# UEFI on Top of U-Boot

Standardizing Boot Flow For ARM Boards

Andreas Färber, Alexander Graf
SUSE
Nürnberg, Germany
{andreas.faerber,alexander.graf}@suse.com

***Abstract*—Booting is hard. Booting in the ARM world is even harder. State of the art are a dozen different boot loaders that may or may not deserve that name. Each gets configured differently and each has its own pros and cons. For a Linux distribution this is a nightmare. Configuring each and every one of them complicates code that really should be very simple. To solve the problem, one can just add another layer of abstraction (GRUB2) on top of another layer of abstraction (UEFI) on top of another layer of abstraction (U-Boot). Learn how all those layers can make life easier for the distribution and how much fun UEFI really is. Read how ARM systems boot, what UEFI really means, how UEFI binaries interact with firmware, how openSUSE is going to move to UEFI based boot on ARM and how UEFI enables convergence of the Enterprise and Embedded markets.**

***Keywords—UEFI; U-Boot; GRUB; Linux; openSUSE; ARM; ARMv8; AArch64***

## I. Introduction

Almost as diverse as ARM based chips are the boot mechanisms encountered on them by software developers. A common way to handle such variance is abstraction layers.

Section II compares boot mechanisms and explains some software abstractions. Section III gives a quick overview of UEFI, section IV presents a novel UEFI implementation and section V concludes with a status summary.

## II. Boot Flow on Intel vs. ARM

In Intel based PC, server and embedded markets, the Basic Input/Output System (BIOS) [3] and in recent years the Unified Extensible Firmware Interface (UEFI) [1] offer a standardized boot flow for developers of operating systems and appliances: Independent of processor model and vendor and board model and vendor, BIOS implementations may load and execute boot code from the first 424 bytes on the boot medium, using a Master Boot Record (MBR). Similarly UEFI implementations may load and execute a certain file from a certain GUID Partition Table (GPT) partition on the boot medium. Because BIOS and UEFI are two alternative boot environments, the GRUB bootloader [4] offers another level of abstraction to operating systems in how alternative kernels and their configuration are stored.

SUSE as one Linux vendor have gone one step further with their SUSE Linux Enterprise Server 12 family of products [5] and standardize on GRUB not just for the 64-bit Intel/AMD instruction set architecture but also for Power Architecture and IBM zSystems mainframes ([6] section 8.4.8.9 “GRUB2 Is the Supported Bootloader”).

Such abstractions, at the cost of a little boot time and storage space, add value through software specialization and through choice of software offerings, ultimately benefiting customers and users.

By contrast, ARM based System-on-Chip (SoC) platforms pose much more variance despite their compatible instruction set architecture. While for the 64-bit ARM server market the Server Base System Architecture (SBSA) [7] and Server Base Boot Requirements (SBBR) [8] specifications have led to adoption of UEFI, it has received little interest in the Embedded and Mobile markets so far. Therefore on many ARM based platforms, system developers still need to deploy a highly hardware-specific firmware image rather than a generic operating system image. The SoC may be designed to load code from an offset in flash (e.g., NAND, SPI, eMMC, SD), or from a file in a certain partition, or via serial or USB protocols. For example, the Raspberry Pi single-board computer (with a Broadcom BCM2835 SoC) loads a file bootcode.bin[1] from the first FAT16 partition in an MBR; the Qualcomm Snapdragon family of SoCs load code from a certain GPT partition; the NXP i.MX 6 family of SoCs load code from a 1024 byte offset; the Marvell Armada 380 family of SoCs load code from offset 512; the 32-bit Allwinner SoCs load code from offset 8192; etc. Since many of these locations are mutually exclusive to each other as well as to UEFI, it is impossible to create a single unified boot medium that boots on all of them. Additionally, parts of the boot code need to be specific to the SoC and board or board family, while SRAM and other size constraints rule out a monolithic boot code implementation.

1 bootcode.bin is actually not ARM but VideoCore IV code, but that does not matter from a software deployment perspective.

