ACPI on ARMv8 Servers

ACPI can be used for ARMv8 general purpose servers designed to follow the ARM SBSA (Server Base System Architecture) [0] and SBBR (Server Base Boot Requirements) [1] specifications. Please note that the SBBR can be retrieved simply by visiting [1], but the SBSA is currently only available to those with an ARM login due to ARM IP licensing concerns.

The ARMv8 kernel implements the reduced hardware model of ACPI version 5.1 or later. Links to the specification and all external documents it refers to are managed by the UEFI Forum. The specification is available at http://www.uefi.org/specifications and documents referenced by the specification can be found via http://www.uefi.org/acpi.

If an ARMv8 system does not meet the requirements of the SBSA and SBBR, or cannot be described using the mechanisms defined in the required ACPI specifications, then ACPI may not be a good fit for the hardware.

While the documents mentioned above set out the requirements for building industry-standard ARMv8 servers, they also apply to more than one operating system. The purpose of this document is to describe the interaction between ACPI and Linux only, on an ARMv8 system -- that is, what Linux expects of ACPI and what ACPI can expect of Linux.

Why ACPI on ARM?

Before examining the details of the interface between ACPI and Linux, it is useful to understand why ACPI is being used. Several technologies already exist in Linux for describing non-enumerable hardware, after all. In this section we summarize a blog post [2] from Grant Likely that outlines the reasoning behind ACPI on ARMv8 servers. Actually, we snitch a good portion of the summary text almost directly, to be honest.

The short form of the rationale for ACPI on ARM is:

- ACPIâ€™s byte code (AML) allows the platform to encode hardware behavior, while DT explicitly does not support this. For hardware vendors, being able to encode behavior is a key tool used in supporting operating system releases on new hardware.
- ACPIâ€™s OSPM defines a power management model that constrains what the platform is allowed to do into a specific model, while still providing flexibility in hardware design.
- In the enterprise server environment, ACPI has established bindings (such as for RAS) which are currently used in production systems. DT does not. Such bindings could be defined in DT at some point, but doing so means ARM and x86 would end up using completely different code paths in both firmware and the kernel.
- Choosing a single interface to describe the abstraction between a platform and an OS is important. Hardware vendors would not be required to implement both DT and ACPI if they want to support multiple operating systems. And, agreeing on a single interface instead of being fragmented into per OS interfaces makes for better interoperability overall.
- The new ACPI governance process works well and Linux is now at the same table as hardware vendors and other OS vendors. In fact, there is no longer any reason to feel that ACPI only belongs to Windows or that Linux is in any way secondary to Microsoft in this arena. The move of ACPI governance into the UEFI forum has significantly opened up the specification development process, and currently, a large portion of the changes being made to ACPI are being driven by Linux.

Key to the use of ACPI is the support model. For servers in general, the responsibility for hardware behaviour cannot solely be the domain of the kernel, but rather must be split between the platform and the kernel, in order to allow for orderly change over time. ACPI frees the OS from needing to understand all the minute details of the hardware so that the OS doesnâ€™t need to be ported to each and every device individually. It allows the hardware vendors to take responsibility for power management behaviour without depending on an OS release cycle which is not under their control.

ACPI is also important because hardware and OS vendors have already worked out the mechanisms for supporting a general purpose computing ecosystem. The infrastructure is in place, the bindings are in place, and the processes are in place. DT does exactly what Linux needs it to when working with vertically integrated devices, but there are no good processes for supporting what the server vendors need. Linux could potentially get there with DT, but doing so really just duplicates something that already works. ACPI already does what the hardware vendors need, Microsoft wonâ€™t collaborate on DT, and hardware vendors would still end up providing two completely separate firmware interfaces -- one for Linux and one for Windows.

Kernel Compatibility

One of the primary motivations for ACPI is standardization, and using that to provide backward compatibility for Linux kernels. In the server market, software and hardware are often used for long periods. ACPI allows the kernel and firmware to agree on a consistent abstraction that can be maintained over time, even as hardware or software change. As long as the abstraction is supported, systems can be updated without necessarily having to replace the kernel.

