BIOS vs. (U)EFI boot

We all know UEFI is newer, and therefore, better, right?

Thanks / Credits

- Thanks to
  - The OpenBSD Project
    - For producing such awesome documentation
  - Intel
    - For open-sourcing part/most of the UEFI specs and implementation, and a little bit of documentation
  - Apple
    - for freaking nothing at all: not following ANY specification, and not even properly documenting what they are doing, either.

The contestants

- BIOS boot
  - Also including UEFI’s CSM (Compatibility Support Module)
  - Boots in real mode, reads a few sectors, executes them
  - Many, many implementations
  - No standard, just “do what they did”
- UEFI boot
  - A complete pre-boot environment
  - Multiple implementations
  - One(-ish) official(-ish) standard

Abbreviated BIOS history

- “BIOS” term originates in CP/M circa 1975
- Written in assembly by IBM for IBM PC
  - Hard disk support added for IBM PC XT
  - 80286 and 16-bit ISA support added for IBM PC AT
    - First occurrence of “CMOS”: 50 bytes, battery-backed
    - First occurrence of ATA support
  - Other BIOS clones reverse-engineered during this era, principally by Compaq, Phoenix, and AMI
- Supports add-in ROMs that extend BIOS functionality
- More and more features added by many vendors, including network boot & many others

Abbreviated UEFI history

- In the beginning, there was darkness…
  - then EFI was created for Itanium, and OpenBoot for SPARCs and PowerPC
- Assembly programming, 16-bit real mode, and pathological coupling to AT hardware made a bunch of people decide that the BIOS sucked now
- And they said “Lo, observe EFI, for it is good”
  - … and then they messed it all up

The first WTF

- UEFI machines are divided into classes:
  - Class 0: Legacy BIOS. As in, no EFI functionality whatsoever
  - Class 1: UEFI in CSM-only mode. Also no EFI functionality whatsoever.
- And it just gets clearer from there…
- Recap:
  - A Class 0 or 1 UEFI system is a pure BIOS/CSM-only system
  - Or, a Class 0 or 1 UEFI system is not really a UEFI system at all
  - Reminds me of recursive acronyms (e.g. GNU) but backwards
    - “UEFI isn’t UEFI”?

Hang on, what’s the CSM?

- Some UEFI implementations come with a “CSM”
- “Compatibility Support Module”
- That’s **backwards** compatibility, i.e. BIOS emulation!

Comparison - 1

Function	BIOS / UEFI CSM	UEFI
Hardware initialization	Integrated into BIOS; if it's not supported, it doesn't get initialized.	Modular approach with "drivers", which can (in theory) be added later by the end-user.
Bootloader	Reads a few sectors from disk at a fixed address and executes them. Then gets out of the way.	Loads an entire mini-OS that selects a file, loads it, and executes it.
Filesystem support	Theoretically infinite, all it cares about are the raw sectors on disk.	FAT16/FAT32 only. Other filesystems can theoretically be supported by UEFI drivers.
Pre-boot Environment and/or Shell	None.	Entire miniature OS exists to load bootable images. Arbitrary executable images can be loaded, including a UEFI Shell.
Processors / Architectures	Re-written in assembler for each CPU/arch.	Recompiled from C (usually) for each CPU/arch.

Comparison - 2

Function	BIOS / UEFI CSM	UEFI
Disk sizes	Depends on implementation. Currently limited to <2TiB.	As long as the EFI partition is within the first 8 ZiB (yes, really), theoretically up to 256 ZiB.
Partition table	Usually “MBR”-style for fixed media, no partitions for removable meia. Can be customized.	GPT on fixed media, can be customized for removable media.
Processor mode	Real-mode (16-bit) only. Some experimental versions did really weird $#@! In 32-bit mode.	32-bit protected mode, because it’s a real (mini-)OS. Only little-endian CPUs are supported at this time.
Accessible memory	1MiB (real-mode)	4GiB (32-bit protected mode)
PCI/PCIe address space	Inside 1MiB (real-mode), can program high addresses but not access them.	Can program and access addresses inside 16EiB (64-bit long mode).

Comparison - 3

Function	BIOS / UEFI CSM	UEFI
Access services from running OS	Requires thunking to 16-bit real-mode.	Well-defined syscall interface from 32-bit or 64-bit protected modes.
Extendability	More BIOS code written in assembly, stored in ROM on add-in cards.	Device driver compiled to EBC (EFI Byte Code), stored in ROM on add-in cards <u>or</u> stored as files in the EFI Service Partition.
Firmware update	Proprietary utility only.	Fully-supported generic firmware update via “UEFI Capsule”.
Cryptographically-secure booting	Zip, zilch, nada. Except by accident, sometimes. (Looking at you, Fujitsu!)	Fully defined (and usually supported) with or without a hardware TPM chip.

Conclusion (or maybe Concussion)

- I’ve told you a bunch of little white lies, much like your 6th grade science teacher, and for the same reason:
  - Reality is too complicated for a 15-minute presentation. For more details, the rabbit hole into an alternate universe starts here:
    - https://en.wikipedia.org/wiki/BIOS
    - https://en.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface

Image credit: Charlie Cottrell, https://xeyeti.com/, https://charliecottrell.com/