tianocore

UEFI & EDK II TRAINING

Advanced Configuration and Power Interface (ACPI) Basics

tianocore.org

tianocore

Agenda

Introduction / Overview

System, Device Power States

Device Enumeration, Configuration

ACPI Namespace

ACPI Hardware, Event Model

ACPI Tables DDST and SSDT

Platform Communication Channel (PCC)

Operation Region (OpRegion) Handling

Platform Runtime Mechanism (PRM)

ACPI - Introduction

Advanced **C**onfiguration and **P**ower **I**nterface is an open standard Interface that OS can use to discover and configure devices and perform Power Management. E.g., Putting unused devices to sleep, Plug and Play device auto configuration, Status monitoring.

ACPI aims to replace the BIOS controlled standard interfaces such as

- Advanced Power Management (APM)
- Multi Processor Specification (MPS)
- PCI BIOS Specification
- Plug and Play (PnP) BIOS specification

ACPI Standard originally developed by ACPI Special Interest Group (ACPI SIG) formed by Intel, Microsoft, Toshiba, HP, Huawei and Phoenix. Later merged with UEFI Forum.

UEFI published latest ACPI revision 6.4 on January 2021

ACPI Specification Releases

Version	Published On	High level Revision Description
1.0	December 1996	Initial Version. Replacement for PnP, APM, MPS
2.0	August 2000	Added 64-bit support, processor P-states
3.0	September 2004	Added support for PCIe, NUMA, SATA (inc _OSC, _DSM), new thermal model
4.0	June 2009	Thermal model enhancements, clock domains, USB3/XHCI
5.0	November 2011	HW-reduced ACPI, FPDT, Arm support (UEFI takes ownership)
6.0	April 2015	LPIT, ASL symbolic operators, persistent memory
6.1	January 2016	NVDIMM _DSM, clarifications
6.2	May 2017	Added HMAT, NVST updates
6.3	January 2019	Minor changes
6.4	January 2021	Corrections, USB4 Host, Added new device structures

ACPI Overview

Hence HW Configuration and Power Management are moved from BIOS to OS level, OSPM is responsible for

- Handling motherboard device configuration events
- Controlling the power, Performance, and Thermal status of the system based on
  - User preference
  - Application requests
  - OS imposed Quality of Service

OSPM – Operating System-directed configuration and Power Management

Functional areas covered by ACPI specification are

- System Power Management / System Power States
- Device Power Management
- Processor Power Management
- Configuration / Plug and Play
- System Events
- Battery Management
- Thermal Management
- Embedded Controller (EC)
- SMBus Controller

ACPI is the key element in OSPM, which provides low-level interfaces that allow OSPM to perform all these functions.

Global System States and Transitions

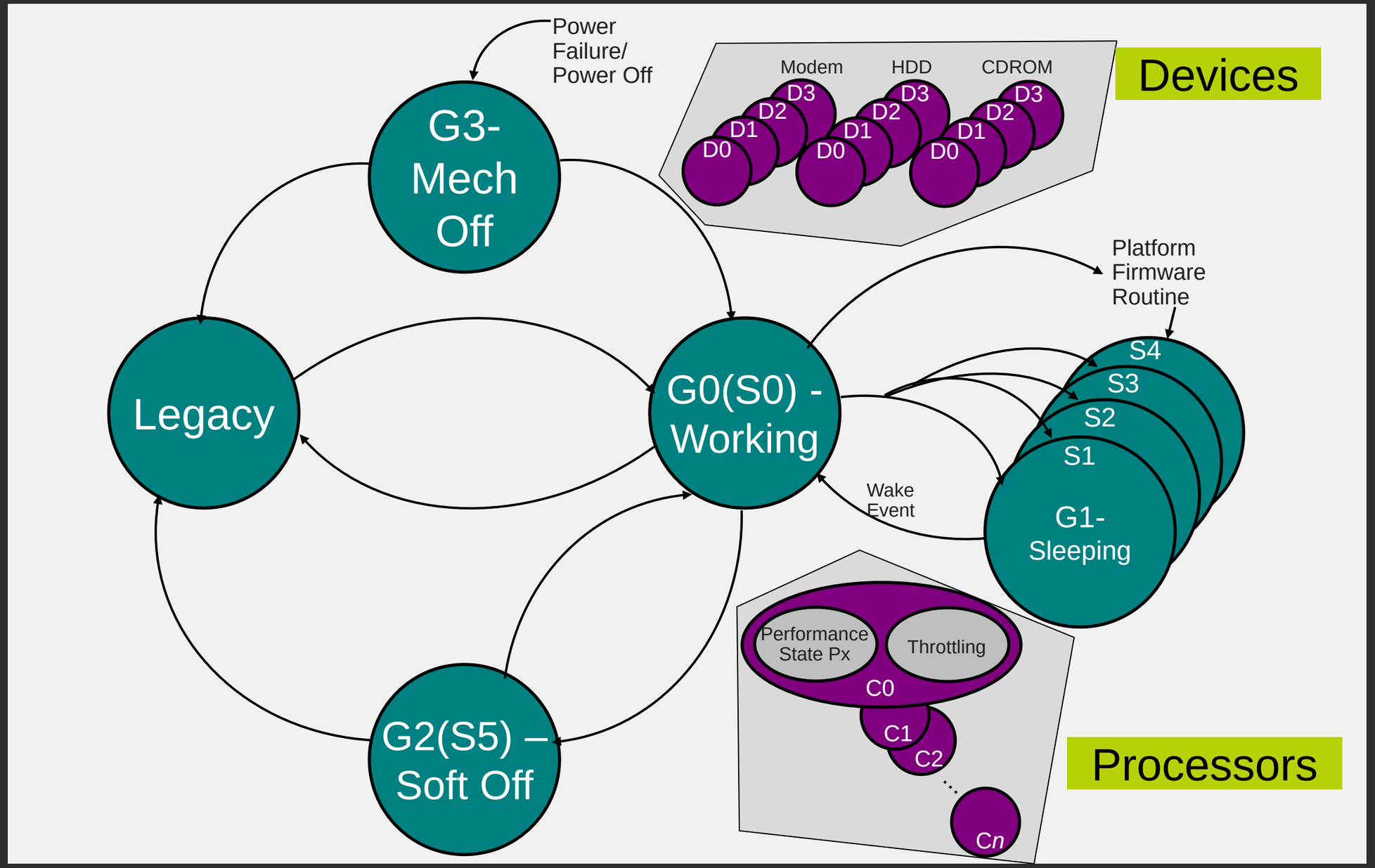


Fig. 3.1: Global States and Their Transitions

System Power Management & Power States

OS directs all system and device power state transitions.

OS uses information from applications and user settings and put the system as a whole into a low-power state.

System states transit between

G3 (Mechanical Off) ←→ G0 (S0) – Working ←→G1 (Sx) / G2 (S5) States

Platform Power management characteristics will vary between Mobile PCs, Desktop PCs and Server PCs

System Power States (ACPI)

S0 (G0)- Working State (includes S0ix)

S1 (G1)- Legacy desktop standby state, deprecated

S3 (G1)- Suspend to RAM (Sleep)

S4 (G1)- Suspend to Disk (Hibernate)

S5 (G2)- Soft Off (Shutdown)

G3- Mechanical Off (power switch, battery removal)

Other:

- Windows: Connected/Modern Standby,
- S0ix, DRIPS

Device Power Management & Power States

To manage power of all devices in the system, OS needs standard methods of sending commands to a device. Defining the standards for each IO interconnect creates a baseline level of power management support that the OS can utilize.

Device states transit between

D3/D2 (Off) ←→ D0 ←→ D1

Processor Power Management & Performance States

To save power in the working state, the OS puts CPU into low-power states (C1, C2 & C3).

Device and Processor performance states define the performance states.

