presented by

vmware®

ACPI-Lite: Exploring a Simplified Mechanism for Abstracting Platforms with ACPI

UEFI 2021 Virtual Plugfest
Tuesday, July 6

Presented by Andrei Warkentin (VMware)

Meet the Presenter

Andrei Warkentin
Arm Enablement Architect
Member Company: VMware

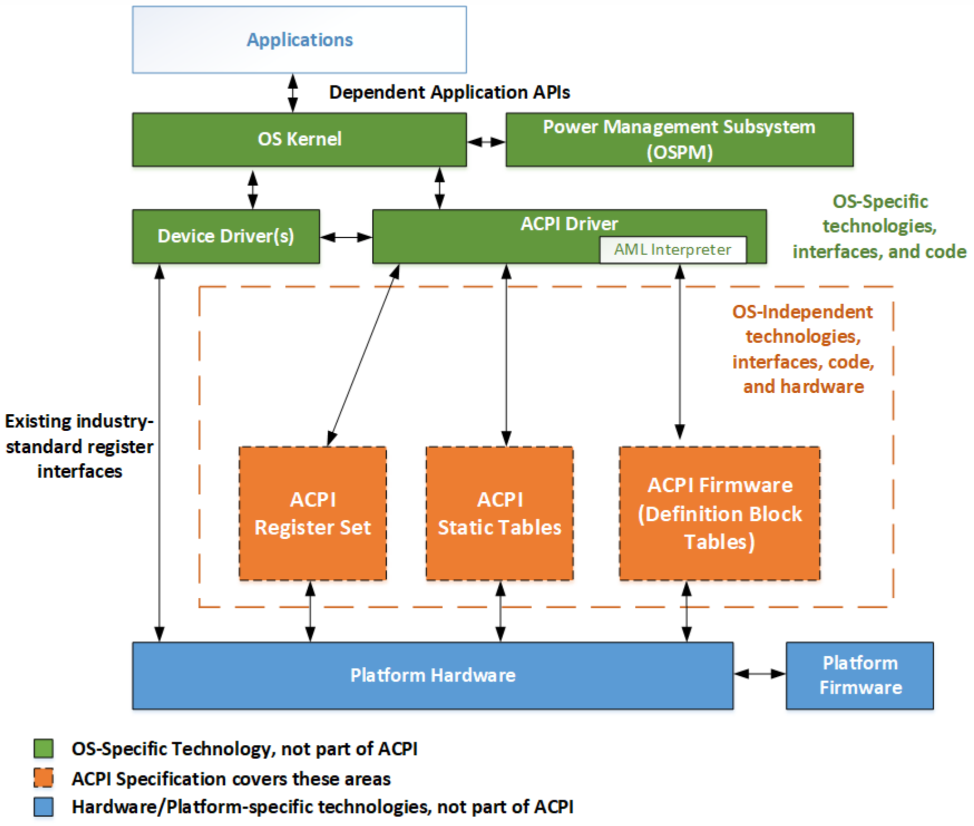
Agenda

- What is ACPI?
- What is Device Tree?
- Challenges with ACPI
- Why bother evolving ACPI?
- Abstracting non-IA platforms
- ACPI and embedded and safety-critical systems
- Becoming better at describing and abstracting

Disclaimer: Not meant as an exhaustive analysis of all areas where ACPI could change/adapt.

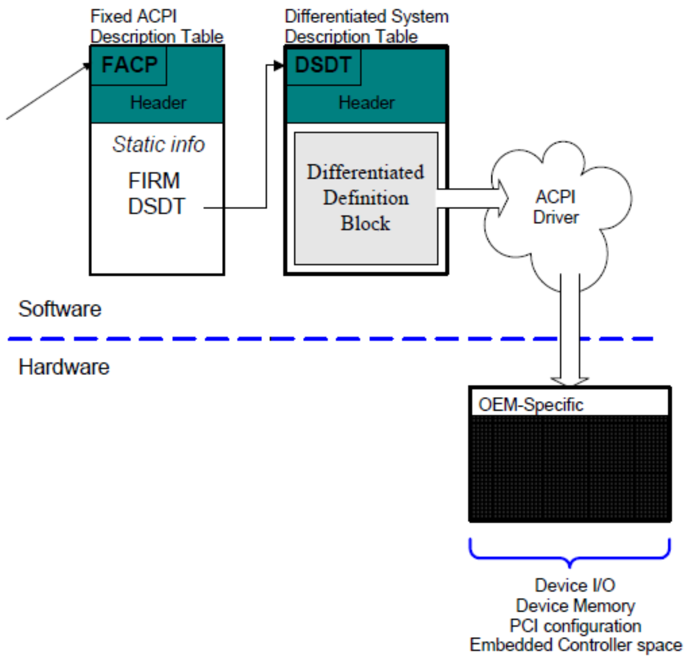
What is ACPI?