When a Linux driver or subsystem is first implemented using ACPI, it by definition ends up requiring a specific version of the ACPI specification -- it's baseline. ACPI firmware must continue to work, even though it may not be optimal, with the earliest kernel version that first provides support for that baseline version of ACPI. There may be a need for additional drivers, but adding new functionality (e.g., CPU power management) should not break older kernel versions. Further, ACPI firmware must also work with the most recent version of the kernel.

Relationship with Device Tree

ACPI support in drivers and subsystems for ARMv8 should never be mutually exclusive with DT support at compile time.

At boot time the kernel will only use one description method depending on parameters passed from the boot loader (including kernel bootargs).

Regardless of whether DT or ACPI is used, the kernel must always be capable of booting with either scheme (in kernels with both schemes enabled at compile time).

Booting using ACPI tables

The only defined method for passing ACPI tables to the kernel on ARMv8 is via the UEFI system configuration table. Just so it is explicit, this means that ACPI is only supported on platforms that boot via UEFI.

When an ARMv8 system boots, it can either have DT information, ACPI tables, or in some very unusual cases, both. If no command line parameters are used, the kernel will try to use DT for device enumeration; if there is no DT present, the kernel will try to use ACPI tables, but only if they are present. In neither is available, the kernel will not boot. If acpi=force is used on the command line, the kernel will attempt to use ACPI tables first, but fall back to DT if there are no ACPI tables present. The basic idea is that the kernel will not fail to boot unless it absolutely has no other choice.

Processing of ACPI tables may be disabled by passing acpi=off on the kernel command line; this is the default behavior.

In order for the kernel to load and use ACPI tables, the UEFI implementation MUST set the ACPI_20_TABLE_GUID to point to the RSDP table (the table with the ACPI signature "RSD PTR "). If this pointer is incorrect and acpi=force is used, the kernel will disable ACPI and try to use DT to boot instead; the kernel has, in effect, determined that ACPI tables are not present at that point.

If the pointer to the RSDP table is correct, the table will be mapped into the kernel by the ACPI core, using the address provided by UEFI.

The ACPI core will then locate and map in all other ACPI tables provided by using the addresses in the RSDP table to find the XSDT (eXtended System Description Table). The XSDT in turn provides the addresses to all other ACPI tables provided by the system firmware; the ACPI core will then traverse this table and map in the tables listed.

The ACPI core will ignore any provided RSDT (Root System Description Table). RSDTs have been deprecated and are ignored on arm64 since they only allow for 32-bit addresses.

Further, the ACPI core will only use the 64-bit address fields in the FADT (Fixed ACPI Description Table). Any 32-bit address fields in the FADT will be ignored on arm64.

Hardware reduced mode (see Section 4.1 of the ACPI 6.1 specification) will be enforced by the ACPI core on arm64. Doing so allows the ACPI core to run less complex code since it no longer has to provide support for legacy hardware from other architectures. Any fields that are not to be used for hardware reduced mode must be set to zero.

For the ACPI core to operate properly, and in turn provide the information the kernel needs to configure devices, it expects to find the following tables (all section numbers refer to the ACPI 6.1 specification):

- RSDP (Root System Description Pointer), section 5.2.5
- XSDT (eXtended System Description Table), section 5.2.8
- FADT (Fixed ACPI Description Table), section 5.2.9
- DSDT (Differentiated System Description Table), section 5.2.11.1
- MADT (Multiple APIC Description Table), section 5.2.12
- GTDT (Generic Timer Description Table), section 5.2.24
- If PCI is supported, the MCFG (Memory mapped ConFiGuration Table), section 5.2.6, specifically Table 5-31.
- If booting without a console=<device> kernel parameter is supported, the SPCR (Serial Port Console Redirection table), section 5.2.6, specifically Table 5-31.
- If necessary to describe the I/O topology, SMMUs and GIC ITSs, the IORT (Input Output Remapping Table, section 5.2.6, specifically Table 5-31).
- If NUMA is supported, the SRAT (System Resource Affinity Table) and SLIT (System Locality distance Information Table), sections 5.2.16 and 5.2.17, respectively.

