Application Note

Linux Boot from OSPI NOR Flash Without eMMC on TI J721S2

Johnny Kim

ABSTRACT

This application note describes how to boot Linux entirely from OSPI NOR flash on the Texas Instruments J721S2 platform without relying on eMMC storage.

The demonstrated boot flow stores all boot components in OSPI NOR flash, including tiboot3.bin, tispl.bin, U-Boot, Linux kernel image, device tree blob (DTB), and the root filesystem. A UBI/UBIFS-based root filesystem is created within the OSPI flash and used as the Linux root filesystem during system startup. The implementation is validated on the J721S2 EVM using Processor SDK Linux. The application note covers OSPI flash partitioning, UBI and UBIFS volume creation, construction of a minimal root filesystem, storage of Linux artifacts within UBIFS, U-Boot configuration, and complete Linux boot from OSPI NOR flash. The resulting solution demonstrates a fully functional Linux boot chain without requiring eMMC storage, providing an alternative boot architecture for systems that require simplified storage configurations or OSPI-based deployment.

Table of Contents

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

1.1 Motivation

The Texas Instruments J721S2 device supports multiple boot media, including SD card, eMMC, OSPI NOR flash, and so on. In many production systems, OSPI NOR flash is commonly used to store early boot components such as tiboot3.bin, tispl.bin, and U-Boot. For Linux-based systems, however, the kernel image, device tree blob (DTB), and root filesystem are often stored on eMMC or other mass-storage devices due to their larger storage capacity. This results in a boot architecture that depends on multiple storage devices. In some applications, it is desirable to eliminate the dependency on eMMC and consolidate all boot components into a single OSPI NOR flash device. Such an approach simplifies storage management, reduces system complexity, and enables deployment on systems where eMMC is not available. This application note demonstrates a Linux boot solution in which all boot components, including the bootloader, kernel image, DTB, and root filesystem, are stored in OSPI NOR flash. The root filesystem is implemented using UBI and UBIFS to provide a robust flash-aware storage design.

1.2 Scope

The objective of this application note is to demonstrate and verify Linux boot from OSPI NOR flash on the J721S2 platform without relying on eMMC storage.

The following items are discussed.

- Creating a UBI/UBIFS root filesystem within OSPI NOR flash
- Building and validating a minimal Linux root filesystem
- Storing the Linux kernel image and DTB in UBIFS
- Loading the kernel image and DTB from UBIFS using U-Boot
- Booting Linux with a UBIFS root filesystem
- Booting the complete system from OSPI NOR flash

The procedures described in this document were validated on the J721S2 EVM using Processor SDK Linux.

1.3 Target Boot Architecture

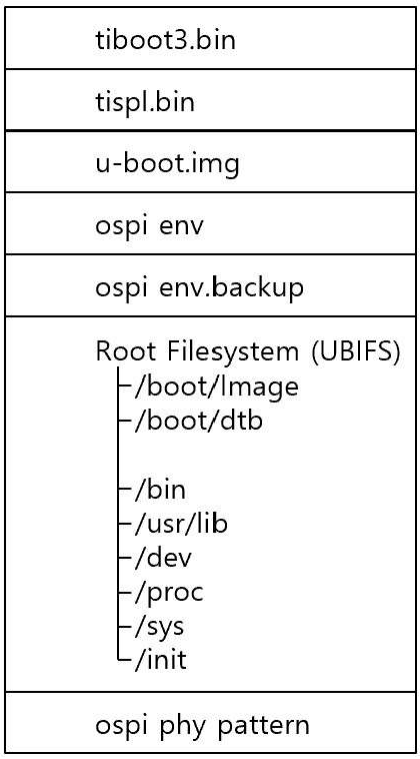

Figure 1-1. OSPI NOR Flash Layout

All software components required for Linux boot are stored in OSPI NOR flash. U-Boot loads the Linux kernel image and DTB from a UBIFS volume located in the OSPI flash, and Linux subsequently mounts the same UBIFS volume as the root filesystem. As a result, the complete boot process is performed without requiring eMMC storage.

2 Background

2.1 J721S2 Boot Flow

The J721S2 device uses a multi-stage boot process to initialize the system and load the Linux operating system.

When the device powers on, the on-chip ROM Bootloader executes first. Based on the configured boot mode, the ROM loads tiboot3.bin from the selected boot media. The tiboot3.bin image initializes essential device resources and loads tispl.bin.

The tispl.bin image performs additional system initialization, including DDR configuration and loading of system firmware components. Once initialization is complete, tispl.bin loads and starts U-Boot.

U-Boot is responsible for loading the Linux kernel image and device tree blob (DTB), preparing the Linux boot arguments, and transferring control to the Linux kernel.

The boot sequence implemented in this application note is shown below.

```
ROM → tiboot3.bin → tispl.bin → U-Boot → Linux Kernel → Linux User Space
```

The objective of this work is to store all components of this boot chain in OSPI NOR flash and eliminate any dependency on eMMC storage.

2.2 OSPI NOR Flash Overview

OSPI NOR flash is commonly used on J721S2 platforms to store bootloader components because it supports direct access by the ROM Bootloader and provides reliable non-volatile storage.

Compared to eMMC devices, OSPI NOR flash typically offers lower storage capacity but provides a simpler storage architecture for embedded systems. A single OSPI device can contain bootloader images, Linux boot artifacts, and a root filesystem if the available capacity is sufficient.