This has resulted in two trends within the Linux ecosystem:

1) Hardware vendors created their own forked Linux distributions, such as Raspbian for the Raspberry Pi, Bananian for the Banana Pi, OrangeOS for the Orange Pi, Parabuntu for the Parallella, Udoobuntu for Udoo, TurrisOS for Turris Omnia, etc. Thereby for a new SoC (and board) users would initially be bound to the vendor's software offering, both for availability of software and for boot know-how.

2) Linux distributions needed to create board-specific installation images, such as various openSUSE JeOS images or the SUSE Linux Enterprise Server 12 SP2 for Raspberry Pi product [12]. This poses Quality Assurance challenges in particular for a rolling-release distribution like Tumbleweed.

For the SUSE Linux Enterprise Server 12 SP2 for ARM product, shipping any SoC-specific media, bootloaders or Device Trees was not a scalable solution. Yet at the same time restricting the product to SBBR-compliant hardware was undesirable for the SUSE Embedded business.

This left two solutions for SUSE partners:

a) Implement a UEFI compliant firmware for SoCs and reference boards. The UEFI compliant firmware then loads SUSE's GRUB from installation or storage medium – GRUB then loads the Linux kernel from filesystem or network. Within Linux, existing tools like *yast-bootloader* and *grub2-mkconfig* can update the GRUB configuration for the next boot.

b) Continue using custom bootloaders, losing Enterprise-class functionality like BTRFS snapshot boot and a convenient installation method. The bootloader must then be able to load the kernel from BTRFS, XFS and Ext4 volumes itself, and the kernel command line would need to be maintained differently for each bootloader. Also, Device Trees would need to get delivered by the operating system rather than by firmware.

In practice, option a) so far meant the vendor either has to buy AMI Aptio [14] or to fork and maintain Tianocore EDK2 [15] as alternative bootloader.

For option b) only very recent versions of U-Boot [2] have sufficient Ext4 filesystem support; and as of v2017.01 it has neither XFS nor BTRFS support yet,[2] forcing the use of a separate Ext4 formatted */boot* partition in that case.

The broad and scalable ARM SoC support in U-Boot led to the idea of implementing a UEFI compliant boot command in U-Boot, to combine the best of both worlds.

## III. Unified Extensible Firmware Interface

Conceptually UEFI [1] provides its applications with two types of services, **Boot Services** and **Runtime Services** ([1] sections 6 and 7). GRUB [4] will consume Boot Services and Runtime Services, whereas Linux [10] consumes Boot Services only during early start and may consume Runtime Services only while it is actively running.[3] Additionally it provides **Configuration Tables** ([1] section 4.6).

2 Turris Omnia carries a patch for BTRFS support.

3 The Linux command line option *efi=noruntime* suppresses use of Runtime Services.

UEFI is often confused with the Advanced Configuration and Power Interface (ACPI) [9]. Since September 2013 both specifications are published by Unified EFI Inc. and both are mandated in the ARM SBBR [8], but UEFI can be used without ACPI.

An EFI application is passed a pointer to a **System Table** ([1] section 4). The System Table provides among others access to function callbacks for Boot Services and Runtime Services as well as a list of the Configuration Tables.

Configuration Tables are identified by a Globally Unique Identifier (GUID), allowing vendors to add tables not defined in the UEFI specification. One such Configuration Table GUID is defined by Linux for the Flat Device Tree (FDT), i. e. by default Linux will search for a Configuration Table with the FDT GUID and use its contents to initialize device drivers, and only when this is absent or when *acpi=force* is used will Linux search for a Configuration Table with an ACPI GUID.

Boot Services include callbacks for memory management, timers, device discovery (e. g., storage, network, console, framebuffer) and vendor extensions, as well as for “exiting” the Boot Services, after which they may no longer be used by the application.

Runtime Services include callbacks for managing time and variables, as well as for system reset. These callbacks remain available even after Boot Services have been discarded.

## IV. Implementing UEFI in U-Boot

Das U-Boot is a bootloader under GNU General Public License with an active community and wide adoption in the Embedded ARM market. It offers an interactive shell with among others boot commands such as *bootm*, *bootz* and *booti*, and a *distro* framework for non-interactive booting of Linux.

