



Platform configuration and Boot

Advanced Operating Systems

Vittorio Zaccaria | Politecnico di Milano | '25/26

Scan below QR code for course website



Seminar announcement

- Thursday, 27 November 2025 at 18:00. "**Secure boot and secure storage**" seminar from [Security Pattern](#)
- Seminar will be held online in my Webex room

Platform configuration » ACPI

Discoverability

Introduction

When booting, the OS must know:

- Which devices are already present on the machine
- How interrupts are managed
- How many processors

Peripheral devices standards such as PCI are not enough **to find and configure everything in a platform.**

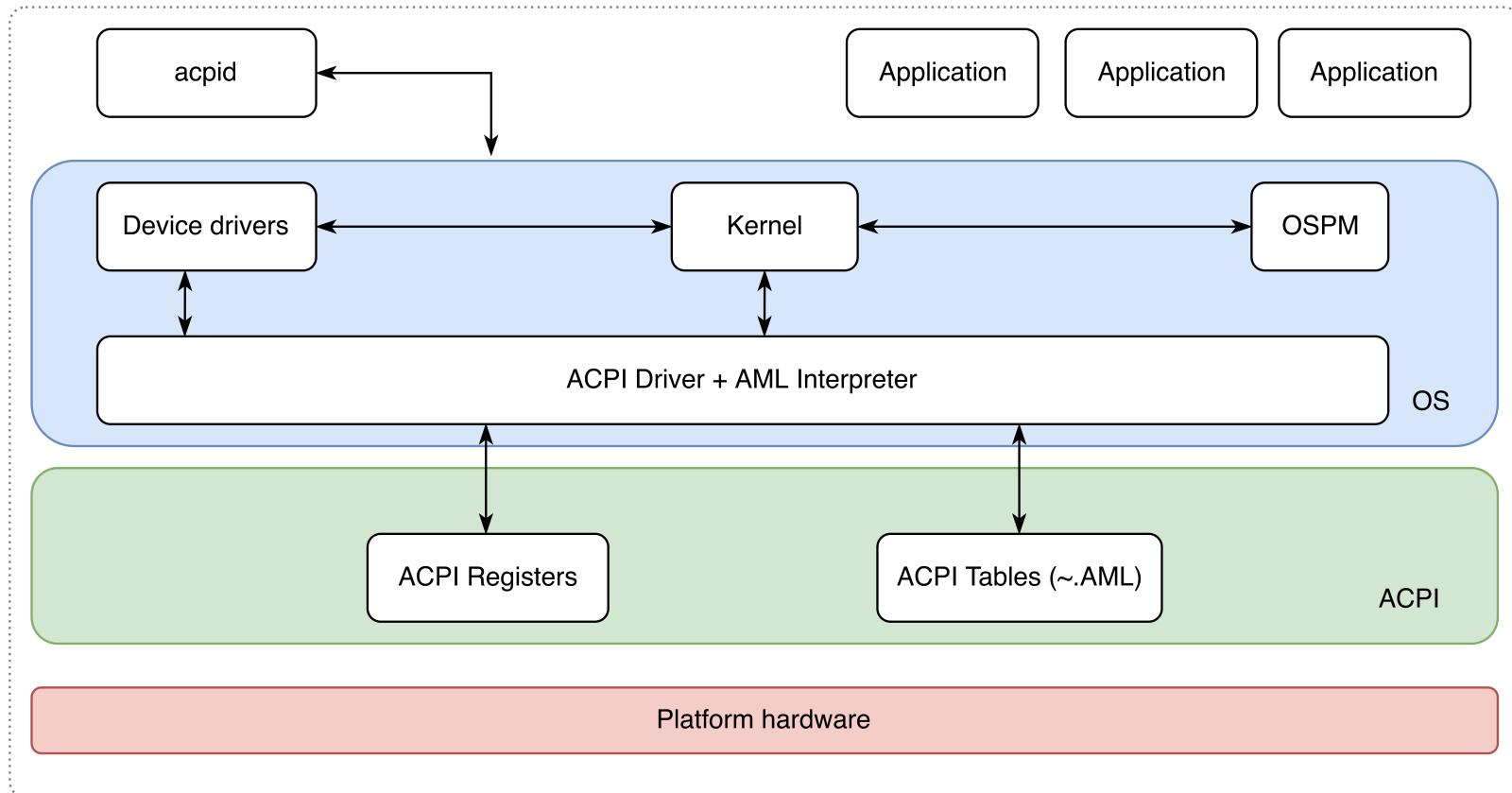
Two standards have evolved to provide the kernel with all the data of a platform (comprising PCI data)

