Embedded Base Boot Requirements (EBBR) Specification

Release v1.0.1

Arm Limited and Contributors

Aug 05, 2020

CONTENTS

Table 1: Revision History

Date	Issue	Changes
20 September 2017	0.51	Confidentiality Change, EBBR version 0.51
6 July 2018	0.6-pre1	• Relicense to CC-BY-SA 4.0 • Added Devicetree requirements • Added Multiprocessor boot requirements • Transitioned to reStructuredText and GitHub • Added firmware on shared media requirements • RTC is optional • Add constraints on sharing devices between firmware and OS
12 July 2018	0.6	• Response to comments on v0.6-pre1 • Add large note on implementation of runtime modification of non-volatile variables
18 October 2018	0.7	• Add AArch32 details • Refactor Runtime Services text after face to fact meeting at Linaro Connect YVR18
12 March 2019	0.8	• Update language around SetVariable() and what is available during runtime services • Editorial changes preparing for v1.0
31 March 2019	1.0	• Remove unnecessary UEFI requirements appendix • Allow for ACPI vendor id in firmware path
5 August 2020	1.0.1	• Update to UEFI 2.8 Errata A • Specify UUID for passing DTB • Typo and editorial fixes • Document the release process

CHAPTER

ONE

ABOUT THIS DOCUMENT

1.1 Introduction

This Embedded Base Boot Requirements (EBBR) specification defines an interface between platform firmware and an operating system that is suitable for embedded platforms. EBBR compliant platforms present a consistent interface that will boot an EBBR compliant operating system without any custom tailoring required. For example, an Arm A-class embedded platform will benefit from a standard interface that supports features such as secure boot and firmware update.

This specification defines the base firmware requirements for EBBR compliant platforms. The requirements in this specification are expected to be minimal yet complete, while leaving plenty of room for innovations and design details. This specification is intended to be OS-neutral.

It leverages the prevalent industry standard firmware specification of [UEFI].

Comments or change requests can be sent to *boot-architecture@lists.linaro.org*.

1.2 Guiding Principles

EBBR as a specification defines requirements on platforms and operating systems, but requirements alone don't provide insight into why the specification is written the way it is, or what problems it is intended to solve. Using the assumption that better understanding of the thought process behind EBBR will result in better implementations, this section is a discussion of the goals and guiding principle that shaped EBBR.

This section should be considered commentary, and not a formal part of the specification.

EBBR was written as a response to the lack of boot sequence standardization in the embedded system ecosystem. As embedded systems are becoming more sophisticated and connected, it is becoming increasingly important for embedded systems to run standard OS distributions and software stacks, or to have consistent behaviour across a large deployment of heterogeneous platforms. However, the lack of consistency between platforms often requires per-platform customization to get an OS image to boot on multiple platforms.

A large part of this ecosystem is based on U-Boot and Linux. Vendors have heavy investments in both projects and are not interested in large scale changes to their firmware architecture. The challenge for EBBR is to define a set of boot standards that reduce the amount of custom engineering required, make it possible for OS distributions to support embedded platforms, while still preserving the firmware stack product vendors are comfortable with. Or in simpler terms, EBBR is designed to solve the embedded boot mess by adding a defined standard (UEFI) to the existing firmware projects (U-Boot).

However, EBBR is a specification, not an implementation. The goal of EBBR is not to mandate U-Boot and Linux. Rather, it is to mandate interfaces that can be implemented by any firmware or OS project, while at the same time work with both Tianocore/EDK2 and U-Boot to ensure that the EBBR requirements are implemented by both projects.[1]

The following guiding principles are used while developing the EBBR specification.

[1] Tianocore/EDK2 and U-Boot are highlighted here because at the time of writing these are the two most important firmware projects that implement UEFI. Tianocore/EDK2 is a full featured UEFI implementation and so should automatically be EBBR compliant. U-Boot is the incumbant firmware project for embedded platforms and has steadily been adding UEFI compliance since 2016.