Performance States (Px) are power consumption and capability states within the active executing states (C0 for processor and D0 for device)

Processor & Device Power States

Processor: C-states

- Only 3 C-states can be described to the platform
  - C0 is implicit
  - C1-C3 are logical C-states, mapped to true C-states via platform (BIOS, PCU)

Device: D-states

- D0-D3 supported, implemented in a bus-specific manner
- PCI (and only PCI) differentiates between $D3_{hot}$ and $D3_{cold}$

Plug and Play Configuration

ACPI provides information that enable OSPM to configure the required resources of Motherboard devices along with their dynamic insertion and removal.

ACPI Definition Blocks (DSDT, SSDT) describe motherboard devices in ACPI Namespace.

ACPI Namespace contains hardware IDs, resources that the device could occupy, an object that reports the resources that are currently used by the device, and objects for configuring those resources.

OS (OSPM) uses these ACPI Namespace information to configure the motherboard devices.

System Events

ACPI uses ‘Event Status Register & Event Enable Register’ to make up a general event model for below event notifications

- Plug and Play
- Thermal
- Power Management

When an event occurs, the core logic sets a bit in the Event Status Register to indicate the event if the corresponding bit in the Event Enable Register is set and assert the SCI interrupt to signal the OS.

When OS receives this interrupt, it will run the control methods corresponding to any bits set in the Event Status Register.

Battery Management

Batteries must comply with ACPI Standards

An ACPI-compatible battery device needs either a Smart Battery subsystem interface or a Control Method Battery interface.

Smart Battery: This is controlled by the OS directly through Embedded Controller (EC)

Control Method Battery: This is accessed by AML code control methods.

Both the interfaces provide a mechanism for the OS to query information from the platform’s battery system. The information like full charged capacity, present battery capacity, rate of discharge and other measures of the battery condition.

Thermal Management

It is necessary for the platform to mandate cooling actions. ACPI allows the OS to control Thermal conditions of the platform and take actions accordingly.

There are two types of cooling methods.

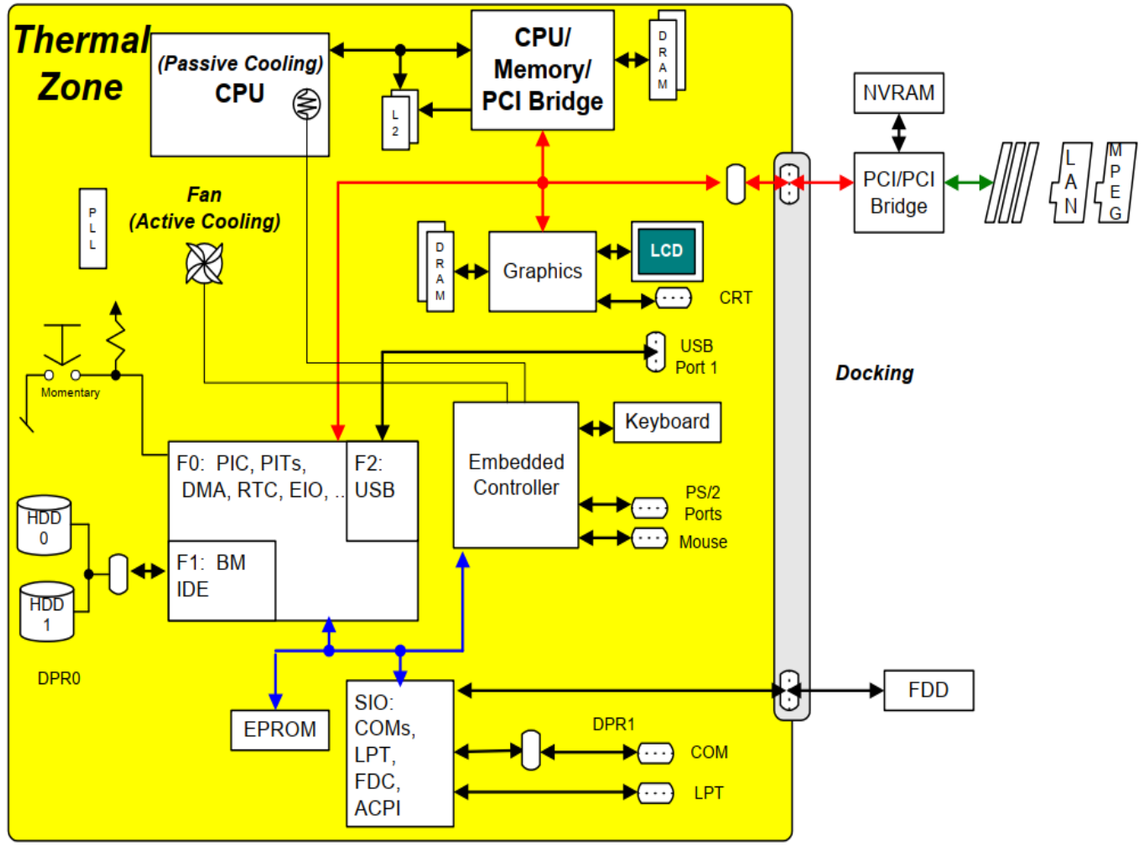
- Passive Cooling
  - OS reduces the power consumption of devices to reduce the temperature of the machine. Which also reduces the system performance.

- Active Cooling
  - OS increases the power consumption of the system by turning on system fan to reduce the temperature of the machine.

OSPM can make cooling decisions based on application load on the CPU as well as the thermal heuristics of the system.

OSPM can also gracefully shutdown the computer in case of high temperature emergencies.

Thermal Zone



The entire PC is one large Thermal Zone. And can be partitioned into several logical thermal zones if necessary.

This picture is example of whole notebook covered as one large thermal zone.

Next slide picture is example for multiple logical thermal zones.