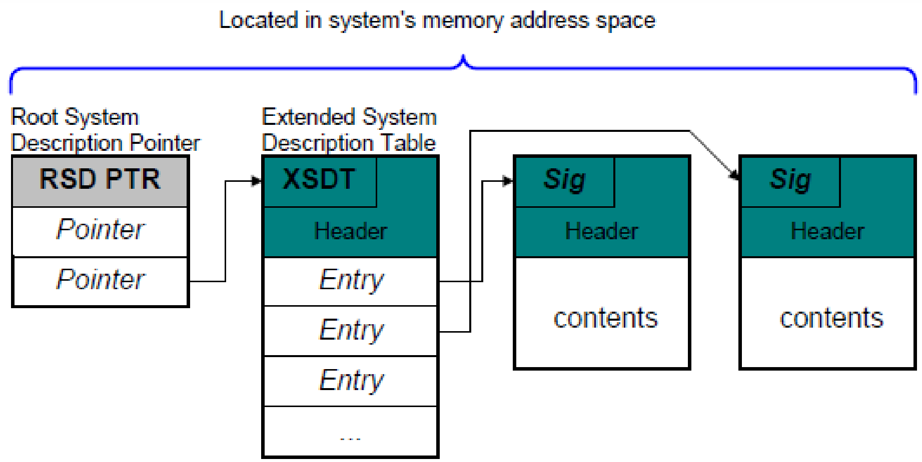
- A set of firmware tables describing hardware
- A set of interfaces between OS and hardware
    - Configuration
    - Power management
- [optionally] ACPI-specific hardware
- Not just a description – an *abstraction* modeling an ”ideal” system
- Agnostic to the OS running on the hardware, ideally



What is ACPI?

- Static tables
- Dynamic bytecode
  - AML interpreter
  - *Generates* an ACPI Namespace tree, a hierarchical description of platform devices
  - Methods to abstract device and platform configuration
  - Interaction with hardware abstracted via Operation Regions (backed by AML interpreter and OS/drivers)





What is Device Tree?

- A hierarchical tree data structure, encoding device characteristics
- Has its roots in a format used to "flatten" OpenFirmware device tree (traversed via CIF calls)
- Unlike ACPI, no dynamic methods, no abstraction, no interpretation
- Unlike OpenFirmware, no CIF, device methods, etc.
- Minimal support logic to use in OS or boot loader environments
- Came from Linux, fairly closely bound to Linux support for various SoCs (e.g. challenging to support with BSDs)
- Platform adaptation/quirks entirely owned by OSV

```
soc {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;

    open-pic {
        clock-frequency = <0>;
        interrupt-controller;
        #address-cells = <0>;
        #interrupt-cells = <2>;
    };

    pci {
        #interrupt-cells = <1>;
        #size-cells = <2>;
        #address-cells = <3>;
        interrupt-map-mask = <0xf800 0 0 7>;
        interrupt-map = <
            /* IDSEL 0x11 - PCI slot 1 */
            0x8800 0 0 1 &open-pic 2 1 /* INTA */
            0x8800 0 0 2 &open-pic 3 1 /* INTB */
            0x8800 0 0 3 &open-pic 4 1 /* INTC */
            0x8800 0 0 4 &open-pic 1 1 /* INTD */
            /* IDSEL 0x12 - PCI slot 2 */
            0x9000 0 0 1 &open-pic 3 1 /* INTA */
            0x9000 0 0 2 &open-pic 4 1 /* INTB */
            0x9000 0 0 3 &open-pic 1 1 /* INTC */
            0x9000 0 0 4 &open-pic 2 1 /* INTD */
        >;
    };
};
```

Challenges with ACPI

- ACPI was defined as an overlay to an existing IA platform. Abstracting non-IA platforms is still a challenge
- Fit for embedded and safety-critical systems
- Choices for servers may not be appropriate for embedded
- Can evolve as a mechanism to meet separate goals
  - Becoming better at *describing* hardware (e.g. configuration for platform-specific OS/driver components)
  - Becoming better at *abstracting* hardware (avoiding platform specific drivers)
  - Goals == capability, not policy. Actual choice of how ACPI is used depends on use-case

Why Bother Evolving ACPI?