This distro framework specifically allows to execute a *boot.scr* script file from a partition on a boot medium. This has been used for openSUSE images and is supported in the *kiwi* appliance tool, but the file and the command line arguments therein are not updated by *yast2-bootloader*, so that the user has to manually maintain the script file with *mkimage*. Kernel version updates are handled by symlinking to the latest kernel, *initrd* and the *dtb* directory – if an updated kernel fails to boot, either the user needs to manually enter boot commands or tweak the symlinks on the boot medium.

The distro framework further allows to load a config file *extlinux/extlinux.conf* that can list multiple kernels that the user can then select from. This comes close to GRUB functionality, but openSUSE does not support the generation of this config file that differs in location and syntax from *grub.cfg*.

GRUB actually has an *arm-uboot* backend for 32-bit ARM. There are two problems with this, apart from not being implemented for *arm64*: 1) GRUB hardcodes the physical address of RAM, making the binary somewhat SoC-specific. 2) It relies on an optional API feature of U-Boot, which is disabled by default, therefore not build-tested for most targets and appeared to be in poor shape. A UEFI implementation could leverage U-Boot drivers, while allowing to boot a generic GRUB binary.

For some boards the distro framework had to be enabled.

The new configuration options *CONFIG_EFI_LOADER=y* and *CONFIG_CMD_BOOTEFI=y* are enabled by default for ARM and x86 targets. The latter makes available a *bootefi* command that similar to other boot commands accepts the address of a previously loaded binary (e. g., GRUB) and optionally the address to use for the FDT Configuration Table.

The distro framework has been extended to scan partitions for *EFI/BOOT/BOOTARM.EFI* or *EFI/BOOT/BOOTAA64.EFI* files, as well as for the board-specific .dtb file, and to invoke *bootefi* for them. If no matching *.dtb* file is found, it may fall back to reusing the internal Device Tree used by U-Boot itself.

For implementing, e.g., reset Runtime Services, care has to be taken to annotate any code and data used in the callback implementation so that they do not get unmapped when Boot Services are exited.

The UEFI implementation resulted in discovery of and fixes for generic cache handling issues [16]. Due to EFI Boot Services implementing their own memory management API, and GRUB implementing yet another memory management layer based on that, memory reservations had to be added for a few SoCs that previously went undiscovered.

All callbacks for UEFI 2.5 compliance are provided, but not all are implemented. For example, time functions would require real-time clock drivers in U-Boot and would require them to be designed to work as Runtime Services. And EFI variables can't safely access the same eMMC or SD device Linux drivers are using. For that reason U-Boot cannot provide a random seed as EFI variable to Linux, resulting in a warning in recent kernels that can be ignored.

## V. Summary and Outlook

As of U-Boot v2017.01, booting Linux kernels via GRUB is known to be working on the following SoCs (boards):

- Allwinner A64 (Pine64)
- Altera Cyclone V (DE0-Nano-SoC)
- Amlogic S905 (Odroid-C2)
- Broadcom BCM2835/2836/2837 (Raspberry Pi 1/2/3)
- HiSilicon Kirin 620 (HiKey)
- Marvell Armada 388 (ClearFog)
- Nvidia Tegra K1 (Jetson TK1)
- NXP i.MX 6SoloX (Udoo Neo)
- NXP QorIQ LS2085A (LS2085ARDB)
- Rockchip RK3288 (Firefly-RK3288)
- Texas Instruments Sitara AM3358 (Beagle Bone)
- Xilinx Zynq UltraScale+ MPSoC (ZCU102)

Many openSUSE JeOS images that were using U-Boot before have been updated to gain a separate EFI FAT partition */boot/efi* alongside an optional */boot* partition and the */* root partition. U-Boot then detects GRUB on the first partition and invokes *bootefi* for it. An openSUSE specific U-Boot patch lets it search for the Device Tree on the second rather than current partition, since openSUSE dtb packages install to */boot/dtb* rather than to EFI-specific */boot/efi/dtb*, which on FAT would not allow symlinks either.

