ACPI Support for Embedded Controllers

BY MARCIN WOJTAS

ARM64 is a single architecture that is used in an extremely wide range of products — it can be found in the smallest embedded devices, but also in mobile devices, enterprise units and even server-grade solutions. The support for the latter imposes certain standards, e.g., the way of booting, interactions with firmware or a platform description. It turnes out this model also fits non-server devices. Let's see how they are supported in FreeBSD, with a focus on handling the embedded controllers in the ACPI world.

Dealing With the Problematic Legacy

When introduced almost a decade ago, the 64-bit variant of ARM directly inherited the ecosystem from its 32-bit predecessor, which had been reigning in the embedded market. However, the usual need of maintaining a fully customized board support package for each platform was a real burden for development of the new architecture. To some extent, the device tree (DT) adoption allowed for better portability and using a single kernel image for various devices, but it did not suffice to solve the problem entirely. This kind of description is very flexible, which was, unfortunately, often abused by vendors and resulted in inconsistent bindings over the time. Even today, it is not uncommon that the device tree blob for U-Boot differs from the one used for booting the OS (they describe the same hardware!), and often also lacks backward compatibility. With such constraints, reaching a long-term goal of a wide software ecosystem and multi-OS support would be problematic.

When introduced almost a decade ago, the 64-bit variant of ARM directly inherited the ecosystem from its 32-bit predecessor.

However, the solution was out there and existed for years. The interfaces used in the x86 world were adopted and extended for ARM64, namely the boot process, EFI, SMBIOS and ACPI. With the server-grade devices that comply with the standards and use proper firmware, it is now possible to install FreeBSD and other OSs or hypervisors out of the box, simply by using installer images. What about smaller, embedded platforms? Fortunately they can also leverage the rich ecosystem the same way. There are conditions though — the hardware must not deviate from the standards (at least not too much) and there are also strict requirements related to firmware. The guidelines are gathered into specifications, consecutively: the BSA (ARM Base System Architecture) and the BBR (ARM Base Boot Requirements). Result — there are ARM64 platforms that can successfully boot the FreeBSD, Windows and multiple Linux distributions, using a single firmware image and ACPI description.

What is special about those devices? Compared to the servers, which traditionally have a significant amount of CPUs, DRAM and PCIE root complexes, in the embedded segment the SoCs also support a wide variety of controllers attached to their internal buses. Therefore, they are not discovered during PCIE enumeration, but require a different treatment. A hardware description must comprise an explicit reference to these interfaces, including the platform data that can be parsed and interpreted by the OS. Recently, the FreeBSD kernel's ability to obtain such information from ACPI tables was extended with some new features.

What is ACPI?

Before jumping to details, it may be worth briefly explaining what the ACPI is — it is an interface between the firmware and OS, used for describing and configuring the hardware. The standard has been developed for almost 3 decades and lists a number of main concepts, i.e., various aspects of power management, thermal/battery handling, hardware configuration and embedded controllers' description. It also defines an ACPI Source Language (ASL), which among others allows for creating low-level hardware configuration routines. It is compiled to a bytecode — ACPI Machine Language (AML), that can be interpreted and executed by the kernel.

The information about a platform is gathered in so-called 'tables,' which are, in fact, a hierarchy of structures in the system's memory address space. The starting point of ACPI is Root System Description Pointer (RSDP) structure — it is configured by firmware and points to Extended System Description Table (XSDT), which further branches out to secondary tables. The first one is always Fixed ACPI Description Table (FADT) — it comprises various fixed-length entries that describe the fixed ACPI features of the hardware.