In this implementation, a 64-MB OSPI NOR flash device is used. The flash is partitioned to store the following items.

- tiboot3.bin
- tispl.bin
- u-boot.img
- U-Boot environment
- UBI/UBIFS root filesystem volume

The Linux kernel image, DTB, and root filesystem are all stored within the UBI/UBIFS partition.

2.3 UBI and UBIFS Overview

Raw flash devices such as NOR and NAND flash are fundamentally different from block storage devices such as eMMC and SSDs. Linux filesystems designed for block devices, such as ext4, are not intended to operate directly on raw flash memory.

To support reliable storage on raw flash devices, Linux provides the UBI (Unsorted Block Images) layer and the UBIFS (UBI File System).

UBI sits between the MTD subsystem and the filesystem. It manages flash-specific requirements such as logical erase block mapping, wear leveling, and bad block handling. UBIFS is a flash-aware filesystem that operates on top of UBI volumes.

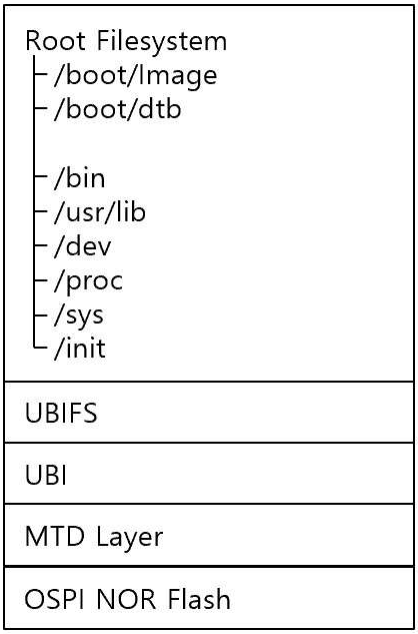

Figure 2-1. UBI/UBIFS Software Stack

Using UBI and UBIFS allows Linux to store and access files reliably on OSPI NOR flash while providing a standard filesystem interface to both U-Boot and Linux.

In this application note, a UBI volume named rootfs is created in the OSPI root filesystem partition. This volume stores the Linux kernel image, DTB, and the Linux root filesystem.

3 OSPI Flash Layout

This chapter describes the OSPI NOR flash partition layout used throughout this application note. The layout allocates dedicated regions for boot components, U-Boot environment storage, and a UBI/UBIFS-based Linux filesystem. The partition organization enables all software components required for Linux boot to reside within a single OSPI NOR flash device.

3.1 Partition Layout

The J721S2 EVM used in this application note contains a 64-MB OSPI NOR flash device. The flash is partitioned to store boot components, U-Boot environment data, and a Linux root filesystem implemented using UBI and UBIFS.

Table 3-1. Summary of the Partition Layout Used for the Implementation

Partition	Size	Purpose
ospi.tiboot3	512KB	Secondary-stage bootloader
ospi.tispl	2MB	A72 SPL
ospi.u-boot	4MB	U-Boot image
ospi.env	256KB	U-Boot environment
ospi.env.backup	256KB	U-Boot backup environment
ospi.rootfs	55.8MB	UBI/UBIFS filesystem
ospi.phypattern	256KB	PHY pattern storage

The largest partition, ospi.rootfs, is used to store the Linux kernel image, device tree blob (DTB), and Linux root filesystem. The partition is formatted as a UBI device and contains a UBIFS volume named rootfs.

3.2 Bootloader Partitions

The first portion of the OSPI NOR flash stores the software components required to initialize the system and start Linux. During power-on, the J721S2 ROM code loads tiboot3.bin from the ospi.tiboot3 partition. The tiboot3.bin image performs early platform initialization and loads tispl.bin from the ospi.tispl partition. The tispl.bin image initializes DDR memory, loads system firmware, and starts U-Boot from the ospi.u-boot partition. U-Boot then loads the Linux kernel image and DTB from the UBIFS filesystem located in the ospi.rootfs partition. The ospi.env and ospi.env.backup partitions store U-Boot environment variables used to control the boot process. The backup partition provides redundancy in the event of environment corruption.

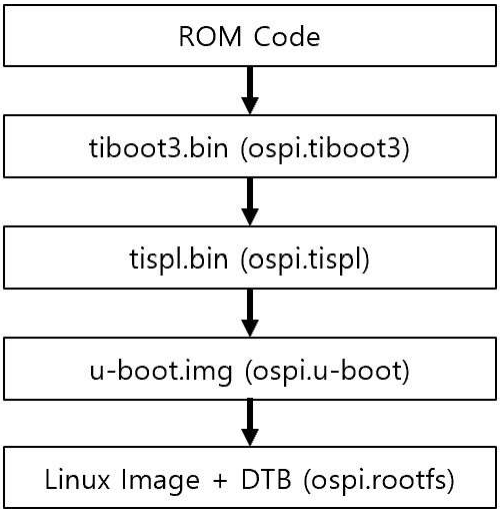

Figure 3-1. Boot Component Flow

The boot component partitions occupy only a small portion of the total flash capacity, leaving most of the OSPI NOR flash available for Linux storage.

3.3 Root Filesystem Partition

The ospi.rootfs partition is used to store all Linux-related software components required for system boot.