- Be agnostic about ACPI and Devicetree.

  EBBR explicitly does not require a specific system description language. Both Devicetree and ACPI are supported. The Linux kernel supports both equally well, and so EBBR doesn't require one over the other. However, EBBR does require the system description to be supplied by the platform, not the OS. The platform must also conform to the relevant ACPI or DT specifications and adhere to platform compatibility rules.[2]

- Focus on the UEFI interface, not a specific codebase

  EBBR does not require a specific firmware implementation. Any firmware project can implement these interfaces. Neither U-Boot nor Tianocore/EDK2 are required.

- Design to be implementable and useful today

  The drafting process for EBBR worked closely with U-Boot and Tianocore developers to ensure that current upstream code will meet the requirements.

- Design to be OS independent

  This document uses Linux as an example but other OS's are expected.

- Support multiple architectures

  Any architecture can implement the EBBR requirements. Architecture specific requirements will clearly marked as to which architecture(s) they apply.

- Design for common embedded hardware

  EBBR support will be implemented on existing developer hardware. Generally anything that has a near-upstream U-Boot implementation should be able to implement the EBBR requirements. EBBR was drafted with readily available hardware in mind, like the Raspberry Pi and BeagleBone families of boards, and it is applicable for low cost boards (<$10).

- Plan to evolve over time

  The v1.0 release of EBBR is firmly targeted at existing platforms so that gaining EBBR compliance may require a firmware update, but will not require hardware changes for the majority of platforms.

  Future EBBR releases will tighten requirements to add features and improve compatibility, which may affect hardware design choices. However, EBBR will not retroactively revoke support from previously compliant platforms. Instead, new requirements will be clearly documented as being over and above what was required by a previous release. Existing platforms will be able to retain compliance with a previous requirement level. In turn, OS projects and end users can choose what level of EBBR compliance is required for their use case.

1.3 Scope

This document defines the boot and runtime services that are expected by an Operating System or hypervisor, for a device which follows the UEFI specification [UEFI].

This specification defines the boot and runtime services for a physical system, including services that are required for virtualization. It does not define a standardized abstract virtual machine view for a Guest Operating System.

This specification is similar to the Arm Server Base Boot Requirements specification [SBBR] in that it defines the firmware interface presented to an operating system. SBBR is targeted at the server ecosystem and places strict requirements on the platform to ensure cross vendor interoperability. EBBR on the other hand allows more flexibility to support embedded designs which do not fit within the SBBR model. For example, a platform that isn't SBBR compliant because the SoC is only supported using Devicetree could be EBBR compliant, but not SBBR compliant.

By definition, all SBBR compliant systems are also EBBR compliant, but the converse is not true.

[2] It must be acknowledged that at the time of writing this document, platform compatibility rules for DT platforms are not well defined or documented. We the authors recognize that this is a problem and are working to solve it in parallel with this specification.

1.4 Cross References

This document cross-references sources that are listed in the References section by using the section sign §.

Examples:

UEFI § 6.1 - Reference to the UEFI specification [UEFI] section 6.1

1.5 Terms and abbreviations

This document uses the following terms and abbreviations.

A64 The 64-bit Arm instruction set used in AArch64 state. All A64 instructions are 32 bits.

AArch32 Arm 32-bit architectures. AArch32 is a roll up term referring to all 32-bit versions of the Arm architecture starting at ARMv4.

AArch64 state The Arm 64-bit Execution state that uses 64-bit general purpose registers, and a 64-bit program counter (PC), Stack Pointer (SP), and exception link registers (ELR).

AArch64 Execution state provides a single instruction set, A64.

EFI Loaded Image An executable image to be run under the UEFI environment, and which uses boot time services.

EL0 The lowest Exception level on AArch64. The Exception level that is used to execute user applications, in Non-secure state.