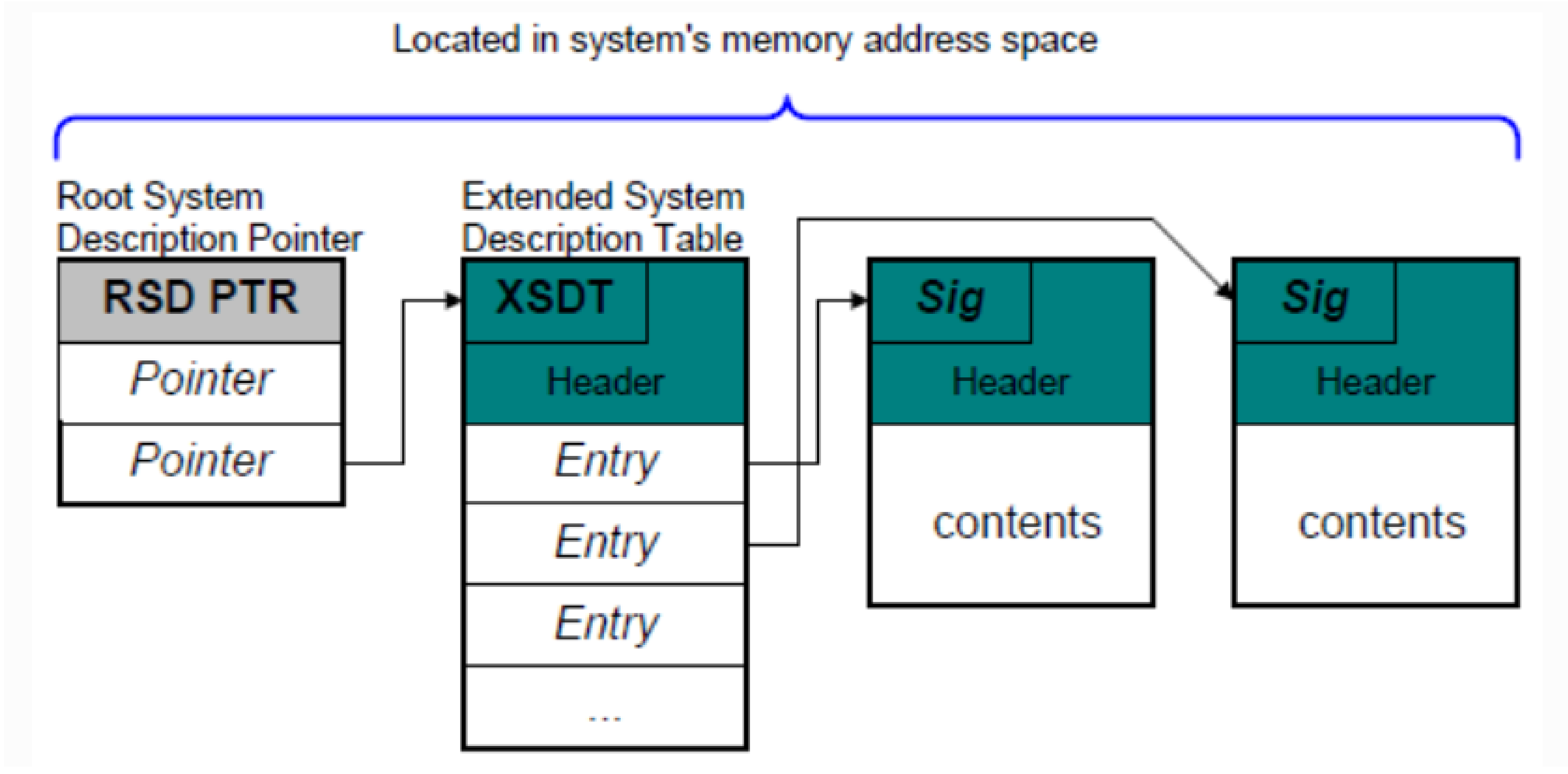
Located in system's memory address space
Root System Description Pointer
RSD PTR
Pointer
Pointer
Extended System Description Table
XSDT
Header
Entry
Entry
Entry
...
Sig
Header
contents
Sig
Header
contents

Fig1. Root System Description Pointer and Table.
Source: https://uefi.org/specs/ACPI/6.4/05_ACPI_Software_Programming_Model/ACPI_Software_Programming_Model.html#overview-of-the-system-description-table-architecture

The ACPI specification defines a number of dedicated tables, however a couple of them can be considered as being more significant in the embedded devices context, e.g.,

- Generic Timer Description Table (GTDT)
- Multiple APIC Description Table (MADT)
- Processor Properties Topology Table (PPTT)
- Serial Port Console Redirection Table (SPCR)
- PCI Express Memory-mapped Configuration Space base address description table (MCFG)
- Differentiated System Description Table (DSDT)

The last of the mentioned tables is particularly important. The DSDT is always referenced by FADT and comprises the list of CPUs, power management features, PCIE root complex and all other embedded controllers description. It often comes with SSDT (Secondary System Description Table) — in single or multiple instances, this structure allows the programmer to logically split various functionalities in the platform description code.