Unlike a conventional J721S2 Linux system that stores the kernel image, DTB, and root filesystem on eMMC, this implementation stores all Linux artifacts within a UBIFS volume located in OSPI NOR flash.

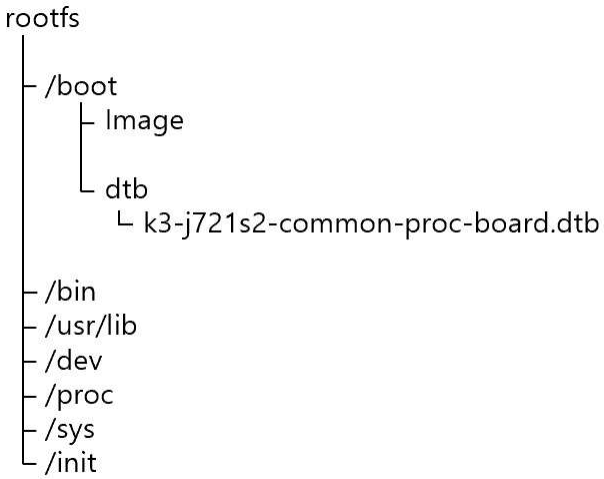

Figure 3-2. Root Filesystem Structure

The partition is formatted as a UBI device and contains a UBIFS volume named rootfs. The volume stores below items.

- Linux kernel image (Image)
- Device tree blob (*.dtb)
- Linux root filesystem
- Startup scripts and runtime libraries

Both U-Boot and Linux access this same UBIFS volume during the boot process. U-Boot loads the kernel image and DTB from the volume, while Linux subsequently mounts the volume as its root filesystem.

At the completion of the implementation described in this application note, the OSPI NOR flash contains all software components required for Linux boot.

- tiboot3.bin
- tispl.bin
- U-Boot
- Linux kernel image
- Device tree blob (DTB)
- Linux root filesystem

This architecture enables Linux boot without any dependency on SD/eMMC storage.

4 Flashing Boot Components

This chapter describes how the boot components are flashed into the OSPI NOR flash. Before flashing the images, the OSPI partition layout is verified through the Linux MTD subsystem to identify the corresponding MTD devices.

4.1 Boot Components Overview

The J721S2 boot flow implemented in this application note uses three boot software components stored in OSPI NOR flash.

- tiboot3.bin – Secondary Bootloader
- tispl.bin – A72 SPL
- u-boot.img – U-Boot

During system startup, the ROM code loads tiboot3.bin from OSPI NOR flash. The boot process then continues through tispl.bin and u-boot.img before Linux is loaded from the UBIFS filesystem stored in the root filesystem partition.

4.2 Identifying OSPI MTD Partitions

Linux exposes OSPI NOR flash partitions through the Memory Technology Device (MTD) subsystem. Each partition is assigned an MTD device that can be accessed through the /dev/mtdX interface.

The available OSPI partitions can be obtained as follows on the J721S2 EVM.

```
root@j721s2-evm:~# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00080000 00040000 "ospi.tiboot3"
mtd1: 00200000 00040000 "ospi.tispl"
mtd2: 00400000 00040000 "ospi.u-boot"
mtd3: 00040000 00040000 "ospi.env"
mtd4: 00040000 00040000 "ospi.env.backup"
mtd5: 037c0000 00040000 "ospi.rootfs"
mtd6: 00040000 00040000 "ospi.phypattern""
```

4.3 Flashing Boot Components into OSPI NOR Flash

After identifying the OSPI MTD partitions, the boot components can be flashed into OSPI NOR flash.

During development, the boot component images were obtained from the SD card boot partition used to boot the J721S2 EVM. The locations of the boot component images will be found at /run/media/BOOT-mmcblk1p1 as follows.

```
root@j721s2-evm:~# ls -l /run/media/BOOT-mmcblk1p1/
total 4906
-rwxrwx--- 1 root disk  465805 Jun 13  2026 tiboot3-j721s2-gp-evm-j721s2-evm-
k3r5-2025.01+git97201+743712b-r0_tisdk_6_edgeai_5.bin
-rwxrwx--- 1 root disk  465805 Jun 13  2026 tiboot3-j721s2-gp-evm.bin
-rwxrwx--- 1 root disk  468394 Jun 13  2026 tiboot3-j721s2-hs-evm.bin
-rwxrwx--- 1 root disk  468394 Jun 13  2026 tiboot3-j721s2-hs-fs-evm.bin
-rwxrwx--- 1 root disk  465805 Jun 13  2026 tiboot3.bin
-rwxrwx--- 1 root disk 1326915 Jun 13  2026 tispl.bin
-rwxrwx--- 1 root disk 1359931 Jun 13  2026 u-boot.img
-rwxrwx--- 1 root disk     941 Jun 13  2026 uEnv.txt
-rwxrwx--- 1 root disk      14 Jun 13  2026 version
```

Table 4-1 shows the mapping between the boot component images and their target OSPI partitions.

