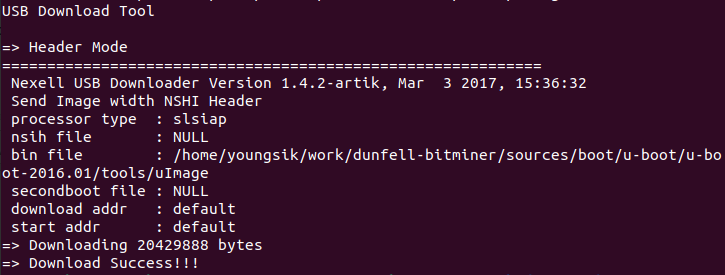


This shows the kernel image is loaded into 0x40080000 and the DTB into 0x49000000

## Load the image into the RAM of the board

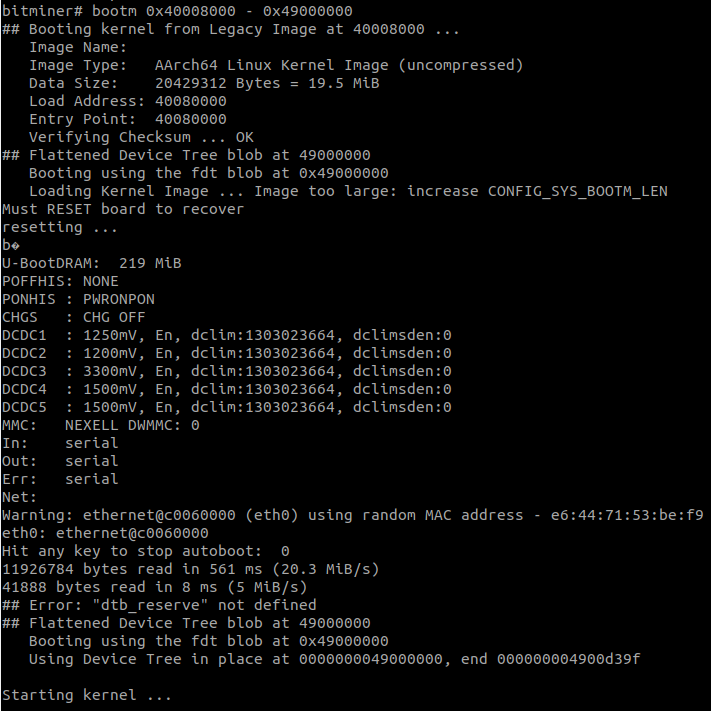
|  |
| --- |
| udown 0x40008000 # Enter **this** in minicom to prepare to load the image into the RAM of the board  ./usb-downloader -t slsiap -f ~/work/dunfell-bitminer/sources/boot/u-boot/u-boot-2016.01/tools/uImage  udown 0x49000000 # **for** the DTB  ./usb-downloader -t slsiap -f ~/work/dunfell-bitminer/build\_bitminer\_bitminer/tmp/deploy/images/s5p6818/s5p6818-bitminer-rev01.dtb |





## Run the uImage in the target board

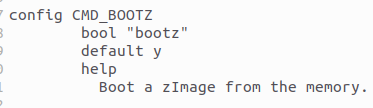
|  |
| --- |
| bootm 0x40008000 - 0x49000000 |



# booti, bootz

* booti: The default boot command of this target board is booti to boot an uncompressed Image. (e.g. booti 0x40008000 - 0x49000000)
* bootz: bootz is disabled by default and should be enabled if it needs to be used. (e.g. bootz 0x40008000 - 0x49000000)

Add this in common/kconfig to enable bootz



# How to upload data into the eMMC using fastboot

## File system



* **Kernel Image (**Image**):** This is the compiled Linux kernel. It's the core of the operating system that manages the hardware and allows other programs to run.
* **Device Tree Blob (**\*.dtb**):** This file describes the hardware in the system so that the kernel knows what devices are present and how they are configured.
* **Ramdisk (**initrd**):** This is a temporary root file system used during the boot process before the real root filesystem is mounted.

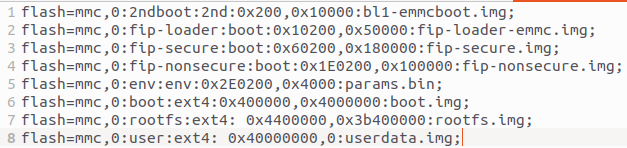
These components are combined into a flashable boot.img to get flashed onto the file system of the eMMC

### How to check if boot.img containts the kernel Image

|  |
| --- |
| **def** find\_pattern(source, pattern):  position **=** source.find(pattern)  **if** position !**=** **-**1:  **print**(f"Found the pattern at position: {position}")  **else**:  **print**("The pattern not found in the file.")    with open('boot.img', 'rb') as f:  boot\_img **=** bytearray(f.read())  with open('Image', 'rb') as f2:  kernel **=** bytearray(f2.read())  find\_pattern(boot\_img, kernel)  kernel[1] **=** 1  find\_pattern(boot\_img, kernel) |



## partmap\_emmc.txt



## Upload

|  |
| --- |
| sudo fastboot flash partmap boot-binary/partmap\_emmc.txt # Map partitions in the eMMC  # Upload  sudo fastboot flash 2ndboot boot-binary/bl1-emmcboot.img  sudo fastboot flash fip-loader boot-binary/fip-loader-emmc.img  sudo fastboot flash fip-secure boot-binary/fip-secure.img  sudo fastboot flash fip-nonsecure boot-binary/fip-nonsecure.img  sudo fastboot flash env boot-binary/params.bin  sudo fastboot flash boot boot.img  sudo fastboot flash rootfs rootfs.img  sudo fastboot flash user userdata.img |