Fig. 3.7 Thermal Zone

Flow of a Thermal Event

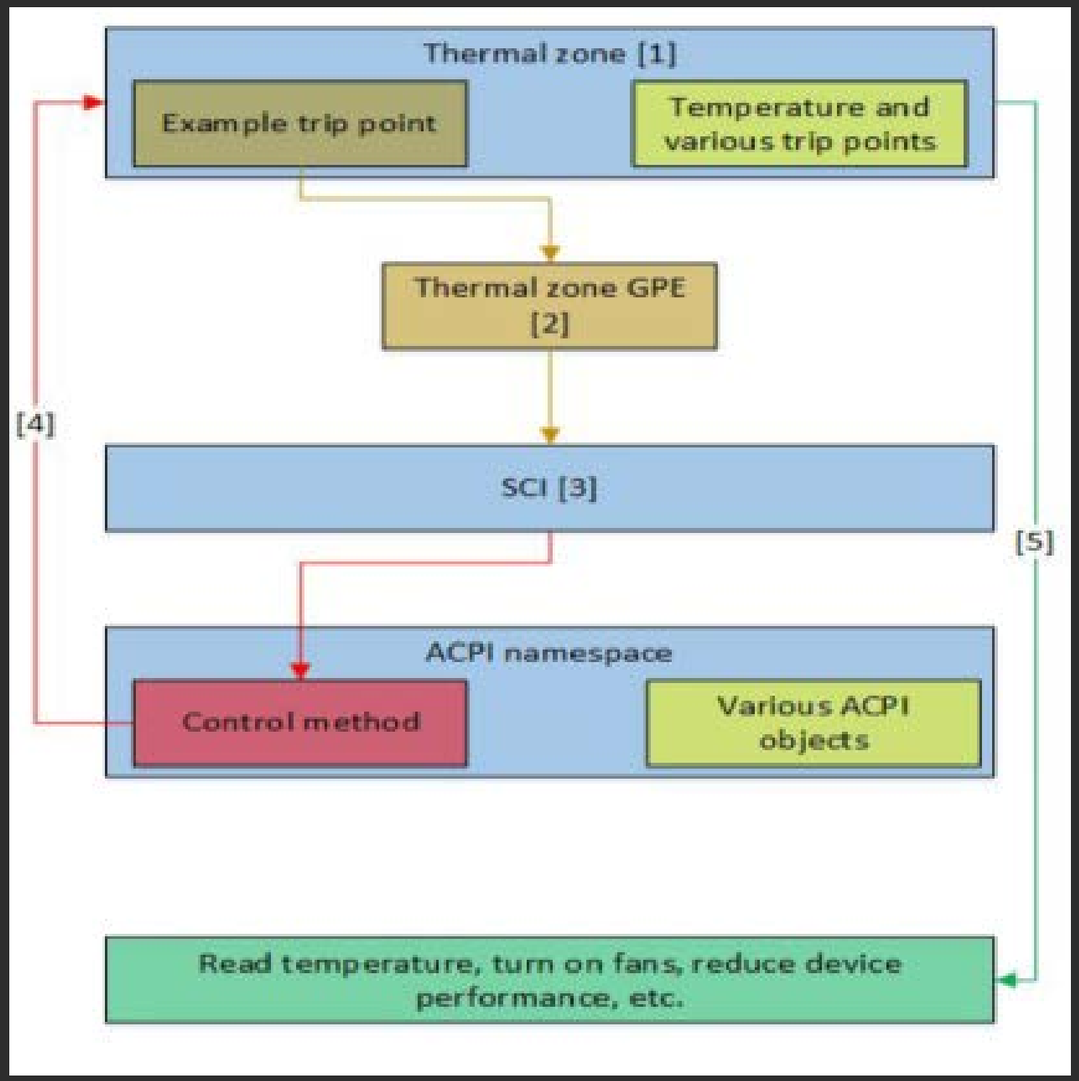


Fig. 5: Runtime Thermal Event

When the system initially finds a thermal zone[1] in the namespace, it loads the thermal zone handler to evaluate the thermal zone to determine the temperature and trip points.

When the temperature reaches a trip point during runtime, a general purpose event GPE [2] occurs

The thermal zone event cause an interrupt via the ACPI system control interrupt (SCI) [3] to occur.

When the OS receives the interrupt, the handler searches the namespace for the control method object[4] corresponding to the GPE interrupt.

Upon finding it, the handler evaluates that object.

The thermal zone handler then takes whatever actions are necessary to handle the event [5].

Embedded Controller (EC)

Embedded Controllers are the general class of microcontrollers used to support OEM specific implementations.

ACPI defines standard H/W & S/W communication interfaces between OS driver and EC. Which allows any OS to provide a standard driver that can directly communicate with EC in the system.

Most common interface incorporated between Host and EC is a Standard Keyboard controller interface. It is accessed at IO Port 0x60 and 0x64.

- IO Port 0x60 – Termed as data register, allows bi-directional data transfer to and from the Host and EC.
- IO Port 0x64 - Termed as Command/Status register. It returns a port status information upon a read, and generates a command sequence to the EC upon a write.

EC controls battery management, Thermal management of the system and more.

System Management Bus (SMBus)

SMBus is a low-speed two-wire interface bus based upon I2C protocol. That provides positive addressing for devices, as well as bus arbitration.

ACPI – AML code uses this address space to access SMBus devices.

See Chapter 13 of the ACPI Specification

ACPI Subsystem

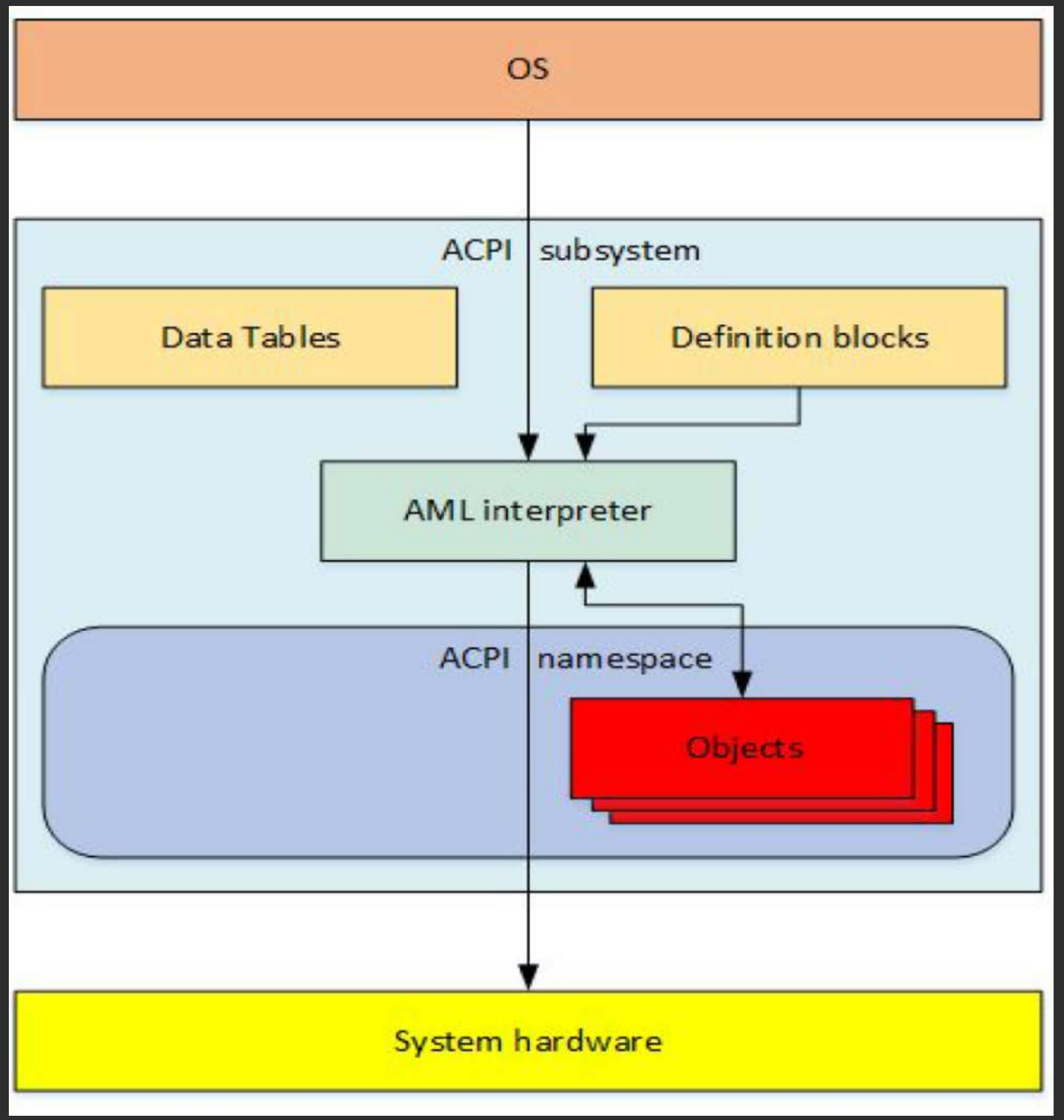


Fig. 2.: ACPI Structure

The ACPI Subsystem consists of two types of data structures: data tables and definition blocks.

Upon initialization, the AML interpreter extracts the byte code in the definition blocks as enumerable objects.

This collection of enumerable objects forms the OS construct called ACPI namespace.

Objects can either have a directly defined value or must be evaluated and interpreted by the AML interpreter.

The AML interpreter, directed by the OS, evaluates objects and interfaces with system hardware to perform necessary operations.