The definitions of the above tables were extended to cover ARM64-specific values and types (e.g., interrupt controllers) — all gathered in ACPICA (ACPI Component Architecture). It is an open source reference code, used and supplemented by OSs. The FreeBSD is maintained to always be on par with the latest version of it. Let's check how the tables are handled in the ARM64 port.

ACPI for ARM64 — the Base Part

The ARM64 SoCs are described by the ACPI tables according to the standards, i.e., the timers and watchdogs are listed in GTDT, the interrupt controller can be found in MADT — currently only GICv2 and GICv3 are supported. Going further to the embedded controllers, the console is described by SPCR (and optionally by the additional DBG2 table) — using ARM SBSA UART (PL011) or the one compatible with 16550 is recommended, although in recent years more types from the ARM world have been added to the list.

Description of the PCIE controller is more complex and must be enclosed in the MCFG and DSDT/SSDT tables. For ARM64 the only allowed type is the one fully compatible with the standardized ECAM generic, supported by pci_host_generic_acpi driver. It is recommended that the new designs comprise an unmodified version of it in the silicon, but for existing products, it is often not possible. Because of that, handling a deviation from the standards is now allowed in the mentioned FreeBSD driver, using the configuration space access quirks. Another solution would be to support a mechanism of executing low-level routines from the firmware via the Secure Monitor Call Calling Convention (SMCCC) interface — currently it is available for Raspberry Pi 4, but this option remains unimplemented in FreeBSD.

It is recommended that the new designs comprise an umodified, generic version of PCIE controllers in the silicon.

Handling of the Embedded Controllers

Embedded controllers that are connected to the SoCs internal bus can be handled twofold ways in the ACPI tables. One option is using 'methods' (instructions) compiled to AML, so the OS can interpret and execute them directly, which is the case of e.g., thermal management, SMBUS or GPIO. Other devices or subsystems that are not explicitly defined in the ACPI specification need to be described by the standard objects that are available for parsing by the OS and obtaining all necessary hardware resources required by the kernel drivers. The latter solution is a key to support non-server ARM64 SoCs in ACPI and is already present in the FreeBSD kernel.