Table 4-1. Boot Component Flashing Target

Image	Target MTD Device	OSPI Partition
tiboot3.bin	/dev/mtd0	ospi.tiboot3
tispl.bin	/dev/mtd1	ospi.tispl
u-boot.img	/dev/mtd2	ospi.u-boot

The boot components are flashed using the Linux **flashcp** utility.

```
root@j721s2-evm:~# flashcp /run/media/BOOT-mmcblk1p1/tiboot3.bin /dev/mtd0
root@j721s2-evm:~# flashcp /run/media/BOOT-mmcblk1p1/tispl.bin   /dev/mtd1
root@j721s2-evm:~# flashcp /run/media/BOOT-mmcblk1p1/u-boot.img  /dev/mtd2
```

After flashing completes, the OSPI NOR flash contains all boot software required to reach the U-Boot prompt. The next section verifies that the flashed images can be successfully loaded when the board is configured for OSPI boot mode.

4.4 Verifying Boot Components

After flashing the boot components, the board will be reached to U-Boot command prompt with OSPI boot mode.

The following U-Boot banner was observed during validation.

```
U-Boot 2025.01-00551-g743712b9ee4b (Jul 31 2025 - 11:32:44 +0000)
...
=>
```

Successful access to the U-Boot prompt confirms that all boot components were flashed correctly and can be loaded from OSPI NOR flash.

At this stage, the complete boot software stack is available in OSPI NOR flash. The next chapter describes how to create a UBI/UBIFS-based root filesystem within the ospi.rootfs partition.

5 Creating a UBIFS Root Filesystem

As discussed in Section 2.3, Linux uses the UBI layer and UBIFS filesystem to manage raw flash devices such as OSPI NOR flash.

In this application note, the ospi.rootfs partition is converted into a UBI device and a UBIFS volume named rootfs is created. This volume is later used to store the Linux kernel image, device tree blob (DTB), and Linux root filesystem.

This chapter describes the procedure used to create and verify the UBIFS volume on the OSPI NOR flash.

5.1 Preparing the RootFS Partition

The OSPI flash layout introduced in Section 3 reserves the ospi.rootfs partition for Linux storage.

Before creating the UBI device, any existing data in the partition must be removed.

```
root@j721s2-evm:~# ubiformat /dev/mtd5
ubiformat: mtd5 (nor), size 58458112 bytes (55.7 MiB), 223 eraseblocks of 262144 bytes (256.0 KiB),
min. I/O size 16 bytes
libscan: scanning eraseblock 222 -- 100 % complete
ubiformat: 223 eraseblocks have valid erase counter, mean value is 1
ubiformat: formatting eraseblock 222 -- 100 % complete
root@j721s2-evm:~#
```

After formatting completes, the partition is ready for UBI attachment.

5.2 Creating a UBI Device

The formatted partition can be attached to the Linux UBI subsystem using the ubiattach command.

```
root@j721s2-evm:~# ubiattach /dev/ubi_ctrl -m 5
[  261.852295] ubi0: attaching mtd5
[  261.864496] ubi0: scanning is finished
[  261.885069] ubi0: attached mtd5 (name "ospi.rootfs", size 55 MiB)
[  261.891218] ubi0: PEB size: 262144 bytes (256 KiB), LEB size: 262016 bytes
[  261.904222] ubi0: min./max. I/O unit sizes: 16/256, sub-page size 16
[  261.910748] ubi0: VID header offset: 64 (aligned 64), data offset: 128
[  261.917331] ubi0: good PEBs: 223, bad PEBs: 0, corrupted PEBs: 0
[  261.923343] ubi0: user volume: 0, internal volumes: 1, max. volumes count: 128
[  261.930563] ubi0: max/mean erase counter: 6/3, WL threshold: 4096, image sequence number:
1854191452
[  261.939690] ubi0: available PEBs: 219, total reserved PEBs: 4, PEBs reserved for bad PEB
handling: 0
[  261.948828] ubi0: background thread "ubi_bgt0d" started, PID 1175
UBI device number 0, total 223 LEBs (58429568 bytes, 55.7 MiB), available 219 LEBs (57381504 bytes,
54.7 MiB), LEB size 262016 bytes (255.8 KiB)
root@j721s2-evm:~#
root@j721s2-evm:~# ubinfo
UBI version:                    1
Count of UBI devices:           1
UBI control device major/minor: 10:126
Present UBI devices:            ubi0
```

A successful attachment creates a UBI device associated with the OSPI root filesystem partition. At this stage, the storage hierarchy becomes

```
OSPI NOR Flash
      |
      +-- /dev/mtd5
                |
                +-- ubi0
```

5.3 Creating a UBIFS Volume

A UBIFS volume named *rootfs* is created within the newly attached UBI device.

```
root@j721s2-evm:~# ubimkvol /dev/ubi0 -N rootfs -m
Set volume size to 57381504
Volume ID 0, size 219 LEBs (57381504 bytes, 54.7 MiB), LEB size 262016 bytes (255.8 KiB), dynamic,
name "rootfs", alignment 1
root@j721s2-evm:~#
```

The -m option allocates all remaining available space to the volume.

The resulting storage hierarchy is as follows.

```
OSPI NOR Flash
      |
      +-- /dev/mtd5
                |
                +-- ubi0
                        |
                        +-- rootfs
```

This volume is used throughout the remainder of this application note.

5.4 Verifying UBIFS Operation

The UBIFS volume can be mounted using the Linux UBIFS driver.