ACPI Initialization

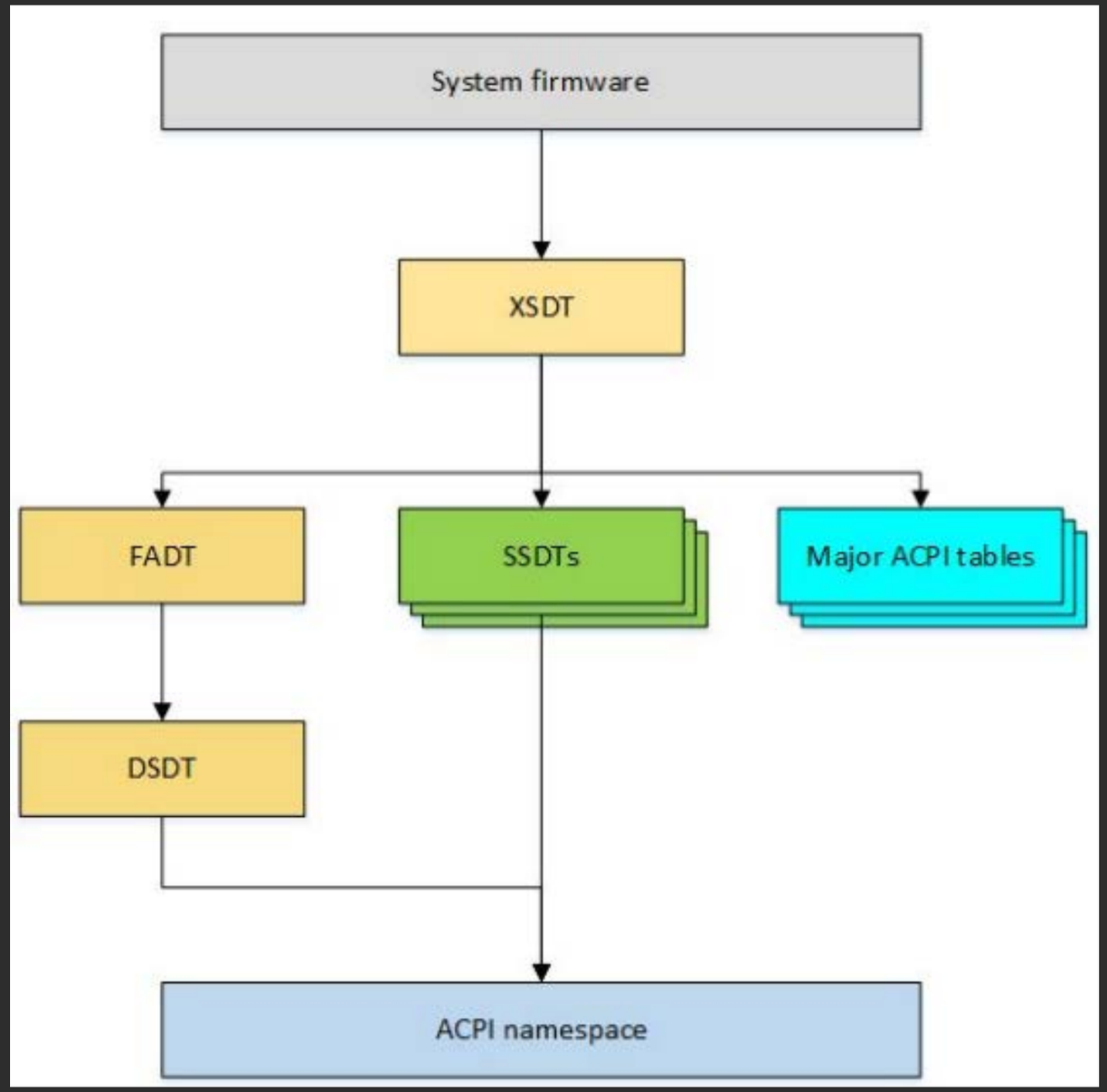


Fig. 4.: ACPI Initiation

1. Power up, Initialization, Self tests
2. System FW Updates ACPI Tables
3. XSDT – (Extended Root System Description)
4. XSDT → FADT (Fixed ACPI Desc. Table)
5. FADT → DSDT (Differentiated System Desc. Table)
6. DSDT – Beginning of Name space
7. ACPI Subsystem consumes DSDT
8. XSDT → SSDT (Secondary System Desc. Table)

System Firmware ACPI

Provides capability to enumerate, configure, and power manage motherboard-based devices

- System bus and device hierarchy is represented in a firmware construct known as the ACPI namespace
- Namespace also includes definition and control of power planes and thermal zones

Defines Hardware registers - implemented in chipset silicon

- Known as “Fixed Hardware”

Defines ACPI Firmware interfaces

- Configuration tables
  - Describes location of fixed hardware and other info e.g. MADT, DBGP
- “Definition Blocks” which constitute the ACPI Namespace
  - Implements “Generic Hardware” - allows OEMs to implement HW features as they deem appropriate through invocation/evaluation of object in the ACPI Namespace a.k.a Control Methods
  - Encoded in a byte code stream known as AML
  - Interpreted at run time by an OS entity known as the AML interpreter
  - Source language (ASL) is compiled to AML and flashed in firmware along with the configuration tables

MADT - Multiple APIC Descript Table
DBGP – Debug Port Table

Device Enumeration

ACPI is used to describe devices that:

- Cannot be discovered via any other standard enumeration mechanism (e.g. legacy devices)
- Need to be described to the OS for boot purposes, prior to availability of ACPI Subsystem (e.g. APICs, debug ports, embedded controller)

ACPI is used to augment discoverable device information with additional platform-specific information

- Interrupt routing
- Connections between devices, controllers (serial bus types, GPIOs)
- Physical connector locations
- Power Resource dependencies, controls

Device Configuration

For legacy or ACPI-enumerated devices, ACPI can be used to:

- Manage resource allocation
- Enable/disable
- Set device power states
- Manage hot-plug events

For augmented device descriptions, some of these can be used to support/supplement OS mechanisms

Table Discovery (1)