- Advanced Configuration and Power Interface (**ACPI**). Used mainly on general purpose platforms (**intel**).
- Device trees. Used mainly on embedded platforms.

Introduction

- Developed by Intel, Microsoft and Toshiba in 1996
- Provides an open standard for operating systems to:
 - discover and configure computer hardware components,
 - perform power management e.g. putting unused hardware components to sleep,
 - perform auto configuration e.g. Plug and Play and hot swapping, and
 - perform status monitoring.
- No need to include platform-specific code for every platform.
- No need to ship a separate (binary) kernel for every platform

Representation



- Kernel receives a **platform description** in terms of tables which contain code and reference registers
- Code is executed with an interpreter on behalf of device drivers and kernel

Namespace

The `acpi` namespace is a hierarchical data structure describing the underlying hardware platform.

```

\
+---+ _PR           ; root
|   +--- CPU0       ; processor
|   |   +--- CPU0    ; first processor
...
...
+---+ _TZ           ; thermal control
|   ...
|   |   +--- PFAN     ; the fan
+---+ _SB           ; system bus
|   ...
|   |   +--- BAT0      ; the battery device
|   |   |   +--- _HID      ; the identifier of the battery
|   |   |   +--- _BST      ; the method to check for the status of the battery
|   |   |   +--- _BIF      ; the method to check for the status of the battery
...
+---+ PCI0          ; PCI Root bridge
|   ...
|   |   +--- GFX        ; The graphics adapter
|   |   |   +--- _ADR;    ; The PCI bus address
|   |   |   +--- LCD       ; The LCD output device
|   |   |   +--- _BCL      ; The method to control the backlight

```

Device example (Battery)

- A battery might appear in the `ACPI` device tree as `BAT0`.
- Catting these `/proc` files results in invoking `_BIF` and `_BST` methods of the `BAT0` object.

```
cat /proc/acpi/battery/BAT0/info
present: yes
design capacity: 5100 mAh
last full capacity: 5100 mAh
battery technology: rechargeable
design voltage: 11100 mV
design capacity warning: 420 mAh
design capacity low: 156 mAh
cycle count: 0
capacity granularity 1: 264 mAh
capacity granularity 2: 3780 mAh
model number: PA3465U
serial number: 3658Q
battery type: Li-Ion
OEM info: COMPAL
```

```
cat /proc/acpi/battery/BAT0/state
present: yes
capacity state: ok
charging state: charged
present rate: 0 mA
remaining capacity: 5100 mAh
present voltage: 11100 mV
```

Device example (Fan)

Here's the actual AML code that the kernel must interpret to control the fan:

```
Scope(\_TZ){  
    ...  
    OperationRegion(FANR, SystemIO, 0x8000, 0x10)  
    Field(FANR ...){ FCTL, 8 }  
    PowerSource(PFAN, 0, 0) {  
        Method(_ON) { Store(0x4, FCTL) }  
        Method(_OFF) { Store(0x0, FCTL) } } }
```

- `OperationRegion(name, space, offset, length)` defines the memory mapped region associated with this device
- `Field` defines a register in the region with a specific field. Here we have a single field named `FCTL` of 8 bits
- To turn on and off the fan, the methods write in this register

Power and thermal management

Background

Power management definition: Tools and techniques to consume **as little power as needed**, given system state, configuration and use case.