Create a mount point and mount the volume.

```
root@j721s2-evm:~# mkdir -p /mnt/ospi_rootfs
root@j721s2-evm:~# mount -t ubifs -o compr=none ubi0:rootfs /mnt/ospi_rootfs
[  378.881540] UBIFS (ubi0:0): default file-system created
[  378.886938] UBIFS (ubi0:0): Mounting in unauthenticated mode
[  378.892793] UBIFS (ubi0:0): background thread "ubifs_bgt0_0" started, PID 1188
[  378.924191] UBIFS (ubi0:0): UBIFS: mounted UBI device 0, volume 0, name "rootfs"
[  378.931604] UBIFS (ubi0:0): LEB size: 262016 bytes (255 KiB), min./max. I/O unit sizes: 16
bytes/256 bytes
[  378.941248] UBIFS (ubi0:0): FS size: 54761344 bytes (52 MiB, 209 LEBs), max 219 LEBs, journal
size 2620160 bytes (2 MiB, 10 LEBs)
[  378.952889] UBIFS (ubi0:0): reserved for root: 2586513 bytes (2525 KiB)
[  378.959495] UBIFS (ubi0:0): media format: w5/r0 (latest is w5/r0), UUID 0D0FF060-F89F-4D12-
B474-1C6AA3288A8D, small LPT model
root@j721s2-evm:~#
```

The mounted filesystem can be verified using standard Linux file operations.

For example,

```
root@j721s2-evm:~# echo hello > /mnt/ospi_rootfs/test.txt
root@j721s2-evm:~# sync
root@j721s2-evm:~# cat /mnt/ospi_rootfs/test.txt
hello
root@j721s2-evm:~#
```

In addition to verification on Linux, the UBIFS volume can be also verified from the U-Boot environment. This step ensures that the filesystem created in Linux can be accessed by U-Boot before storing Linux boot artifacts.

The following commands were used.

=> ubi part ospi.rootfs

=> ubifsmount ubi0:rootfs

=> ubifsls /

Example output is below.

```
=> ubi part ospi.rootfs
k3-navss-ringacc ringacc@2b800000: Ring Accelerator probed rings:286, gp-rings[96,20] sci-dev-id:272
k3-navss-ringacc ringacc@2b800000: dma-ring-reset-quirk: disabled
jedec_spi_nor flash@0: non-uniform erase sector maps are not supported yet.
cadence_spi spi@47040000: Pattern not found. Skipping calibration
SF: Detected s28hs512t with page size 256 Bytes, erase size 256 KiB, total 64 MiB
jedec_spi_nor flash@0: Software reset enable failed: -524
cadence_spi spi@47050000: Pattern not found. Skipping calibration
SF: Detected mt25qu512a with page size 256 Bytes, erase size 64 KiB, total 64 MiB
Could not find a valid device for spi-nand0
ubi0: attaching mtd7
ubi0: scanning is finished
ubi0: attached mtd7 (name "ospi.rootfs", size 55 MiB)
ubi0: PEB size: 262144 bytes (256 KiB), LEB size: 262016 bytes
ubi0: min./max. I/O unit sizes: 16/256, sub-page size 16
ubi0: VID header offset: 64 (aligned 64), data offset: 128
ubi0: good PEBs: 223, bad PEBs: 0, corrupted PEBs: 0
ubi0: user volume: 1, internal volumes: 1, max. volumes count: 128
ubi0: max/mean erase counter: 7/3, WL threshold: 4096, image sequence number: 1854191452
ubi0: available PEBs: 0, total reserved PEBs: 223, PEBs reserved for bad PEB handling: 0
=> ubifsmount ubi0:rootfs
UBIFS (ubi0:0): recovery needed
UBIFS (ubi0:0): recovery deferred
UBIFS (ubi0:0): UBIFS: mounted UBI device 0, volume 0, name "rootfs", R/O mode
UBIFS (ubi0:0): LEB size: 262016 bytes (255 KiB), min./max. I/O unit sizes: 16 bytes/256 bytes
UBIFS (ubi0:0): FS size: 54761344 bytes (52 MiB, 209 LEBs), journal size 2620160 bytes (2 MiB, 10
LEBs)
UBIFS (ubi0:0): reserved for root: 2586513 bytes (2525 KiB)
UBIFS (ubi0:0): media format: w5/r0 (latest is w4/r0), UUID 0D0FF060-F89F-4D12-B474-1C6AA3288A8D,
small LPT model
=> ubifsls /
                6  Fri May 30 02:46:03 2025  test.txt
=>
```

The successful execution of these commands confirms that the UBI metadata and UBIFS volume created in Linux are recognized correctly by U-Boot and can be accessed before Linux boot files are stored.

Successful read and write operations confirm below.

- The OSPI NOR flash partition is accessible.
- The UBI device is functioning correctly.
- The UBIFS volume can be mounted and used from both Linux and U-Boot environments.

At this point, the rootfs volume is ready to store the Linux kernel image, DTB, and root filesystem components described in the next chapter.

6 Storing Linux Boot Artifacts in UBIFS

The UBIFS volume created in Chapter 5 was verified from both Linux and U-Boot environments. At this stage, the volume contains only the files used during filesystem validation.

To boot Linux entirely from OSPI NOR flash, the UBIFS volume must be populated with the Linux root filesystem, kernel image, and device tree blob (DTB). This chapter describes the procedure used to prepare these boot artifacts.