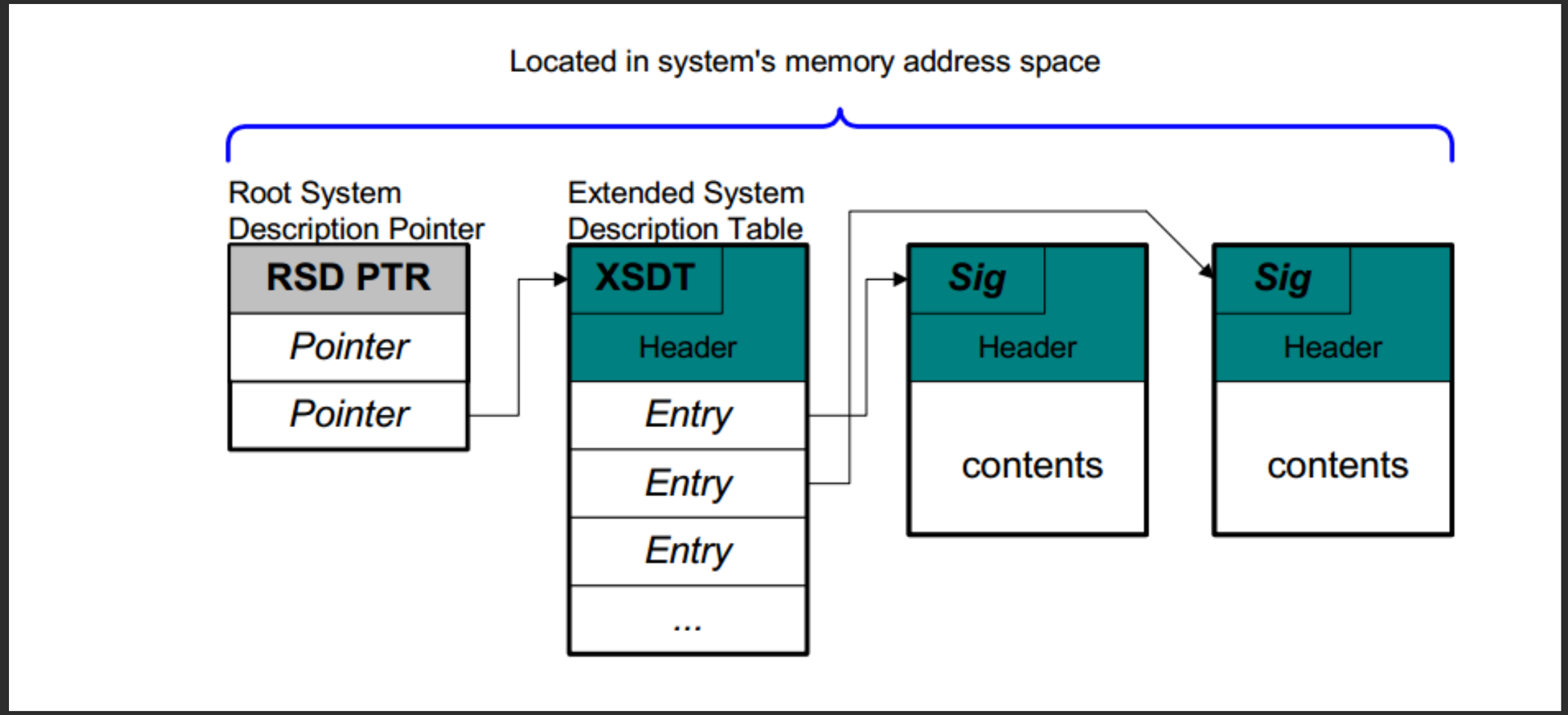


Fig. 5.1: Root System Description Pointer and Table

Table Discovery (2)

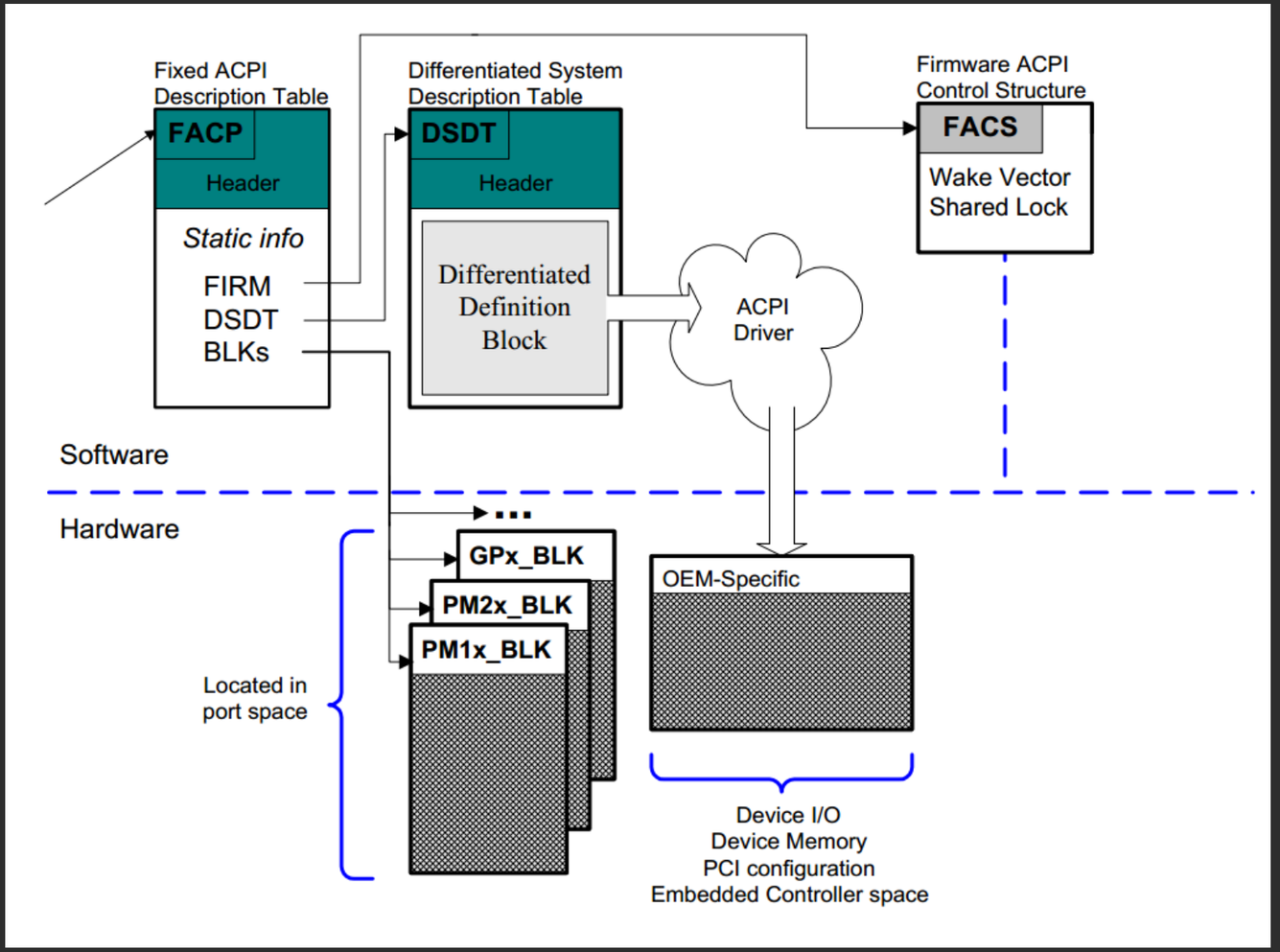


Fig. 5.2: Overall of the System Description Table Architecture

Static ACPI Tables

Static tables are provided for a number of purposes:

Describe generic ACPI HW (FACP/FADT)

- Fixed ACPI Description Table, signature - “FACP”

Pointers to other tables (RSDT/XSDT)

- Root System Description Table, Extended System Description Table

Info needed before AML Interpreter is available (MADT, ECDT, BGRT, NFIT)

- Multiple APIC Description Table
- Embedded Controller Boot Resources Table
- Boot Graphics Resource Table
- NVDIMM Firmware Interface Table

System Firmware/OS shared structure (FACS)

- Firmware ACPI Control Structure

Dynamic Performance Data (FPDT)

- Firmware Performance Data Table

Tables defined outside of the ACPI spec, but signatures reserved by ACPI (HPET, DBG2)