If the above tables are not all present, the kernel may or may not be able to boot properly since it may not be able to configure all of the devices available. This list of tables is not meant to be all inclusive; in some environments other tables may be needed (e.g., any of the APEI tables from section 18) to support specific functionality.

ACPI Detection

Drivers should determine their probe() type by checking for a null value for ACPI_HANDLE, or checking .of_node, or other information in the device structure. This is detailed further in the "Driver Recommendations" section.

In non-driver code, if the presence of ACPI needs to be detected at run time, then check the value of acpi_disabled. If CONFIG_ACPI is not set, acpi_disabled will always be 1.

Device Enumeration

Device descriptions in ACPI should use standard recognized ACPI interfaces. These may contain less information than is typically provided via a Device Tree description for the same device. This is also one of the reasons that ACPI can be useful -- the driver takes into account that it may have less detailed information about the device and uses sensible defaults instead. If done properly in the driver, the hardware can change and improve over time without the driver having to change at all.

Clocks provide an excellent example. In DT, clocks need to be specified and the drivers need to take them into account. In ACPI, the assumption is that UEFI will leave the device in a reasonable default state, including any clock settings. If for some reason the driver needs to change a clock value, this can be done in an ACPI method; all the driver needs to do is invoke the method and not concern itself with what the method needs to do to change the clock. Changing the hardware can then take place over time by changing what the ACPI method does, and not the driver.

In DT, the parameters needed by the driver to set up clocks as in the example above are known as "bindings"; in ACPI, these are known as "Device Properties" and provided to a driver via the _DSD object.

ACPI tables are described with a formal language called ASL, the ACPI Source Language (section 19 of the specification). This means that there are always multiple ways to describe the same thing -- including device properties. For example, device properties could use an ASL construct that looks like this: Name(KEY0, "value0"). An ACPI device driver would then retrieve the value of the property by evaluating the KEY0 object. However, using Name() this way has multiple problems: (1) ACPI limits names ("KEY0") to four characters unlike DT; (2) there is no industry wide registry that maintains a list of names, minimizing re-use; (3) there is also no registry for the definition of property values ("value0"), again making re-use difficult; and (4) how does one maintain backward compatibility as new hardware comes out? The _DSD method was created to solve precisely these sorts of problems; Linux drivers should ALWAYS use the _DSD method for device properties and nothing else.

The _DSM object (ACPI Section 9.14.1) could also be used for conveying device properties to a driver. Linux drivers should only expect it to be used if _DSD cannot represent the data required, and there is no way to create a new UUID for the _DSD object. Note that there is even less regulation of the use of _DSM than there is of _DSD. Drivers that depend on the contents of _DSM objects will be more difficult to maintain over time because of this; as of this writing, the use of _DSM is the cause of quite a few firmware problems and is not recommended.

Drivers should look for device properties in the _DSD object ONLY; the _DSD object is described in the ACPI specification section 6.2.5, but this only describes how to define the structure of an object returned via _DSD, and how specific data structures are defined by specific UUIDs. Linux should only use the _DSD Device Properties UUID [5]:

- UUID: daffd814-6eba-4d8c-8a91-bc9bbf4aa301
- https://www.uefi.org/sites/default/files/resources/_DSD-device-properties-UUID.pdf

The UEFI Forum provides a mechanism for registering device properties [4] so that they may be used across all operating systems supporting ACPI. Device properties that have not been registered with the UEFI Forum should not be used.

Before creating new device properties, check to be sure that they have not been defined before and either registered in the Linux kernel documentation as DT bindings, or the UEFI Forum as device properties. While we do not want to simply move all DT bindings into ACPI device properties, we can learn from what has been previously defined.