6.1 Mounting the UBIFS Root Filesystem Volume

First, attach the UBI device and mount the UBIFS root filesystem volume.

```
root@j721s2-evm:~# ubiattach /dev/ubi_ctrl -m 5
[   24.823337] ubi0: attaching mtd5
[   24.835461] ubi0: scanning is finished
[   24.860253] ubi0: attached mtd5 (name "ospi.rootfs", size 55 MiB)
[   24.866388] ubi0: PEB size: 262144 bytes (256 KiB), LEB size: 262016 bytes
[   24.873293] ubi0: min./max. I/O unit sizes: 16/256, sub-page size 16
[   24.879662] ubi0: VID header offset: 64 (aligned 64), data offset: 128
[   24.886203] ubi0: good PEBs: 223, bad PEBs: 0, corrupted PEBs: 0
[   24.892210] ubi0: user volume: 1, internal volumes: 1, max. volumes count: 128
[   24.899434] ubi0: max/mean erase counter: 7/3, WL threshold: 4096, image sequence number:
1854191452
[   24.908559] ubi0: available PEBs: 0, total reserved PEBs: 223, PEBs reserved for bad PEB
handling: 0
[   24.917690] ubi0: background thread "ubi_bgt0d" started, PID 1174
UBI device number 0, total 223 LEBs (58429568 bytes, 55.7 MiB), available 0 LEBs (0 bytes), LEB
size 262016 bytes (255.8 KiB)
root@j721s2-evm:~# mkdir -p /mnt/ospi_rootfs
root@j721s2-evm:~#
root@j721s2-evm:~# mount -t ubifs -o compr=none ubi0:rootfs /mnt/ospi_rootfs
[   36.600493] UBIFS (ubi0:0): Mounting in unauthenticated mode
[   36.606414] UBIFS (ubi0:0): background thread "ubifs_bgt0_0" started, PID 1180
[   36.621952] UBIFS (ubi0:0): recovery needed
[   38.692922] UBIFS (ubi0:0): recovery completed
[   38.697465] UBIFS (ubi0:0): UBIFS: mounted UBI device 0, volume 0, name "rootfs"
[   38.705150] UBIFS (ubi0:0): LEB size: 262016 bytes (255 KiB), min./max. I/O unit sizes: 16
bytes/256 bytes
[   38.714909] UBIFS (ubi0:0): FS size: 54761344 bytes (52 MiB, 209 LEBs), max 219 LEBs, journal
size 2620160 bytes (2 MiB, 10 LEBs)
[   38.726619] UBIFS (ubi0:0): reserved for root: 2586513 bytes (2525 KiB)
[   38.733285] UBIFS (ubi0:0): media format: w5/r0 (latest is w5/r0), UUID 0D0FF060-F89F-4D12-
B474-1C6AA3288A8D, small LPT model
root@j721s2-evm:~#
```

6.2 Creating the Root Filesystem Structure with BusyBox

Create the minimal directory structure required by the root filesystem.

```
mkdir -p /mnt/ospi_rootfs/bin
mkdir -p /mnt/ospi_rootfs/usr/lib
mkdir -p /mnt/ospi_rootfs/dev
mkdir -p /mnt/ospi_rootfs/proc
mkdir -p /mnt/ospi_rootfs/sys
```

Copy BusyBox into the root filesystem.

```
cp /usr/bin/busybox /mnt/ospi_rootfs/bin/
```

Copy the required shared libraries.

```
cp /usr/lib/libm.so.6 /mnt/ospi_rootfs/usr/lib/
cp /usr/lib/libc.so.6 /mnt/ospi_rootfs/usr/lib/
cp /usr/lib/ld-linux-aarch64.so.1 /mnt/ospi_rootfs/usr/lib/
```

These libraries can be identified using *ldd* command.

```
ldd /usr/bin/busybox
```

Note

This application note uses a BusyBox-based minimal root filesystem to reduce complexity and focus on the essential Linux boot process.

6.3 Verifying BusyBox Execution

Verify that BusyBox can execute using only the libraries stored in the root filesystem as follows.

```
root@j721s2-evm:~# LD_LIBRARY_PATH=/mnt/ospi_rootfs/usr/lib /mnt/ospi_rootfs/bin/busybox --help
BusyBox v1.36.1 () multi-call binary.
BusyBox is copyrighted by many authors between 1998-2015.
Licensed under GPLv2. See source distribution for detailed
copyright notices.
:
:
root@j721s2-evm:~# LD_LIBRARY_PATH=/mnt/ospi_rootfs/usr/lib /mnt/ospi_rootfs/bin/busybox echo
"busybox test"
busybox test
```

This confirms that BusyBox and all required shared libraries were loaded successfully.