For example, the openSUSE *JeOS-raspberrypi3.aarch64* image on an SD card will result in the boot ROM loading *bootcode.bin* from the first partition, in turn loading *u-boot.bin* from that partition; U-Boot loads *EFI/BOOT/BOOTAA64.EFI* from there, as well as *dtb/broadcom/bcm2837-rpi-3-b.dtb* if it exists on the following Ext4 partition, using U-Boot's SDHCI, partition and filesystem drivers; GRUB then loads, e.g., *Image-4.9.0-1-default* and *initrd-4.9.0-1-default* via EFI Boot Services, indirectly using the SDHCI driver of U-Boot but using partition and filesystem drivers of GRUB; finally Linux gets the Device Tree indirectly through the FDT Configuration Table, uses EFI Boot Services for initial output, loads its own drivers and exits EFI Boot Services. When the user reboots, Linux calls into EFI Runtime Services, where the tiny remaining part of U-Boot performs the hardware reset.

But that is a rather traditional example that still mixes boot firmware and operating system. GRUB does not need to be located on the same boot medium, so the same U-Boot on SD card could boot from USB disk. It becomes even more PC-like when U-Boot can be stored internally, e. g., on eMMC, freeing removable media slots for pure operating system images – now the generic openSUSE *JeOS-efi.aarch64* image can be used, or openSUSE or SUSE Linux Enterprise Server 12 SP2 for ARM installation disks for custom manual or *AutoYaST* based partitioning, package selection, etc.

In summary, U-Boot v2016.05 and later versions offer a quick path to booting UEFI-enabled Linux distributions on platforms where no other UEFI implementation is available. The implementation adds conveniently little overhead; it does not implement all possible UEFI based features, but it has been demonstrated to be sufficient for booting Linux. No investment into ACPI development is necessary due to FDT usage.

Obviously, forks of older U-Boot versions will not benefit from this new feature and its future development, so vendors are encouraged to forward-port and submit their patches in order to merge them and to benefit from such contributions.

For new ARM SoCs no special considerations are needed; enabling *CONFIG_DISTRO_DEFAULTS=y* and adding the distro framework macros to the environment is encouraged, to facilitate a seemless boot flow.

Nothing fundamentally limits this implementation to the ARM architecture, but at least linker scripts would need adaptations for other architectures.

Similarly, nothing limits this to booting Linux distributions; the same boot flow might be used for booting, e. g., Android. Dual-booting could in that case be realized as boot menu entries at the GRUB layer.

## References

[1] Unified EFI Inc., "Unified Extensible Firmware Interface Specification," Version 2.6, January 2016.

[2] Wolfgang Denk et al., "Das U-Boot," http://www.denx.de/wiki/U-Boot/WebHome

[3] Compaq Computer Corp., Phoenix Technologies Ltd. and Intel Corp., "BIOS Boot Specification," Version 1.01, January 11, 1996.

[4] Yoshinori K. Okuji, Free Software Foundation Inc. et al., "GNU GRUB," https://www.gnu.org/software/grub/

[5] SUSE LLC, "SUSE Linux Enterprise 12 Now Available," October 27 2014, https://www.suse.com/newsroom/post/2014/suse-linux-enterprise-12-now-available/

[6] SUSE LLC, "SUSE Linux Enterprise Server 12 Release Notes," https://www.suse.com/releasenotes/x86_64/SUSE-SLES/12/

[7] ARM Ltd., "Server Base System Architecture Specification," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0029/index.html

[8] ARM Ltd., "ARM Server Base Boot Requirements (SBBR)," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0044b/index.html

[9] Unified EFI Inc., "Advanced Configuration and Power Interface Specification," Version 6.1, January 2016.

[10] Linus Torvalds et al., Linux kernel, https://www.kernel.org/

[11] Alexander Graf, "[PATCH 0/9] EFI payload / application support," U-Boot mailing list, December 22 2015.

[12] SUSE LLC, "SUSE Linux Enterprise Server for Raspberry Pi," https://www.suse.com/products/arm/raspberry-pi

[13] SUSE LLC, "SUSE Linux Enterprise Server for ARM," https://www.suse.com/products/arm/

[14] American Megatrends Inc., "Aptio V: ARM Benefits," https://ami.com/products/bios-uefi-firmware/aptio-v/arm-benefits/

[15] Intel Corp. et al., "EDK II," http://www.tianocore.org/edk2/

[16] Ziyuan Xu, "[PATCH v3 4/4] usb: dwc2: invalidate dcache before starting DMA," http://patchwork.ozlabs.org/patch/645189/