If it is necessary to define a new device property, or if it makes sense to synthesize the definition of a binding so it can be used in any firmware, both DT bindings and ACPI device properties for device drivers have review processes. Use them both. When the driver itself is submitted for review to the Linux mailing lists, the device property definitions needed must be submitted at the same time. A driver that supports ACPI and uses device properties will not be considered complete without their definitions. Once the device property has been accepted by the Linux community, it must be registered with the UEFI Forum [4], which will review it again for consistency within the registry. This may require iteration. The UEFI Forum, though, will always be the canonical site for device property definitions.

It may make sense to provide notice to the UEFI Forum that there is the intent to register a previously unused device property name as a means of reserving the name for later use. Other operating system vendors will also be submitting registration requests and this may help smooth the process.

Once registration and review have been completed, the kernel provides an interface for looking up device properties in a manner independent of whether DT or ACPI is being used. This API should be used [6]; it can eliminate some duplication of code paths in driver probing functions and discourage divergence between DT bindings and ACPI device properties.

Programmable Power Control Resources

Programmable power control resources include such resources as voltage/current providers (regulators) and clock sources.

With ACPI, the kernel clock and regulator framework is not expected to be used at all.

The kernel assumes that power control of these resources is represented with Power Resource Objects (ACPI section 7.1). The ACPI core will then handle correctly enabling and disabling resources as they are needed. In order to get that to work, ACPI assumes each device has defined D-states and that these can be controlled through the optional ACPI methods _PS0, _PS1, _PS2, and _PS3; in ACPI, _PS0 is the method to invoke to turn a device full on, and _PS3 is for turning a device full off.

There are two options for using those Power Resources. They can:

- be managed in a _PSx method which gets called on entry to power state Dx.
- be declared separately as power resources with their own _ON and _OFF methods. They are then tied back to D-states for a particular device via _PRx which specifies which power resources a device needs to be on while in Dx. Kernel then tracks number of devices using a power resource and calls _ON/_OFF as needed.

The kernel ACPI code will also assume that the _PSx methods follow the normal ACPI rules for such methods:

- If either _PS0 or _PS3 is implemented, then the other method must also be implemented.
- If a device requires usage or setup of a power resource when on, the ASL should organize that it is allocated/enabled using the _PS0 method.
- Resources allocated or enabled in the _PS0 method should be disabled or de-allocated in the _PS3 method.
- Firmware will leave the resources in a reasonable state before handing over control to the kernel.

Such code in _PSx methods will of course be very platform specific. But, this allows the driver to abstract out the interface for operating the device and avoid having to read special non-standard values from ACPI tables. Further, abstracting the use of these resources allows the hardware to change over time without requiring updates to the driver.

Clocks

ACPI makes the assumption that clocks are initialized by the firmware -- UEFI, in this case -- to some working value before control is handed over to the kernel. This has implications for devices such as UARTs, or SoC-driven LCD displays, for example.

When the kernel boots, the clocks are assumed to be set to reasonable working values. If for some reason the frequency needs to change -- e.g., throttling for power management -- the device driver should expect that process to be abstracted out into some ACPI method that can be invoked (please see the ACPI specification for further recommendations on standard methods to be expected). The only exceptions to this are CPU clocks where CPPC provides a much richer interface than ACPI methods. If the clocks are not set, there is no direct way for Linux to control them.

If an SoC vendor wants to provide fine-grained control of the system clocks, they could do so by providing ACPI methods that could be invoked by Linux drivers. However, this is NOT recommended and Linux drivers should NOT use such methods, even if they are provided. Such methods are not currently standardized in the ACPI specification, and using them could tie a kernel to a very specific SoC, or tie an SoC to a very specific version of the kernel, both of which we are trying to avoid.

Driver Recommendations

DO NOT remove any DT handling when adding ACPI support for a driver. The same device may be used on many different systems.