6.4 Creating BusyBox Command Links

Symbolic links can be used to execute basic Linux commands seamlessly.

```
cd /mnt/ospi_rootfs/bin

ln -sf busybox sh
ln -sf busybox mount
ln -sf busybox echo
ln -sf busybox cat
ln -sf busybox ls
ln -sf busybox mkdir
ln -sf busybox mknod
ln -sf busybox ps
ln -sf busybox dmesg
ln -sf busybox sleep
ln -sf busybox umount
ln -sf busybox uname
ln -sf busybox pwd
```

6.5 Creating the init Script

Create the root filesystem initialization script.

```
cat > /mnt/ospi_rootfs/init << 'EOF'
#!/bin/sh

echo ""
echo "Starting OSPI RootFS Initialization..."
echo ""

mount -t proc proc /proc
mount -t sysfs sysfs /sys
mount -t devtmpfs devtmpfs /dev 2>/dev/null

echo ""
echo "Booted from OSPI UBIFS RootFS"
echo ""

exec /bin/sh
EOF
```

Make the script executable.

```
chmod +x /mnt/ospi_rootfs/init
sync
```

6.6 Testing the Root Filesystem Using chroot

Verify that the root filesystem can operate independently by using *chroot*.

```
root@j721s2-evm:~# chroot /mnt/ospi_rootfs /bin/sh
@j721s2-evm:/# pwd
/
@j721s2-evm:/# ls
bin       boot      dev       init      proc      sys       test.txt  usr
@j721s2-evm:/# /bin/busybox echo "chroot busybox OK"
chroot busybox OK
@j721s2-evm:/# /bin/busybox echo "chroot busybox OK"
chroot busybox OK
@j721s2-evm:/# cat /init
#!/bin/sh

echo ""
echo "Starting OSPI RootFS Initialization..."
echo ""

mount -t proc proc /proc
mount -t sysfs sysfs /sys
mount -t devtmpfs devtmpfs /dev

echo ""
echo "Booted from OSPI UBIFS RootFS"
echo ""

exec /bin/sh
@j721s2-evm:/#
```

6.7 Copying the Linux Kernel Image and DTB

Create the boot directories, and copy the kernel image and device tree.

```
root@j721s2-evm:~# mkdir -p /mnt/ospi_rootfs/boot
root@j721s2-evm:~# mkdir -p /mnt/ospi_rootfs/boot/dtb
root@j721s2-evm:~# cp /boot/Image /mnt/ospi_rootfs/boot/
root@j721s2-evm:~# cp /boot/dtb/ti/k3-j721s2-common-proc-board.dtb /mnt/ospi_rootfs/boot/dtb/
root@j721s2-evm:~# sync
```

7 Configuring Linux Boot from UBIFS

After creating the minimal root filesystem and copying the Linux kernel image and device tree into the UBIFS volume, the next step is to configure U-Boot to load these files and boot Linux directly from OSPI NOR Flash.

The following sections describe how to configure the boot arguments, load the kernel and device tree from UBIFS, and verify successful Linux boot.

7.1 Configuring Boot Arguments

Linux requires boot arguments to identify the root filesystem location and the initial process to execute.

Configure the boot arguments as follows.

```
setenv bootargs 'console=ttyS2,115200n8 earlycon=ns16550a,mmio32,0x02880000 ubi.mtd=ospi.rootfs
root=ubi0:rootfs rootfstype=ubifs rw init=/init'
```

The key parameters are listed below.

Table 7-1. Key Parameters

Parameter	Description
console=ttyS2,115200n8	Linux console output
earlycon=ns16550a,...	Early boot debug messages
ubi.mtd=ospi.rootfs	Attach the UBI device from the OSPI rootfs partition
root=ubi0:rootfs	Use the rootfs UBI volume as the root filesystem
rootfstype=ubifs	root filesystem type
rw	Mount root filesystem read/write
init=/init	Execute the custom initialization script

7.2 Loading the Kernel and DTB from UBIFS

Attach the OSPI rootfs partition and mount the UBIFS volume.

```
ubi part ospi.rootfs
ubifsmount ubi0:rootfs
```

Load the Linux kernel image and load the device tree.

```
ubifsload ${loadaddr} /boot/Image
ubifsload ${fdtaddr} /boot/dtb/k3-j721s2-common-proc-board.dtb
```

7.3 Testing Linux Boot

Boot Linux using the loaded kernel image and device tree.

```
booti ${loadaddr} - ${fdtaddr}
```

During boot, Linux should attach the UBI device and mount the UBIFS root filesystem.

Key messages include below.

```
ubi0: attached mtd5 (name "ospi.rootfs", size 55 MiB)

UBIFS (ubi0:0): UBIFS: mounted UBI device 0, volume 0, name "rootfs"

VFS: Mounted root (ubifs filesystem)

Run /init as init process
```

After the kernel launches the initialization script, the following messages should appear as below.

```
Starting OSPI RootFS Initialization...

Booted from OSPI UBIFS RootFS
```

Finally, the BusyBox shell should become available.

```
~ #
```

This confirms that Linux has booted successfully using the root filesystem stored entirely inside OSPI NOR Flash.

- Linux kernel was loaded from OSPI NOR Flash
- Device tree was loaded from OSPI NOR Flash
- Root filesystem was mounted from UBIFS
- /init executed successfully
- BusyBox shell started successfully

These observations demonstrate that Linux boot no longer depends on eMMC or SD storage devices.

7.4 Creating a Reusable Boot Command

Once boot has been verified, the commands can be combined into a reusable boot sequence.

```
setenv ospi_boot '
ubi part ospi.rootfs;
ubifsmount ubi0:rootfs;
ubifsload ${loadaddr} /boot/Image;
ubifsload ${fdtaddr} /boot/dtb/k3-j721s2-common-proc-board.dtb;
setenv bootargs "console=ttyS2,115200n8 earlycon=ns16550a,mmio32,0x02880000 ubi.mtd=ospi.rootfs
root=ubi0:rootfs rootfstype=ubifs rw init=/init";
booti ${loadaddr} - ${fdtaddr}
'
```

Execute the complete boot sequence as below.

```
run ospi_boot
```

This simplifies future testing and deployment by reducing the Linux boot procedure to a single U-Boot command.