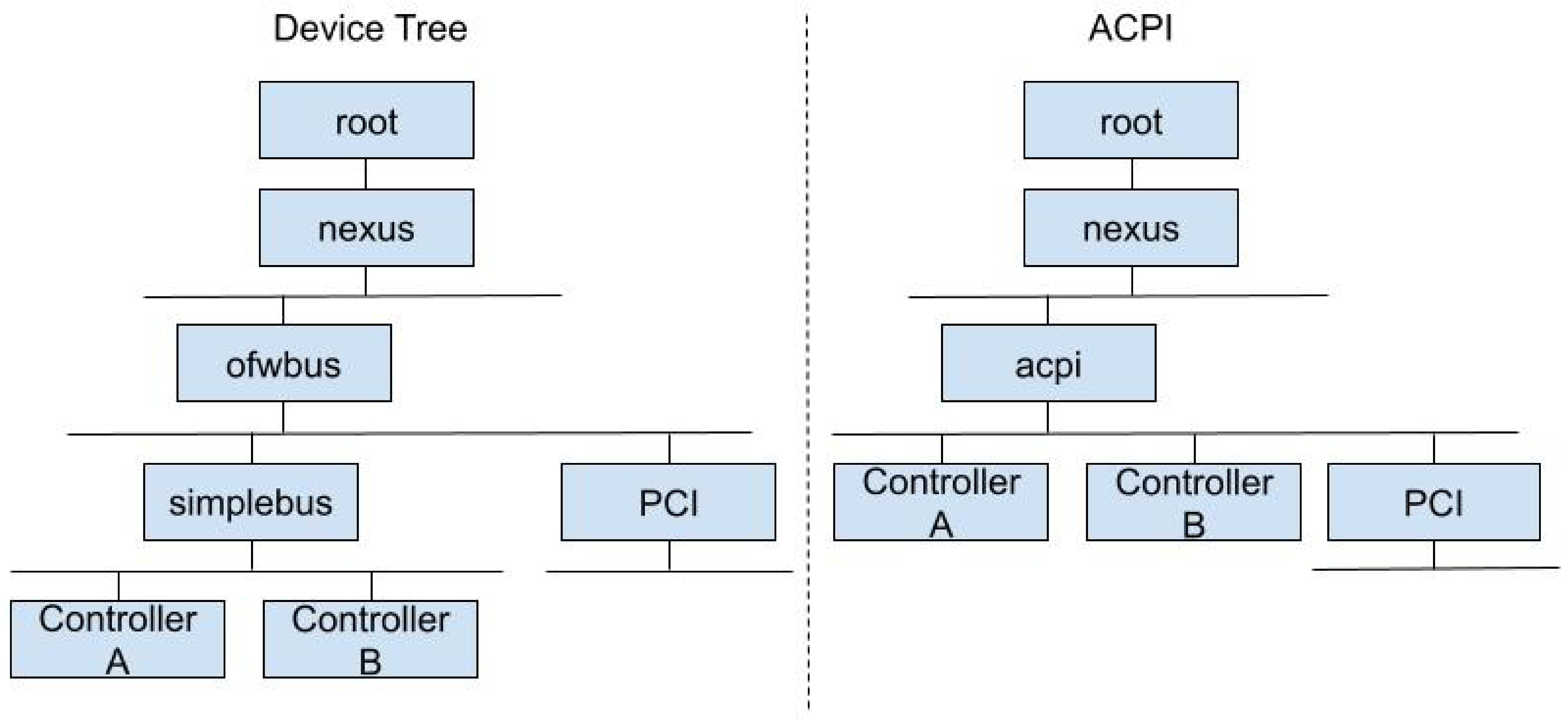

Fig 2. High level comparison of example FreeBSD bus hierarchies in ACPI and Device Tree worlds.

In high level, the FreeBSD bus hierarchies of embedded controllers are similar for both ACPI and DT worlds (ref. Fig. 2). It is helpful for designing the device drivers, as the platform data structures can be filled likewise in each case during the kernel initialization phase. The probed drivers can be later matched by the ACPI _HID field value, which can be treated as an equivalent to the compatible string known from the Device Tree. The other standard types of resources are also handled in an analogous way.

The first two types of ARM64 embedded controllers supported by ACPI in FreeBSD are USB and SATA. The latter is interesting, because it is matched with a driver in a bit of a different way, i.e., by a device class value (ACPI _CLS object; ref. Listing 1).

```
Device (AHC0)
{
    Name (_HID, "LNRO001E")     // _HID: Hardware ID
    Name (_UID, 0x00)           // _UID: Unique ID
    Name (_CCA, 0x01)           // _CCA: Cache Coherency Attribute
    Method (_STA)               // _STA: Device status
    {
        Return (0xF)
    }
    Name (_CLS, Package (0x03)  // _CLS: Class Code
    {
        0x01,
        0x06,
        0x01
    })
```

```
        Name (_CRS, ResourceTemplate ()  // _CRS: Current Resource Settings
        {
            Memory32Fixed (ReadWrite,
                0xF2540000,         // Address Base (MMIO)
                0x00030000,         // Address Length
                )
            Interrupt (ResourceConsumer, Level, ActiveHigh, Exclusive, ,, )
            {
              CP_GIC_SPI_CP0_SATA_H0
            }
        })
    }
```

Listing 1. Example AHCI controller description in ACPI table

The FreeBSD XHCI and AHCI drivers expect fully generic descriptions in DSDT/SSDT. An example of the former is presented in Listing 1. It contains objects referring to a unique ID, information about cache coherency and memory/interrupt resources. All deviations, such as a non-standard register configuration, clocks or power management handling have to be implemented and pre-configured by firmware.