EL1 Privileged Exception level on AArch64. The Exception level that is used to execute Operating Systems, in Non-secure state.

EL2 Hypervisor Exception level on AArch64. The Exception level that is used to execute hypervisor code. EL2 is always in Non-secure state.

EL3 Secure Monitor Exception level on AArch64. The Exception level that is used to execute Secure Monitor code, which handles the transitions between Non-secure and Secure states. EL3 is always in Secure state.

Logical Unit (LU) A logical unit (LU) is an externally addressable, independent entity within a device. In the context of storage, a single device may use logical units to provide multiple independent storage areas.

OEM Original Equipment Manufacturer. In this document, the final device manufacturer.

SiP Silicon Partner. In this document, the silicon manufacturer.

UEFI Unified Extensible Firmware Interface.

UEFI Boot Services Functionality that is provided to UEFI Loaded Images during the UEFI boot process.

UEFI Runtime Services Functionality that is provided to an Operating System after the ExitBootServices() call.

CHAPTER

TWO

UEFI

This chapter discusses specific UEFI implementation details for EBBR compliant platforms.

2.1 UEFI Version

This document uses version 2.8 Errata A of the UEFI specification [UEFI].

2.2 UEFI Compliance

EBBR compliant platforms shall conform to the requirements in [UEFI] § 2.6, except where explicit exemptions are provided by this document.

2.2.1 Block device partitioning

The system firmware must implement support for MBR, GPT and El Torito partitioning on block devices. System firmware may also implement other partitioning methods as needed by the platform, but OS support for other methods is outside the scope of this specification.

2.3 UEFI System Environment and Configuration

The resident UEFI boot-time environment shall use the highest non-secure privilege level available. The exact meaning of this is architecture dependent, as detailed below.

Resident UEFI firmware might target a specific privilege level. In contrast, UEFI Loaded Images, such as third-party drivers and boot applications, must not contain any built-in assumptions that they are to be loaded at a given privilege level during boot time since they can, for example, legitimately be loaded into either EL1 or EL2 on AArch64.

2.3.1 AArch64 Exception Levels

On AArch64 UEFI shall execute as 64-bit code at either EL1 or EL2, depending on whether or not virtualization is available at OS load time.

UEFI Boot at EL2

Most systems are expected to boot UEFI at EL2, to allow for the installation of a hypervisor or a virtualization aware Operating System.

UEFI Boot at EL1

Booting of UEFI at EL1 is most likely within a hypervisor hosted Guest Operating System environment, to allow the subsequent booting of a UEFI-compliant Operating System. In this instance, the UEFI boot-time environment can be provided, as a virtualized service, by the hypervisor and not as part of the host firmware.

2.4 UEFI Boot Services

2.4.1 Memory Map

The UEFI environment must provide a system memory map, which must include all appropriate devices and memories that are required for booting and system configuration.

All RAM defined by the UEFI memory map must be identity-mapped, which means that virtual addresses must equal physical addresses.

The default RAM allocated attribute must be EFI_MEMORY_WB.

2.4.2 Configuration Tables

A UEFI system that complies with this specification may provide the additional tables via the EFI Configuration Table.

Compliant systems are required to provide one, but not both, of the following tables:

- an Advanced Configuration and Power Interface [ACPI] table, or
- a Devicetree [DTSPEC] system description

EBBR systems must not provide both ACPI and Devicetree tables at the same time. Systems that support both interfaces must provide a configuration mechanism to select either ACPI or Devicetree, and must ensure only the selected interface is provided to the OS loader.