Goals:

- Improvement in reliability (**temperature**). Heat (a measure of the transfer of thermal energy between components) is proportional to P and the operational time. Higher temperatures reduce the reliability of components.
- Improvement in battery lifetime (**energy consumption**). Energy is the integration of power over time:

$$E = \int_t P(t)dt$$

- Ensure regulatory compliance (e.g., mandated cap on power consumption for a device to get a certain label[†]).

† See for example [EU requirements on mobile phones starting from 2025](#)

Let us focus on reliability and how power management can improve it.

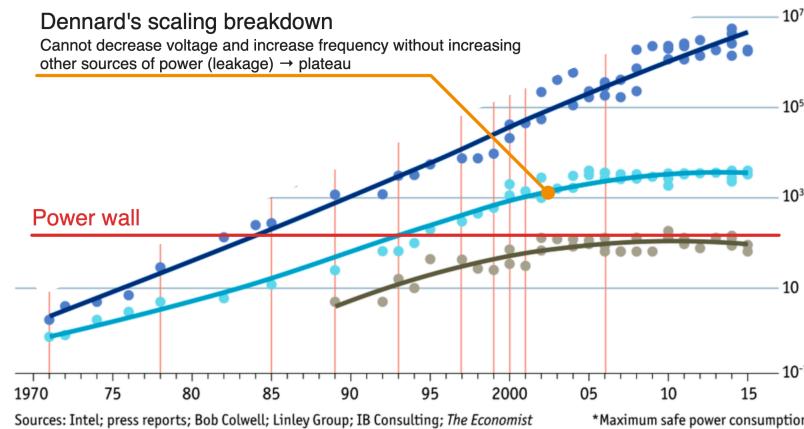
Reliability » Thermal design power

Thermal design power is the (average[†]) value of physical power that a cooling system must be able to dissipate to ensure **sustained reliability**.

System	TDP (W)	Cooling approach
Low-power embedded	< 5	Passive
Smartphone	4 - 5	Passive
Netbooks/Tablets	4 - 12	Mostly passive and possibly a fan
Low -power notebooks (e.g. Chromebook)	10 - 15	Passive and possibly a fan
Notebooks/Laptops	45 - 60	Heat-sink and a fan
Desktops	90 - 130	Heat-sink, multiple fans and (optionally) cooling tubes
Small servers	80 - 165	Heat-sink, multiple fans and (optionally) cooling tubes
Large servers and supercomputers	~ 300	Complex solutions in controlled rooms

[†] Actual power can temporarily go higher (for a short time)

Reliability » Thermal design power



Transistors

Increasing transistors exponentially with a plateaued TDP is facilitated by

- energy efficient computing (novel architectures, heterogeneous cores), and
- (drastic) power management techniques (DVFS, dark silicon).

Frequency

After the end of Dennard scaling, higher frequencies mean a lot more TDP so frequency has reached a plateau as well.

TDP

TDP increase has seemingly reached a plateau (power wall) because higher TDPs would require prohibitive cooling costs and put the product out of market.

The power management problem

Consideration

Today, the main source of consumption is the frequency and voltage of each system device

Problem:

- What voltage to use for each device (or even turn it off)?
- What frequency use for each device?
- What is the impact of the switching overhead on the system (how much time to turn it back on)?