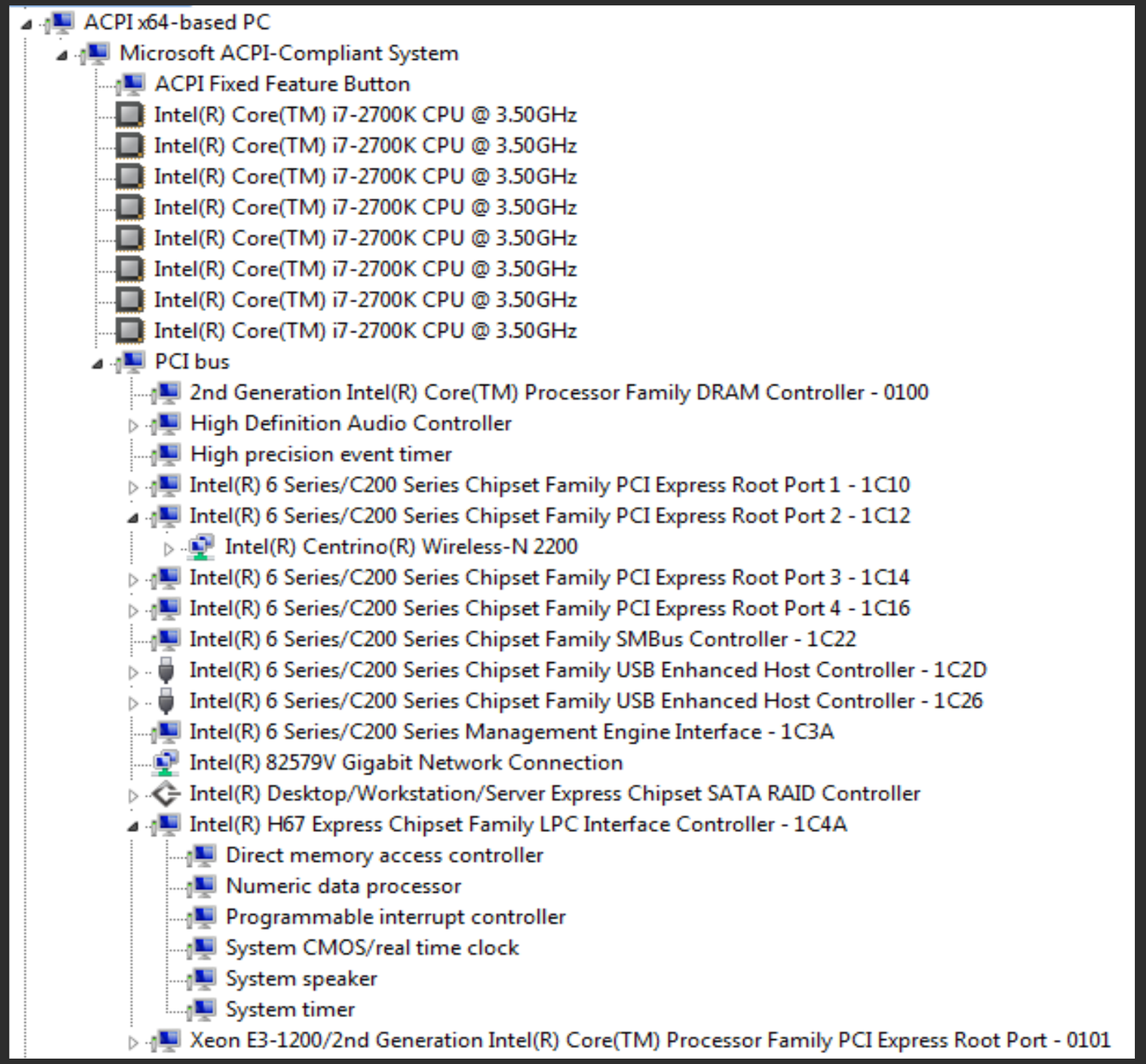
- IA-PC High Precision Event Timer Table, Debug Port Table 2.

Namespace

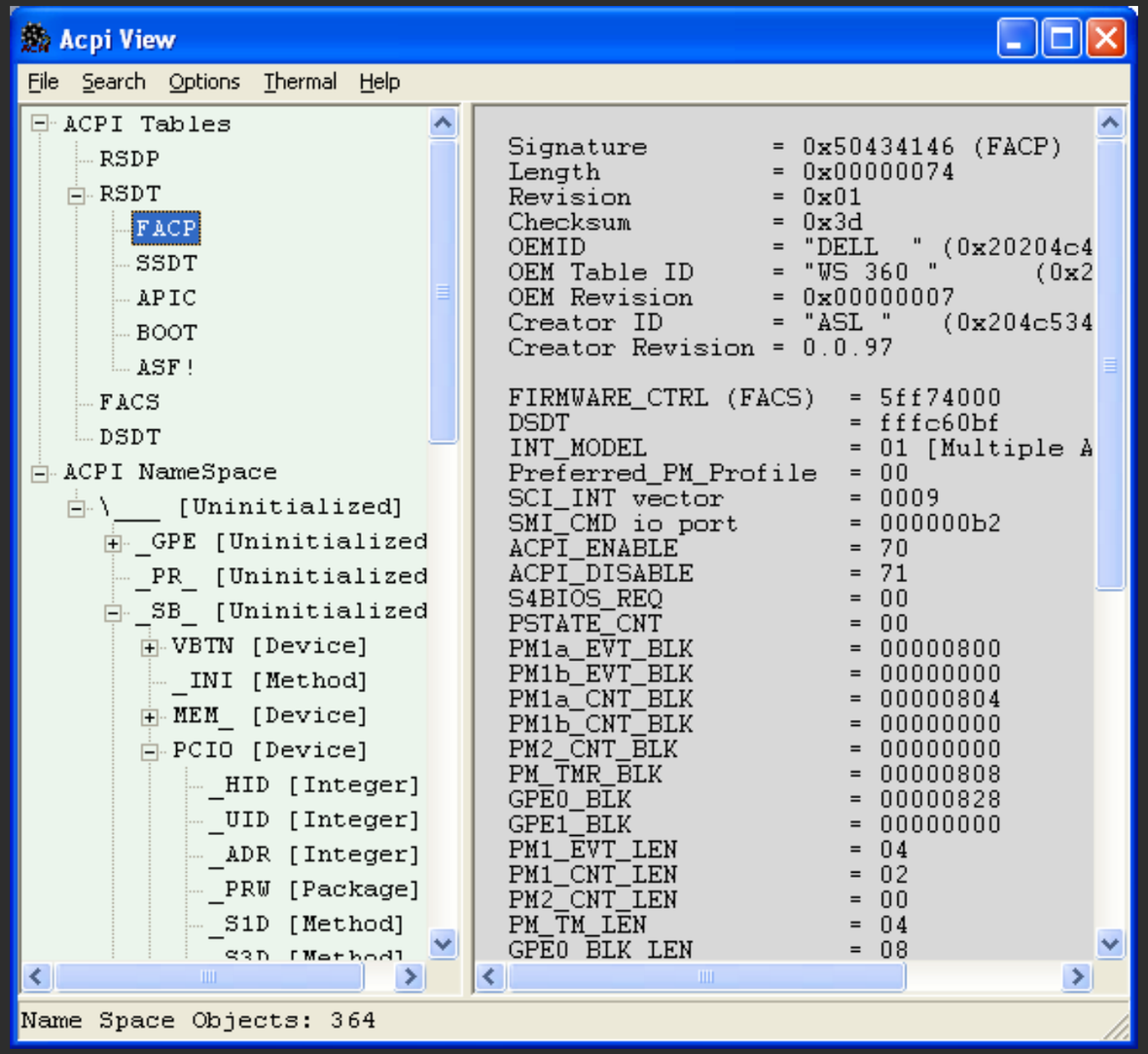
ACPI Namespace is a firmware-provided hierarchical view of the hardware platform