Devicetree

If firmware provides a Devicetree system description then it must be provided in Flattened Devicetree Blob (DTB) format version 17 or higher as described in [DTSPEC] § 5.1. The following GUID must be used in the EFI system table ([UEFI] § 4) to identify the DTB. The DTB must be contained in memory of type EfiACPIReclaimMemory. EfiACPIReclaimMemory was chosen to match the recommendation for ACPI tables which fulfill the same task as the DTB.

```
#define EFI_DTB_GUID \
     EFI_GUID(0xb1b621d5, 0xf19c, 0x41a5, \
              0x83, 0x0b, 0xd9, 0x15, 0x2c, 0x69, 0xaa, 0xe0)
```

Firmware must have the DTB resident in memory and installed in the EFI system table before executing any UEFI applications or drivers that are not part of the system firmware image. Once the DTB is installed as a configuration table, the system firmware must not make any modification to it or reference any data contained within the DTB.

UEFI applications are permitted to modify or replace the loaded DTB. System firmware must not depend on any data contained within the DTB. If system firmware makes use of a DTB for its own configuration, it should use a separate private copy that is not installed in the EFI System Table or otherwise be exposed to EFI applications.

2.4.3 UEFI Secure Boot (Optional)

UEFI Secure Boot is optional for this specification.

If Secure Boot is implemented, it must conform to the UEFI specification for Secure Boot. There are no additional requirements for Secure Boot.

2.5 UEFI Runtime Services

UEFI runtime services exist after the call to ExitBootServices() and are designed to provide a limited set of persistent services to the platform Operating System or hypervisor. Functions contained in EFI_RUNTIME_SERVICES are expected to be available during both boot services and runtime services. However, it isn't always practical for all EFI_RUNTIME_SERVICES functions to be callable during runtime services due to hardware limitations. If any EFI_RUNTIME_SERVICES functions are only available during boot services then firmware shall provide the *EFI_RT_PROPERTIES_TABLE* to indicate which functions are available during runtime services. Functions that are not available during runtime services shall return EFI_UNSUPPORTED.

Table 2.1 details which EFI_RUNTIME_SERVICES are required to be implemented during boot services and runtime services.

Table 2.1: EFI_RUNTIME_SERVICES Implementation Requirements

EFI_RUNTIME_SERVICES function	Boot Services	Runtime Services
EFI_GET_TIME	Optional	Optional
EFI_SET_TIME	Optional	Optional
EFI_GET_WAKEUP_TIME	Optional	Optional
EFI_SET_WAKEUP_TIME	Optional	Optional
EFI_SET_VIRTUAL_ADDRESS_MAP	N/A	Required
EFI_CONVERT_POINTER	N/A	Required
EFI_GET_VARIABLE	Required	Optional
EFI_GET_NEXT_VARIABLE_NAME	Required	Optional
EFI_SET_VARIABLE	Required	Optional
EFI_GET_NEXT_HIGH_MONO_COUNT	N/A	Optional
EFI_RESET_SYSTEM	Required	Optional
EFI_UPDATE_CAPSULE	Optional	Optional
EFI_QUERY_CAPSULE_CAPABILITIES	Optional	Optional
EFI_QUERY_VARIABLE_INFO	Optional	Optional

2.5.1 Runtime Device Mappings

Firmware shall not create runtime mappings, or perform any runtime IO that will conflict with device access by the OS. Normally this means a device may be controlled by firmware, or controlled by the OS, but not both. E.g. if firmware attempts to access an eMMC device at runtime then it will conflict with transactions being performed by the OS.

Devices that are provided to the OS (i.e., via PCIe discovery or ACPI/DT description) shall not be accessed by firmware at runtime. Similarly, devices retained by firmware (i.e., not discoverable by the OS) shall not be accessed by the OS.

Only devices that explicitly support concurrent access by both firmware and an OS may be mapped at runtime by both firmware and the OS.

Real-time Clock (RTC)

Not all embedded systems include an RTC, and even if one is present, it may not be possible to access the RTC from runtime services. e.g., The RTC may be on a shared I2C bus which runtime services cannot access because it will conflict with the OS.

If firmware does not support access to the RTC, then GetTime() and SetTime() shall return EFI_UNSUPPORTED, and the OS must use a device driver to control the RTC.