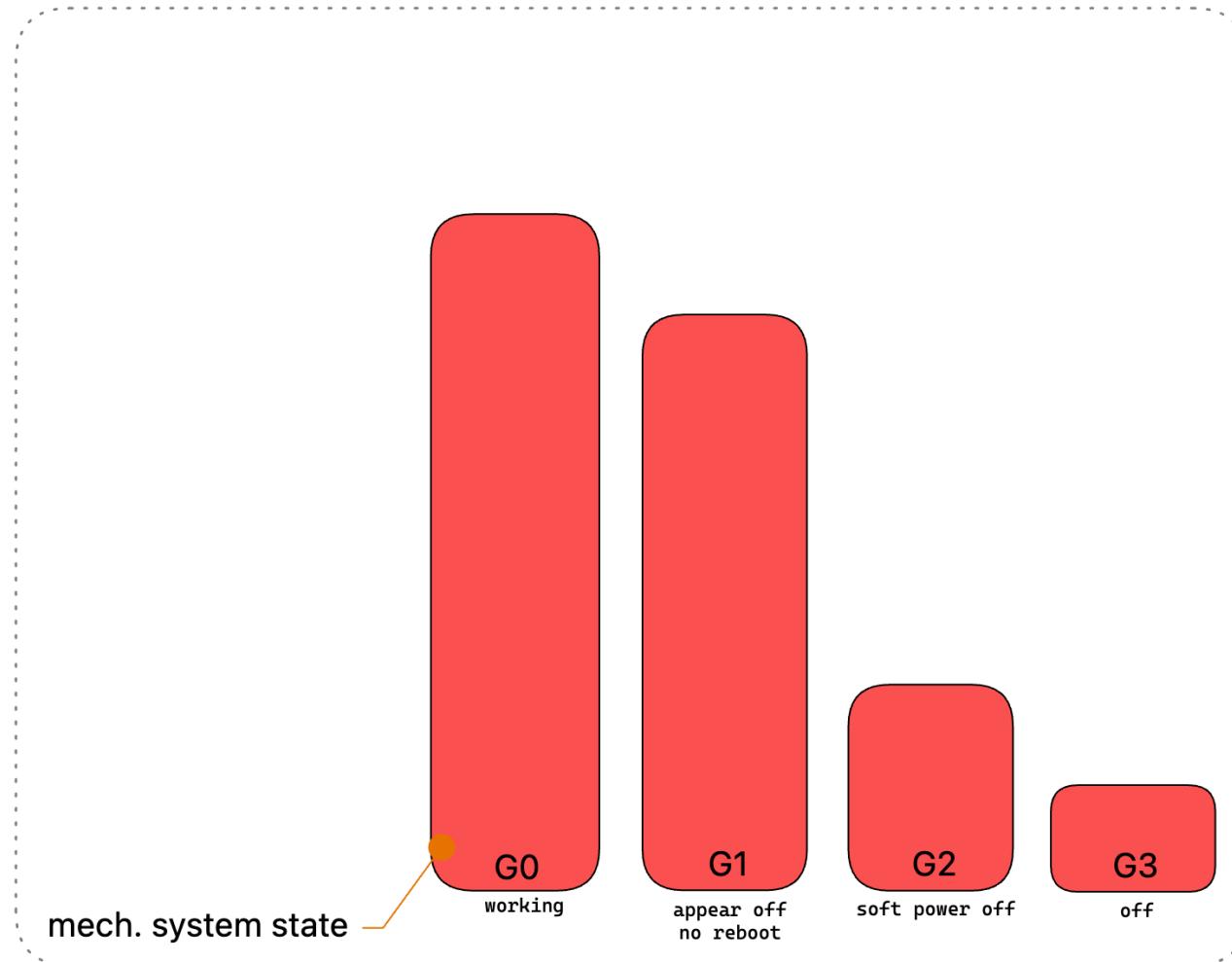
This is a difficult problem to address mathematically in an industrial setting.