8 Booting Linux Entirely from OSPI NOR Flash

8.1 Automating the Boot Sequence in U-Boot

The previous chapter demonstrated Linux boot using manually entered U-Boot commands. While this approach is useful during development and debugging, production systems typically require the boot process to start automatically after power-on.

One method is to define a default boot command in the U-Boot configuration. This allows U-Boot to automatically mount the UBIFS volume, load the Linux kernel image and device tree from OSPI NOR Flash, and start Linux without user interaction. The default boot command can be added to the following U-Boot configuration file, configs/j721s2_evm_a72_defconfig.

The following configuration can be used in this example.

```
cd ti-processor-sdk-linux-adas-j721s2-evm-11_01_00_03/board-support/ti-u-boot-2025.01+git
cat >> configs/j721s2_evm_a72_defconfig << 'EOF'
CONFIG_USE_BOOTCOMMAND=y
CONFIG_BOOTCOMMAND="setenv bootargs 'console=ttyS2,115200n8 earlycon=ns16550a,mmio32,0x02880000
ubi.mtd=ospi.rootfs root=ubi0:rootfs rootfstype=ubifs rw init=/init'; ubi part ospi.rootfs;
ubifsmount ubi0:rootfs; ubifsload ${loadaddr} /boot/Image; ubifsload ${fdtaddr} /boot/dtb/k3-j721s2-
common-proc-board.dtb; booti ${loadaddr} - ${fdtaddr}"
EOF
```

After rebuilding U-Boot and flashing the updated u-boot.img into the ospi.u-boot partition, the board automatically executes the complete Linux boot sequence during power-on.

8.2 End-to-End Boot Flow

Figure 8-1 shows the complete Linux boot sequence from OSPI NOR Flash.

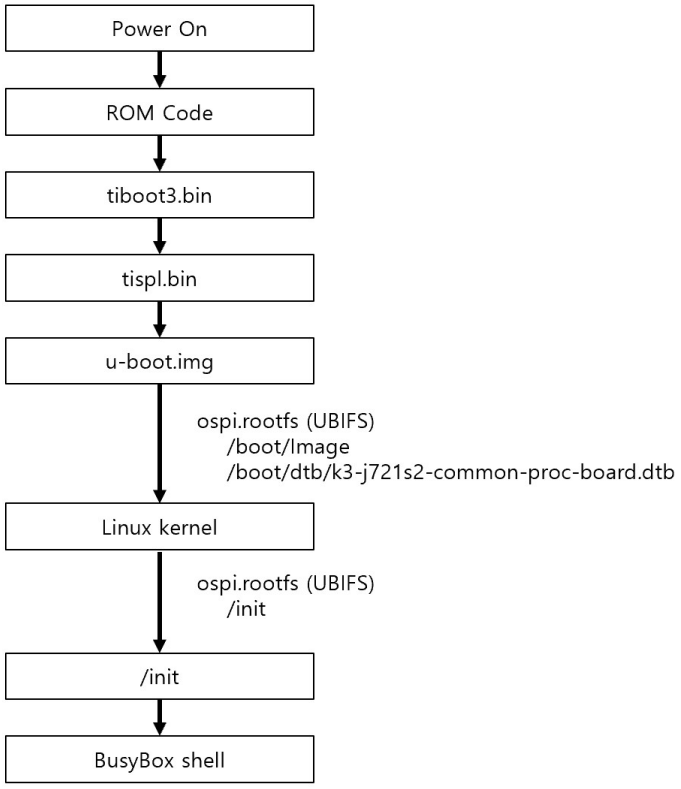

Figure 8-1. End-to-End Boot Flow

After U-Boot mounts the UBIFS volume, it loads the Linux kernel image and device tree from the OSPI rootfs partition. Linux subsequently mounts the same UBIFS volume as its root filesystem and executes the /init script, which launches the BusyBox shell environment.

9 Summary

This application note demonstrated how to boot Linux entirely from OSPI NOR Flash on the TI J721S2 platform using a UBIFS-based root filesystem.

A minimal root filesystem was created using BusyBox, a small set of runtime libraries, and a custom initialization script. The Linux kernel image and device tree were stored within the same UBIFS volume, enabling U-Boot to load all required software components directly from OSPI NOR Flash. The resulting system successfully booted Linux without requiring eMMC, SD card, or other external storage devices. By combining UBI and UBIFS with a minimal BusyBox-based root filesystem, a compact and self-contained Linux boot solution can be implemented using only OSPI NOR Flash. This approach is particularly useful for embedded systems that require simplified storage architecture, reduced component count, or operation without removable storage media.

10 References

- Texas Instruments, *TDA4VE*, product page.
- Texas Instruments, *Texas Instruments, J721S2, TDA4AL, TDA4VL, TDA4VE, AM68A Technical Reference Manual*, technical reference manual.
- Texas Instruments, *Texas Instruments, TDA4VE TDA4AL TDA4VL Jacinto™ Processors, Silicon Revision 1.0 datasheet*, datasheet.
- Texas Instruments, *PROCESSOR-SDK-J721S2*, product page.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Last updated 10/2025