2.5.2 UEFI Reset and Shutdown

ResetSystem() is required to be implemented in boot services, but it is optional for runtime services. During runtime services, the operating system should first attempt to use ResetSystem() to reset the system. If firmware doesn't support ResetSystem() during runtime services, then the call will immediately return EFI_UNSUPPORTED, and the OS should fall back to an architecture or platform specific reset mechanism.

On AArch64 platforms implementing [PSCI], if ResetSystem() is not implemented then the Operating System should fall back to making a PSCI call to reset or shutdown the system.

2.5.3 Runtime Variable Access

There are many platforms where it is difficult to implement SetVariable() for non-volatile variables during runtime services because the firmware cannot access storage after ExitBootServices() is called.

e.g., If firmware accesses an eMMC device directly at runtime, it will collide with transactions initiated by the OS. Neither U-Boot nor Tianocore have a generic solution for accessing or updating variables stored on shared media.[1]

If a platform does not implement modifying non-volatile variables with SetVariable() after ExitBootServices(), then firmware shall return EFI_UNSUPPORTED for any call to SetVariable(), and must advertise that SetVariable() isn't available during runtime services via the *RuntimeServicesSupported* value in the *EFI_RT_PROPERTIES_TABLE* as defined in [UEFI] § 4.6. EFI applications can read *RuntimeServicesSupported* to determine if calls to SetVariable() need to be performed before calling ExitBootServices().

Even when SetVariable() is not supported during runtime services, firmware should cache variable names and values in EfiRuntimeServicesData memory so that GetVariable() and GetNextVeriableName() can behave as specified.

[1] It is worth noting that OP-TEE has a similar problem regarding secure storage. OP-TEE's chosen solution is to rely on an OS supplicant agent to perform storage operations on behalf of OP-TEE. The same solution may be applicable to solving the UEFI non-volatile variable problem, but it requires additional OS support to work. Regardless, EBBR compliance does not require SetVariable() support during runtime services.
https://github.com/OP-TEE/optee_os/blob/master/documentation/secure_storage.md

CHAPTER

THREE

PRIVILEGED OR SECURE FIRMWARE

3.1 AArch32 Multiprocessor Startup Protocol

There is no standard multiprocessor startup or CPU power management mechanism for ARMv7 and earlier platforms. The OS is expected to use platform specific drivers for CPU power management. Firmware must advertize the CPU power management mechanism in the Devicetree system description or the ACPI tables so that the OS can enable the correct driver. At ExitBootServices() time, all secondary CPUs must be parked or powered off.

3.2 AArch64 Multiprocessor Startup Protocol

On AArch64 platforms, Firmware resident in Trustzone EL3 must implement and conform to the Power State Coordination Interface specification [PSCI].

Platforms without EL3 must implement one of:

- PSCI at EL2 (leaving only EL1 available to an operating system)
- Linux AArch64 spin tables [LINUXA64BOOT] (Devicetree only)

However, the spin table protocol is strongly discouraged. Future versions of this specification will only allow PSCI, and PSCI should be implemented in all new designs.

CHAPTER

FOUR

FIRMWARE STORAGE

In general, EBBR compliant platforms should use dedicated storage for boot firmware images and data, independent of the storage used for OS partitions and the EFI System Partition (ESP). This could be a physically separate device (e.g. SPI flash), or a dedicated logical unit (LU) within a device (e.g. eMMC boot partition,[1] or UFS boot LU[2]).

However, many embedded systems have size, cost, or implementation constraints that make separate firmware storage unfeasible. On such systems, firmware and the OS reside in the same storage device. Care must be taken to ensure firmware kept in normal storage does not conflict with normal usage of the media by an OS.