ACPI Power states

ACPI took a pragmatic approach and simplified it through the use of providing:

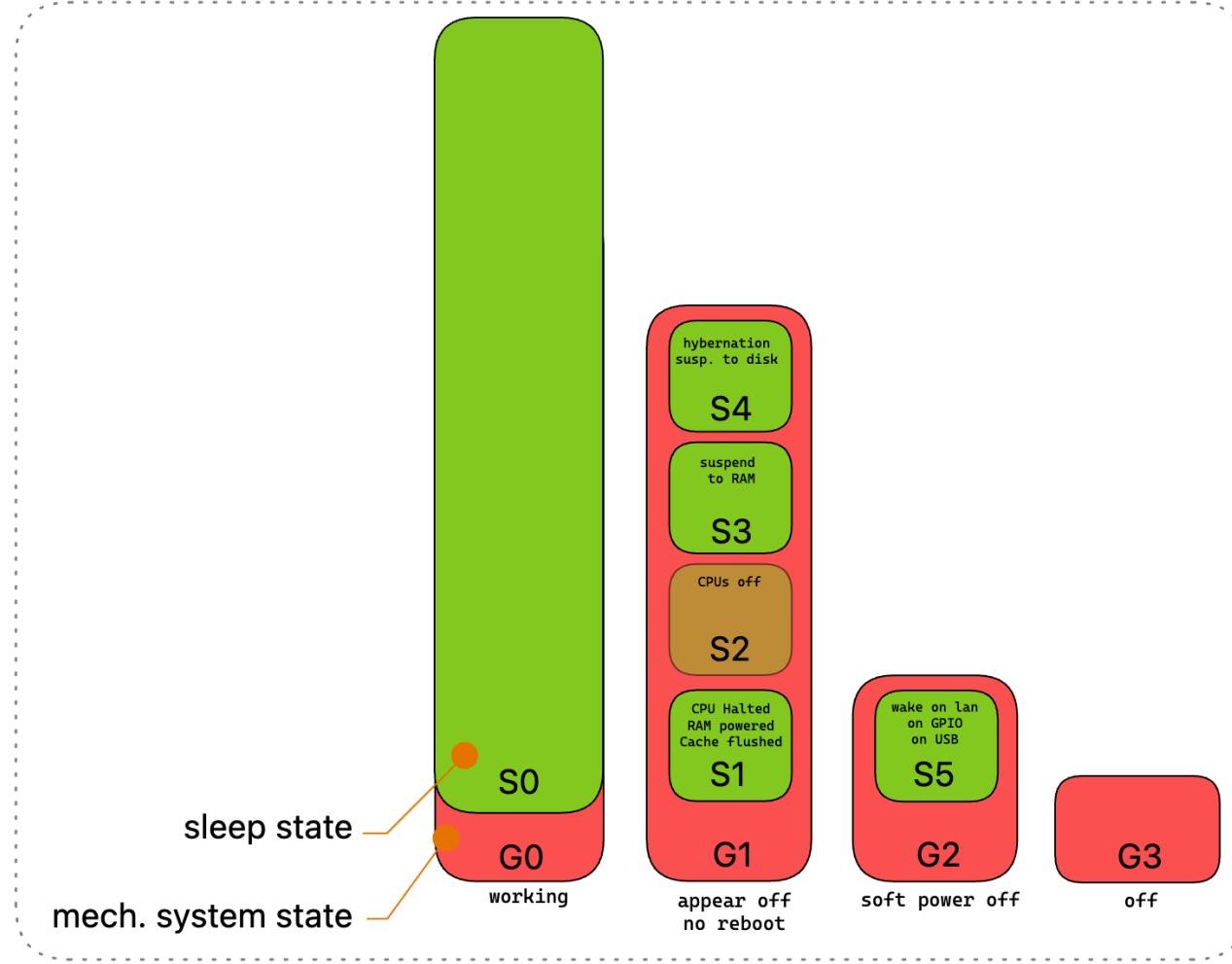
- Mechanisms for putting the computer as a whole in and out of system sleeping states.
- Tables that describe attached devices and their power states, the power planes the devices are connected to, and **controls for putting devices into different power states**.
- The OS must reason on a simplified state machine. The policy is left to the OSPM