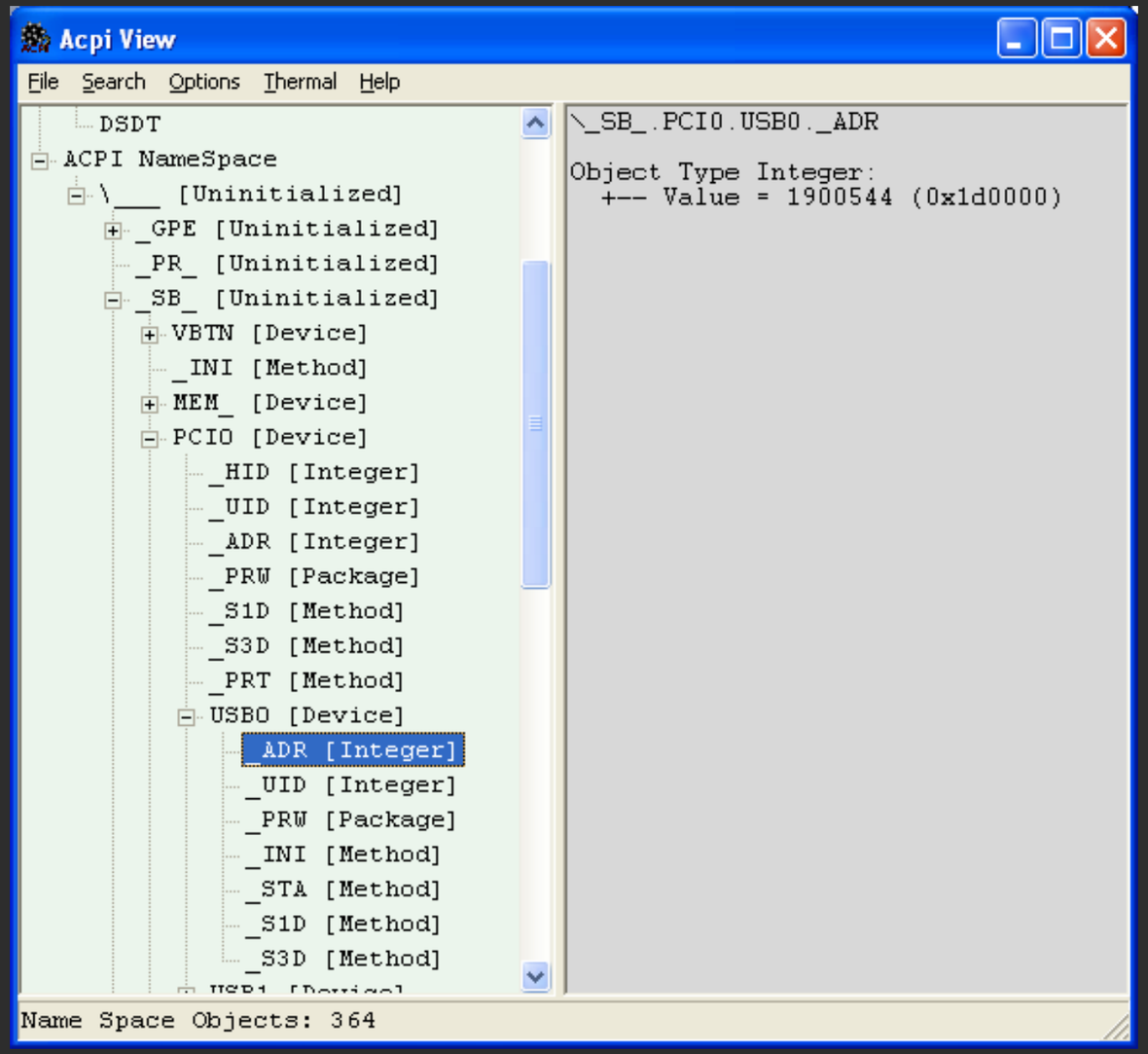
It includes ProcessorObject and DeviceObject declarations, PowerResource declarations, GPE event handlers, ThermalZone mappings

Looks suspiciously like the Windows Device Manager “Devices by Connection” view



View ACPI Tables Windows Platforms





RW - Read & Write Utility 1.7 – Download Link

Obtaining the Namespace

There are several ways to get a dump of a machine's ACPI Namespace:

- Read/WriteEverything
  - Freeware tool, very useful for other things as well
- Windows kernel debugger
  - !amli dns
- Microsoft ASL compiler (asl.exe)
  - Can also extract tables from a live machine, and disassemble
- ACPI Component Architecture
  - AcpiDump, AcpiXtract, iASL (disassembly)
  - https://acpica.org/downloads/binary-tools

tianocore

Fixed Hardware

ACPI Hardware Model includes registers for:

- Sleep state transitions
- Processor power state transitions (legacy)
- Power Management Timer, RTC
- SCI events (control, status)
  - Fixed-feature events, like Power, Sleep buttons
  - General Purpose events (GPIOs)

Alternate Model: ACPI Reduced Hardware

OS Layer

Primarily two OS implementations

- Windows, the de facto standard
- ACPI CA, used by everyone else
  - OSX, Linux, HP-UX,
  - Maintained by SSP/OTC

Components include AML interpreter, OS-specific interfaces throughout the device stacks

- Windows loads an ACPI filter driver above the function driver for each DeviceObject described in the namespace

Differentiated System Description Table (DSDT)

The DSDT contains the Differentiated Definition Block, which supplies the implementation and configuration information about the base system.

An OEM must supply a DSDT to an ACPI-compatible OS. The OS always inserts the DSDT information into the ACPI Namespace at system boot time and never removes it.

The OS dynamically changes the contents of the namespace at run-time by loading definition blocks from the ACPI Tables that reside in the ACPI system firmware.

Secondary System Description Table (SSDT)

SSDTs are a continuation of the DSDT. Multiple SSDTs can be used as part of a platform description.

After the DSDT is loaded into the ACPI Namespace, each secondary description table listed in the RSDT/XSDT with a unique OEM Table ID is loaded.

This allows the OEM to provide the base support in one table, while adding smaller system options in other tables.

OEM might put dynamic object definitions into a secondary table such that the firmware can construct the dynamic information at boot without needing to edit the static DSDT. A SSDT can only rely on the DSDT being loaded prior to it.

tianocore

ACPI definition blocks

The ACPI subsystem processes the DSDT & SSDTs and begins building the namespace from the ACPI definition blocks.

ACPI Definition Blocks, describe motherboard devices in a hierarchical format called the ACPI namespace.

The OS enumerates motherboard devices simply by reading through the ACPI Namespace looking for devices with hardware IDs.

Each device enumerated by ACPI includes ACPI-defined objects in the ACPI Namespace that report the hardware resources that the device could occupy, an object that reports the resources that are currently used by the device, and objects for configuring those resources.

OEMs and platform firmware vendors write definition blocks using the ACPI Source Language (ASL) and use a translator to produce the byte stream encoding called “Definition Block Encoding”.

Platform Communications Channel (PCC)

The platform communication channel is a generic mechanism for OSPM to communicate with an entity in the platform (e.g. a platform controller, or a Baseboard Management Controller (BMC)). Neither the entity that OSPM communicates with, nor any aspects of the information passed back and forth is defined in this section. That information is defined by the actual interface that that employs PCC register address space as the communication channel.

PCC defines a new address space type (PCC Space, 0xA), which is implemented as one or more
independent communications channels, or subspaces.

tianocore

Operation Region

Operation regions are regions in some space that contain hardware registers for exclusive use by ACPI control methods.

Operation Regions define the overall base address and length of a hardware region, but they cannot be accessed directly by AML code.

The following example ASL code shows the use of OperationRegion combined with Field to describe IDE 0 and 1 controlled through general I/O space, using one FET.

```
OperationRegion (GIO, SystemIO, 0x125, 0x1)
Field (GIO, ByteAcc, NoLock, Preserve) {
IDEI, 1, // IDEISO_EN - isolation buffer
IDEP, 1, // IDE_PWR_EN - power
IDER, 1 // IDERST#_EN - reset#
}
```

Platform Runtime Mechanism (PRM) - Introduction

introduces the capability of transitioning certain usages that were hitherto executed out of SMM, to a code that executes with the OS/VMM context.

Such usages are those that don’t require SMM privileges and a sub-set of HW SMI Handlers that don’t require SMM privileges.

This eliminates many of the cons present when executing the same code within the SMM environment.