DO try to structure the driver so that it is data-driven. That is, set up a struct containing internal per-device state based on defaults and whatever else must be discovered by the driver probe function. Then, have the rest of the driver operate off of the contents of that struct. Doing so should allow most divergence between ACPI and DT functionality to be kept local to the probe function instead of being scattered throughout the driver. For example:

```
static int device_probe_dt(struct platform_device *pdev)
{
	/* DT specific functionality */
	...
}

static int device_probe_acpi(struct platform_device *pdev)
{
	/* ACPI specific functionality */
	...
}

static int device_probe(struct platform_device *pdev)
{
	...
	struct device_node node = pdev->dev.of_node;
	...

	if (node)
		ret = device_probe_dt(pdev);
	else if (ACPI_HANDLE(&pdev->dev))
		ret = device_probe_acpi(pdev);
	else
		/* other initialization */
		...
	/* Continue with any generic probe operations */
	...
}
```

DO keep the MODULE_DEVICE_TABLE entries together in the driver to make it clear the different names the driver is probed for, both from DT and from ACPI:

```
static struct of_device_id virtio_mmio_match[] = {
        { .compatible = "virtio,mmio", },
        { }
};
MODULE_DEVICE_TABLE(of, virtio_mmio_match);

static const struct acpi_device_id virtio_mmio_acpi_match[] = {
        { "LNRO0005", },
        { }
};
MODULE_DEVICE_TABLE(acpi, virtio_mmio_acpi_match);
```

ASWG

The ACPI specification changes regularly. During the year 2014, for instance, version 5.1 was released and version 6.0 substantially completed, with most of the changes being driven by ARM-specific requirements. Proposed changes are presented and discussed in the ASWG (ACPI Specification Working Group) which is a part of the UEFI Forum. The current version of the ACPI specification is 6.1 release in January 2016.

Participation in this group is open to all UEFI members. Please see http://www.uefi.org/workinggroup for details on group membership.

It is the intent of the ARMv8 ACPI kernel code to follow the ACPI specification as closely as possible, and to only implement functionality that complies with the released standards from UEFI ASWG. As a practical matter, there will be vendors that provide bad ACPI tables or violate the standards in some way. If this is because of errors, quirks and fix-ups may be necessary, but will be avoided if possible. If there are features missing from ACPI that preclude it from being used on a platform, ECRs (Engineering Change Requests) should be submitted to ASWG and go through the normal approval process; for those that are not UEFI members, many other members of the Linux community are and would likely be willing to assist in submitting ECRs.

Linux Code

Individual items specific to Linux on ARM, contained in the Linux source code, are in the list that follows:

ACPI_OS_NAME
: This macro defines the string to be returned when an ACPI method invokes the _OS method. On ARM64 systems, this macro will be "Linux" by default. The command line parameter acpi_os=<string> can be used to set it to some other value. The default value for other architectures is "Microsoft Windows NT", for example.

ACPI Objects

Detailed expectations for ACPI tables and object are listed in the file Documentation/arm64/acpi_object_usage.rst.

References

[0] http://silver.arm.com
: document ARM-DEN-0029, or newer: "Server Base System Architecture", version 2.3, dated 27 Mar 2014

[1] http://infocenter.arm.com/help/topic/com.arm.doc.den0044a/Server_Base_Boot_Requirements.pdf
: Document ARM-DEN-0044A, or newer: "Server Base Boot Requirements, System Software on ARM Platforms", dated 16 Aug 2014

[2] http://www.secretlab.ca/archives/151,
: 10 Jan 2015, Copyright (c) 2015, Linaro Ltd., written by Grant Likely.

[3] AMD ACPI for Seattle platform documentation
: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Seattle_ACPI_Guide.pdf

[4] http://www.uefi.org/acpi
: please see the link for the "ACPI _DSD Device Property Registry Instructions"

[5] http://www.uefi.org/acpi
: please see the link for the "_DSD (Device Specific Data) Implementation Guide"

[6] Kernel code for the unified device
: property interface can be found in include/linux/property.h and drivers/base/property.c.

Authors

- Al Stone <al.stone@linaro.org>
- Graeme Gregory <graeme.gregory@linaro.org>
- Hanjun Guo <hanjun.guo@linaro.org>
- Grant Likely <grant.likely@linaro.org>, for the "Why ACPI on ARM?" section