ACPI Power states » Global states

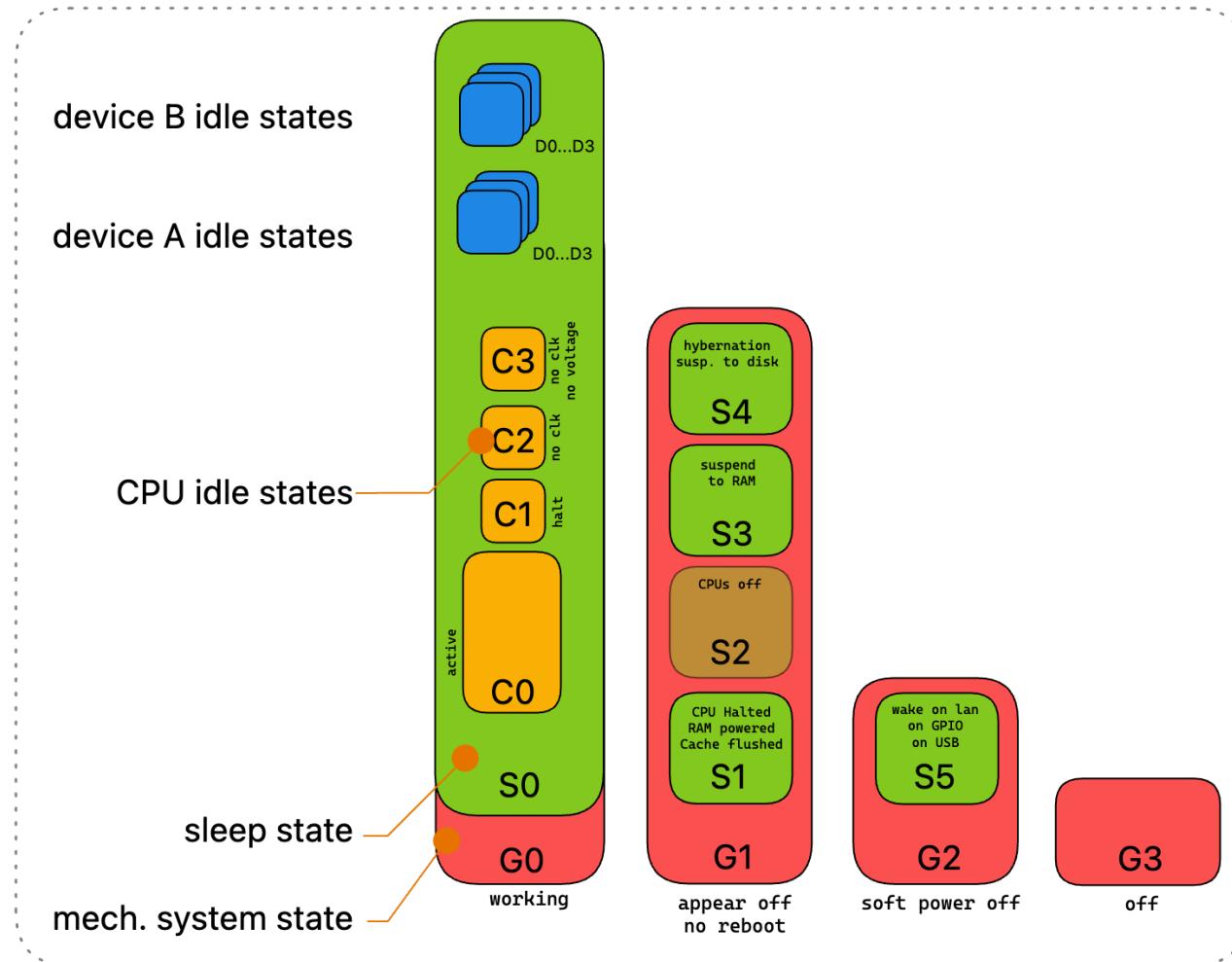