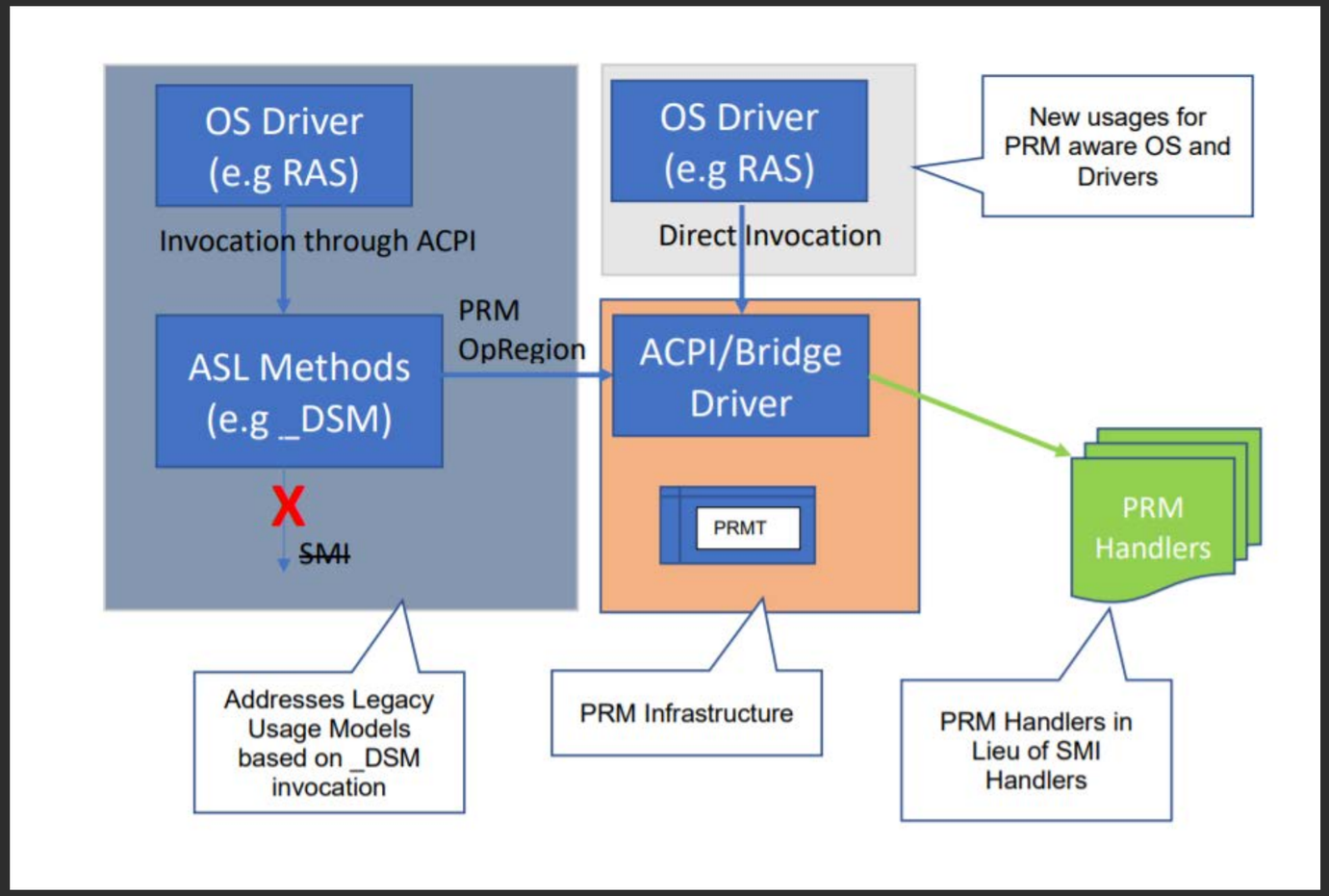
Platform Runtime Mechanism (PRM) - Continued

Platform Runtime Mechanism provides an ACPI Interpreter based infrastructure to invoke runtime platform firmware handlers. These runtime handlers are called PRM Handlers and are placed by the BIOS during boot (and updatable in OS runtime) in a runtime area reserved for firmware usage (such as UEFI Runtime area).

PRM handlers can be invoked by two means

1. Directly from an OS driver - if the OS driver and the OS ACPI subsystem is PRM aware.
2. 2. From ASL context – if the OS driver is not PRM aware and uses _DSM instead, or platform events that trigger SCI.

PRM Overview



During boot, the firmware discovers PRM modules included in the platform firmware than publishes the PRM ACPI table (PRMT) to describe the PRM modules, handlers, and related structures and allocates any required buffers

During OS runtime, OS code invokes PRM handlers via the direct call mechanism or with a _DSM.

The ACPI Interpreter based PRM infrastructure is the PRM OpRegion Handler in ACPI subsystem and can also logically be implemented as an independent\Bridge Driver in certain implementations.

http://uefi.org/sites/default/files/resources/Platform%20Runtime%20Mechanism%20-%20with%20legal%20notice.pdf

SMI use cases that can be converted to PRM

SW SMI triggers from an ACPI _DSM methods that invoke SW SMI so that complex algorithms and tasks can be handled in a native code execution context.

Examples include DSMs for RAS (such as address translation) and DSMs for supporting Non-Volatile DIMMs.

SMIs used for correctable error harvesting and reporting. By generating a SCI instead of SMI for these error conditions, ASL code can be invoked which can utilize PRM for error harvesting and reporting.

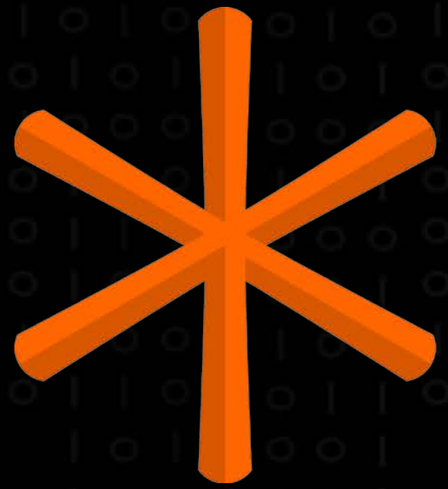
Resources

ACPI Specification	https://uefi.org/specifications
_DSD Implementation Guide	https://uefi.org/sites/default/files/resources/_DSD-implementation-guide-toplevel-1_2-3.htm
Other ACPI-related docs	https://uefi.org/acpi
ACPI Component Architecture	https://www.acpica.org
Platform Runtime Mechanism	https://uefi.org/sites/default/files/resources/Platform%20Runtime%20Mechanism%20-%20with%20legal%20notice.pdf


Questions?

Return to Main Training Page

Return to Training Table of contents for next presentation link



tianocore

ACKNOWLEDGEMENTS

example

```
DefinitionBlock ("DSDT.aml", "DSDT", 2, "OEM", "FOOBOOK", 0x1000)
{
...
If (LEqual (CFG1 (), 1))
{
...
Scope (_SB.PCI0.XHC.RHUB)
{
...
If (LEqual (CFG2 (), 1))
{
…
Device (HS11)
{
…
If (LEqual (CFG3 (), 1))
{
…
Device (CAM0)
{
…
}
…
}
…
}
```

ASL Example

```
// ASL Example
DefinitionBlock (
"forbook.aml", // Output Filename
"DSDT", // Signature
0x02, // DSDT Compliance Revision
"OEM", // OEMID
"forbook", // TABLE ID
0x1000 // OEM Revision
)
{ // start of definition block
OperationRegion(\GIO, SystemIO, 0x125, 0x1)
Field(\GIO, ByteAcc, NoLock, Preserve) {
CT01, 1,
}
```

```
Scope(\_SB) // start of scope
Device(PCI0) { // start of device
PowerResource(FET0, 0, 0) { // start of pwr
Method (_ON) {
Store (Ones, CT01) // assert power
Sleep (30) // wait 30ms
}
Method (_OFF) {
Store (Zero, CT01) // assert reset#
}
Method (_STA) {
Return (CT01)
}
} // end of power
} // end of device
} // end of scope
} // end of definition block
```

Example

An Operation Region object implicitly supports Mutex synchronization. Updates to the object, or a Field data object for the region, will automatically synchronize on the Operation Region object; however, a control method may also explicitly synchronize to a region to prevent other accesses to the region (from other control methods). Notice that according to the control method execution model, control method execution is non-preemptive. Because of this, explicit synchronization to an Operation Region needs to be done only in cases where a control method blocks or yields execution and where the type of register usage requires such synchronization.

The following example ASL code shows the use of OperationRegion combined with Field to describe IDE
0 and 1 controlled through general I/O space, using one FET.

```
OperationRegion (GIO, SystemIO, 0x125, 0x1)
Field (GIO, ByteAcc, NoLock, Preserve) {
IDEI, 1, // IDEISO_EN - isolation buffer
IDEP, 1, // IDE_PWR_EN - power
IDER, 1 // IDERST#_EN - reset#
}
```