- Firmware must be stored on the media in a way that does not conflict with normal partitioning and usage by the operating system.
- Normal operation of the OS must not interfere with firmware files.
- Firmware needs a method to modify variable storage at runtime while the OS controls access to the device.[3]

4.1 Partitioning of Shared Storage

A shared storage device shall use GPT partitioning unless it is incompatible with the platform boot sequence. In which case, MBR partitioning shall be used.[4]

> **Warning:** MBR partitioning is deprecated and only included for legacy support. All new platforms are expected to use GPT partitioning. GPT partitioning supports a much larger number of partitions, and has built in resiliency.
>
> A future issue of this specification will remove the MBR allowance.

Firmware images and data in shared storage should be contained in partitions described by the GPT or MBR. The platform should locate firmware by searching the partition table for the partition(s) containing firmware.

However, some SoCs load firmware from a fixed offset into the storage media. In this case, to protect against partitioning tools overwriting firmware, the firmware image shall either reside entirely within the first 1MiB of storage, or should be covered by a protective partition entry in the partition table as described in sections *GPT partitioning* and *MBR partitioning*.

[1] Watch out for the ambiguity of the word 'partition'. In most of this document, a 'partition' is a contiguous region of a block device as described by a GPT or MBR partition table, but eMMC devices also provide a dedicated 'boot partition' that is addressed separately from the main storage region, and does not appear in the partition table.

[2] For the purposes of this document, logical units are treated as independent storage devices, each with their own GPT or MBR partition table. A platform that uses one LU for firmware, and another LU for OS partitions and the ESP is considered to be using dedicated firmware storage.

[3] Runtime access to firmware data may still be an issue when firmware is stored in a dedicated LU, simply because the OS remains in control of the storage device command stream. If firmware doesn't have a dedicated channel to the storage device, then the OS must proxy all runtime storage IO.

[4] For example, if the boot ROM doesn't understand GPT partitioning, and will only work with an MBR, then the storage must be partitioned using an MBR.

Automatic partitioning tools (e.g. an OS installer) must not create partitions within the first 1MiB of storage, or delete, move, or modify protective partition entries. Manual partitioning tools should provide warnings when modifying protective partitions or creating partitions within the first 1MiB.

> **Warning:** Fixed offsets to firmware data is supported only for legacy reasons. All new platforms are expected to use partitions to locate firmware files.
>
> A future issues of this specification will disallow the use of fixed offsets.

4.1.1 GPT partitioning

The partition table must strictly conform to the UEFI specification and include a protective MBR authored exactly as described in [UEFI] § 5 (hybrid partitioning schemes are not permitted).

Protective partitions must have the Platform Required Attribute Flag set.

MBR partitioning

Protective partitions should have a partition type of 0xF8 unless some immutable feature of the platform makes this impossible.

4.2 Firmware Partition Filesystem

Where possible, firmware images and data should be stored in a filesystem. Firmware can be stored either in a dedicated firmware partition, or in certain circumstances in the UEFI System Partition (ESP). Using a filesystem makes it simpler to manage multiple firmware files and makes it possible for a single disk image to contain firmware for multiple platforms.

When firmware is stored in the ESP, the ESP should contain a directory named `/FIRMWARE` in the root directory, and all firmware images and data should be stored in platform vendor subdirectories under `/FIRMWARE`.

Dedicated firmware partitions should be formatted with a FAT filesystem as defined by the UEFI specification. Dedicated firmware partitions should use the same `/FIRMWARE` directory hierarchy. OS tools shall ignore dedicated firmware partitions, and shall not attempt to use a dedicated firmware partition as an ESP.

Vendors may choose their own subdirectory name under `/FIRMWARE`, but shall choose names that do not conflict with other vendors. Normally the vendor name will be the name of the SoC vendor, because the firmware directory name will be hard coded in the SoC's boot ROM. Vendors are recommended to use their Devicetree vendor prefix or ACPI vendor ID as their vendor subdirectory name.

Vendors are free to decide how to structure subdirectories under their own vendor directory, but they shall use a naming convention that allows multiple SoCs to be supported in the same filesystem.