Customizing the ACPI Description

What if the controller requires a custom binding handled by its own, dedicated driver? Until recently it was possible in FreeBSD only in the DT world, using the nodes' properties. However, the ACPI specification defines an optional object called _DSD (Device Specific Data), that can contain the same information. Leveraging the FreeBSD bus hierarchy (ref. Fig 2.), a new generic solution was designed and implemented, to support obtaining controller specific data in a description-agnostic way. Additional helper functions were introduced:

- device_get_property
- device_has_property

They allow access to device specific data provided by the parent bus in a way that the consumer driver can execute exactly the same code path, regardless of the system booting with ACPI or DT. This solution was later extended to cover various types of properties available in both cases.

The FreeBSD XHCI and AHCI drivers expect fully generic descriptions in DSDT/SSDT.

An example of the above was implemented in the SD/MMC subsystem, both in a generic code and a driver for Marvell Xenon controller. The latter was divided into three files: common part and small pieces responsible for attaching either via ACPI or as a child of simplebus. Apart from different DRIVER_MODULE/DEFINE_CLASS_1 macro usage, the latter comprises additional parsing of the regulators and card detect GPIO pins, whereas in ACPI these are set up by firmware.

```
&ap_sdhci0 {
      compatible = "marvell,armada-cp110-sdhci";
      reg = <0x780000 0x300>;
      interrupts = <27 IRQ_TYPE_LEVEL_HIGH>;
      clock-names = "core", "axi";
      clocks = <&CP11X_LABEL(clk) 1 4>, <&CP11X_LABEL(clk) 1 18>;
      dma-coherent;
      bus-width = <8>;
      /*
       * Not stable in HS modes - phy needs "more calibration", so add
       * the "slow-mode" and disable SDR104, SDR50 and DDR50 modes.
      */
      marvell,xenon-phy-slow-mode;
      no-1-8-v;
      no-sd;
      no-sdio;
      non-removable;
      status = "okay";
      vqmmc-supply = <&v_vddo_h>;
};
```

Listing 2. Marvell Xenon SD/MMC controller in Device Tree

```
Device (MMC0)
{
    Name (_HID, "MRVL0002")     // _HID: Hardware ID
    Name (_UID, 0x00)           // _UID: Unique ID
    Name (_CCA, 0x01)           // _CCA: Cache Coherency Attribute
    Method (_STA)               // _STA: Device status
    {
        Return (0xF)
    }
    Name (_CRS, ResourceTemplate ()  // _CRS: Current Resource Settings
    {
        Memory32Fixed (ReadWrite,
            0xF06E0000,         // Address Base (MMIO)
            0x00000300,         // Address Length
            )
        Interrupt (ResourceConsumer, Level, ActiveHigh, Exclusive, ,, )
        {
          48
        }
    })
    Name (_DSD, Package () {
        ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
        Package () {
               Package () { "clock-frequency", 400000000 },
               Package () { "bus-width", 8 },
               Package () { "marvell,xenon-phy-slow-mode", 0x1 },
               Package () { "no-1-8-v", 0x1 },
               Package () { "no-sd", 0x1 },
               Package () { "no-sdio", 0x1  },
               Package () { "non-removable", 0x1  },
         }
     })
 }
```

Listing 3. Marvell Xenon SD/MMC controller in ACPI

Listings 2. and 3. show example DT and ACPI nodes of the same controller instance, in order to demonstrate the similarities of both descriptions. Thanks to the new FreeBSD kernel methods, the controller can successfully operate in all firmware configurations.

Conclusion

With the recent additions to the FreeBSD kernel, the contemporary ARM64 SoCs used in the embedded products can be supported with a similar set of features both with ACPI and the Device Tree. The hierarchical representation of custom controllers turned out to be flexible enough for most kinds of devices and subsystems also in the ACPI case. There are examples that confirm it is possible even with more sophisticated network controllers and the generic MDIO layer. Now there are no limits for FreeBSD to follow this path, especially that the bus architecture allows doing it in a clean and elegant way, as demonstrated in the SD/MMC example.

MARCIN WOJTAS is Head of Engineering at Semihalf and also a FreeBSD src commiter (mw@). He is passionate about embedded software and hardware, and contributor to a number of open source projects, including Linux kernel, Tianocore EDK2 and TF-A.

Write For Us!

Contact Jim Maurer with your article ideas.

(jmaurer@freebsdjournal.com)