- Why would anyone want to use ACPI for real-time, embedded, etc.?
- Why not Device Tree?
  - Linux is not the only OS
  - Device Tree is (today) heavily intertwined with Linux (bindings). Already a problem for BSDs
  - No platform abstraction, even for areas where there’s no benefit from proliferating differences
- Why try to avoid platform-dependent code in the OS to enable ACPI?
  - Not every OS is Linux or Windows
  - Generalize OS-specific extensions/assumptions
  - Avoid cost of development/maintenance by OSVs for basic platform support/quirks

Abstracting Non-IA Platforms

- ACPI 5.0 introduced reduced-hardware mode.
  - No longer requires fixed ACPI hardware (yay!)
  - Relies on OS-backed drivers to provide similar functionality (*sigh)*
  - ACPI encapsulates configuration while requiring OS support for low-level platform internals (e.g. GPE)
  - Addressed in an OS-specific manner via PEPs ("platform extension plug-ins", a Microsoft-only extension for dynamic runtime ACPI method via native code)
- ACPI and DT are getting intertwined
  - ACPI devices that mirror DT ones (PRP0001, _DSD properties)
  - ACPI used more and more to describe, not abstract
  - Trails in abstracting the embedded-style hardware that is well-described today by DT (clocks, power resources, composite devices, complex NIC devices, etc.)
- AML is a bit primitive and very high overhead
  - Asynchronous communication with hardware (e.g. Time and Alarm device without an I2C OpReg)
  - IA memory model (how to communicate with cache coherent hardware? PCIe atomics, barriers, etc.)
  - No quick escapes to other firmware outside of IA SMM to reduce dependence on OS drivers

Embedded and Safety-Critical

AML interpreter is huge

- ACPI-CA is 331k lines.
- Requires significant OS support
- Slow (global locked, interpretation)
- High complexity (security, implementation)

Becoming Better at *Describing*.

How about a valid subset of AML that does not require an interpreter (has strict scoping rules, no control flow) and “canned” encodings of returning static data via methods?

- **Use case:** embedded to support purpose-built software that can’t embed ACPI-CA (yet is still compatible with a “normal” AML interpreter)
- Many simpler SoCs don’t really need dynamic behavior
- …still fully compatible with a regular AML interpreter
- Can have a lighter-weight ACPI OS implementation that only supports fixed/static data with no bytecode interpretation?

Could Take This One Step Further…

Be able to compile DTS (textual) or DTB (binary) into this strict subset of ACPI

- **Use case:** transitioning embedded hardware/software vendors to support general purpose (ACPI) OSes
- Compatible with both regular ACPI interpreter and the lighter-weight one
- Outside of register / interrupt resources, remaining properties map to ACPI using Device Properties _DSD

Becoming Better at *Abstracting*.

Allow AML methods to be selectively implemented in native code (e.g. via Platform Runtime Mechanism, see **PRMT** under https://uefi.org/acpi)

- **Use case:**
  - Provide optimal implementations for performance sensitive parts (or parts that are hard to model with AML), with better sandboxing guarantees that AML
  - Get-out-of-jail free card from AML (escape to firmware, ACPI should be about capable mechanisms, not policy)
- Native != OS-distributed code. Native == machine code owned by SiP/OEM.
- Standardized PEP replacement ("platform extension plug-ins")
  - No reliance on OS-specific extensions
  - Mechanism common across any OS
  - Code owned/maintained by SiP/OEM, not OSV
- No, you don't have to use it or allow it in specific solutions. Not all AML needs to be converted this way, but it's a way to avoid "why would you want to do *that*" kind of conversations around adapting actual hardware to ACPI

Combine Both as Necessary

A strict AML subset (static data) + native methods enable the development of a smaller, lighter weight and more flexible ACPI subsystem, while retaining compatibility with traditional implementations (ACPI-CA)

Call to Action

- Investigate what a “reduced static AML” could looks like.
  - Opt-in, Code-first with ACPI-CA
  - Lighter ACPI-CA alternative only supporting “reduced static AML”
  - DT -> “reduced static AML” converter
- Investigate what “native” AML method could look like.
  - Opt-in, code-first with ACPI-CA
  - Consider different back-ends (PRM? Raw RT? Higher-privilege calls?)

Questions?

Thanks for attending the UEFI 2021 Virtual Plugfest

For more information on UEFI Forum and UEFI Specifications, visit http://www.uefi.org

presented by

vmware®