For example, a vendor named Acme with two SoCs, AM100 & AM300, could choose to use the SoC part number as a subdirectory in the firmware path:

```
/FIRMWARE
  /ACME
    /AM100
      fw.img
    /AM300
      fw.img
```

It is also recommended for dedicated firmware partitions to use the `/FIRMWARE` file hierarchy.

The following is a sample directory structure for firmware files:

```
/FIRMWARE
  /<Vendor 1 Directory>
     /<SoC A Directory>
        <Firmware image>
        <Firmware data>
     /<SoC B Directory>
        <Firmware image>
        <Firmware data>
  /<Vendor 2 Directory>
     <Common Firmware image>
     <Common Firmware data>
  /<Vendor 3 Directory>
     /<SoC E Directory>
        <Firmware image>
```

Operating systems and installers should not manipulate any files in the `/FIRMWARE` hierarchy during normal operation.

The sections below discuss the requirements when using both fixed and removable storage. However, it should be noted that the recommended behaviour of firmware should be identical regardless of storage type. In both cases, the recommended boot sequence is to first search for firmware in a dedicated firmware partition, and second search for firmware in the ESP. The only difference between fixed and removable storage is the recommended factory settings for the platform.

4.2.1 Fixed Shared Storage

Fixed storage is storage that is permanently attached to the platform, and cannot be moved between systems. eMMC and Universal Flash Storage (UFS) device are often used as shared fixed storage for both firmware and the OS.

Where possible, it is prefered for the system to boot from a dedicated boot region on media that provides one (e.g., eMMC) that is sufficiently large. Otherwise, the platform storage should be pre-formatted in the factory with a partition table, a dedicated firmware partition, and firmware binaries installed.

Operating systems must not use the dedicated firmware partition for installing EFI applications including, but not limited to, the OS loader and OS specific files. Instead, a normal ESP should be created. OS partitioning tools must take care not to modify or delete dedicated firmware partitions.

4.2.2 Removable Shared Storage

Removable storage is any media that can be physically removed from the system and moved to another machine as part of normal operation (e.g., SD cards, USB thumb drives, and CDs).

There are two primary scenarios for storing firmware on removable media.

1. Platforms that only have removable media (e.g., The Raspberry Pi has an SD card slot, but no fixed storage).
2. Recovery when on-board firmware has been corrupted. If firmware on fixed media has been corrupted, some platforms support loading firmware from removable media which can then be used to recover the platform.

In both cases, it is desirable to start with a stock OS boot image, copy it to the media (SD or USB), and then add the necessary firmware files to make the platform bootable. Typically, OS boot images won't include a dedicated firmware partition, and it is inconvenient to repartition the media to add one. It is simpler and easier for the user if they are able to copy the required firmware files into the `/FIRMWARE` directory tree on the ESP using the basic file manager tools provided by all desktop operating systems.

On removable media, firmware should be stored in the ESP under the `/FIRMWARE` directory structure as described in *Firmware Partition Filesystem*. Platform vendors should support their platform by providing a single .zip file that places all the required firmware files in the correct locations when extracted in the ESP `/FIRMWARE` directory. For simplicity sake, it is expected the same .zip file will recover the firmware files in a dedicated firmware partition.

BIBLIOGRAPHY

[ACPI] Advanced Configuration and Power Interface specification v6.2A, September 2017, UEFI Forum

[DTSPEC] Devicetree specification v0.2, Devicetree.org

[LINUXA64BOOT] Linux Documentation/arm64/booting.rst, Linux kernel

[PSCI] Power State Coordination Interface Issue C (PSCI v1.0) 30 January 2015, Arm Limited

[SBBR] Arm Server Base Boot Requirements specification Issue B (v1.0) 8 March 2016, Arm Limited

[UEFI] Unified Extensable Firmware Interface Specification v2.8 Errata A, February 2020, UEFI Forum

INDEX