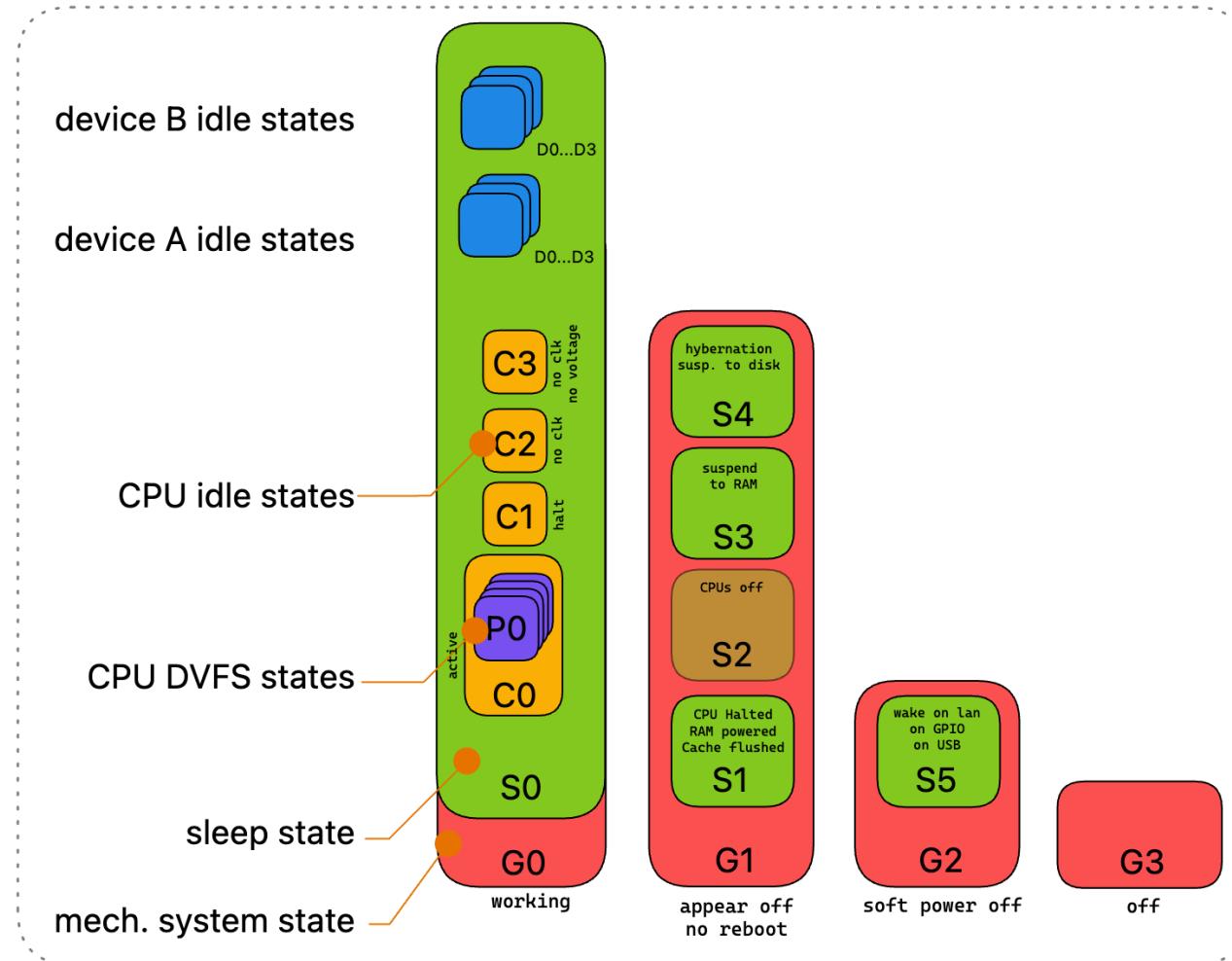
ACPI Power states » Sleep states



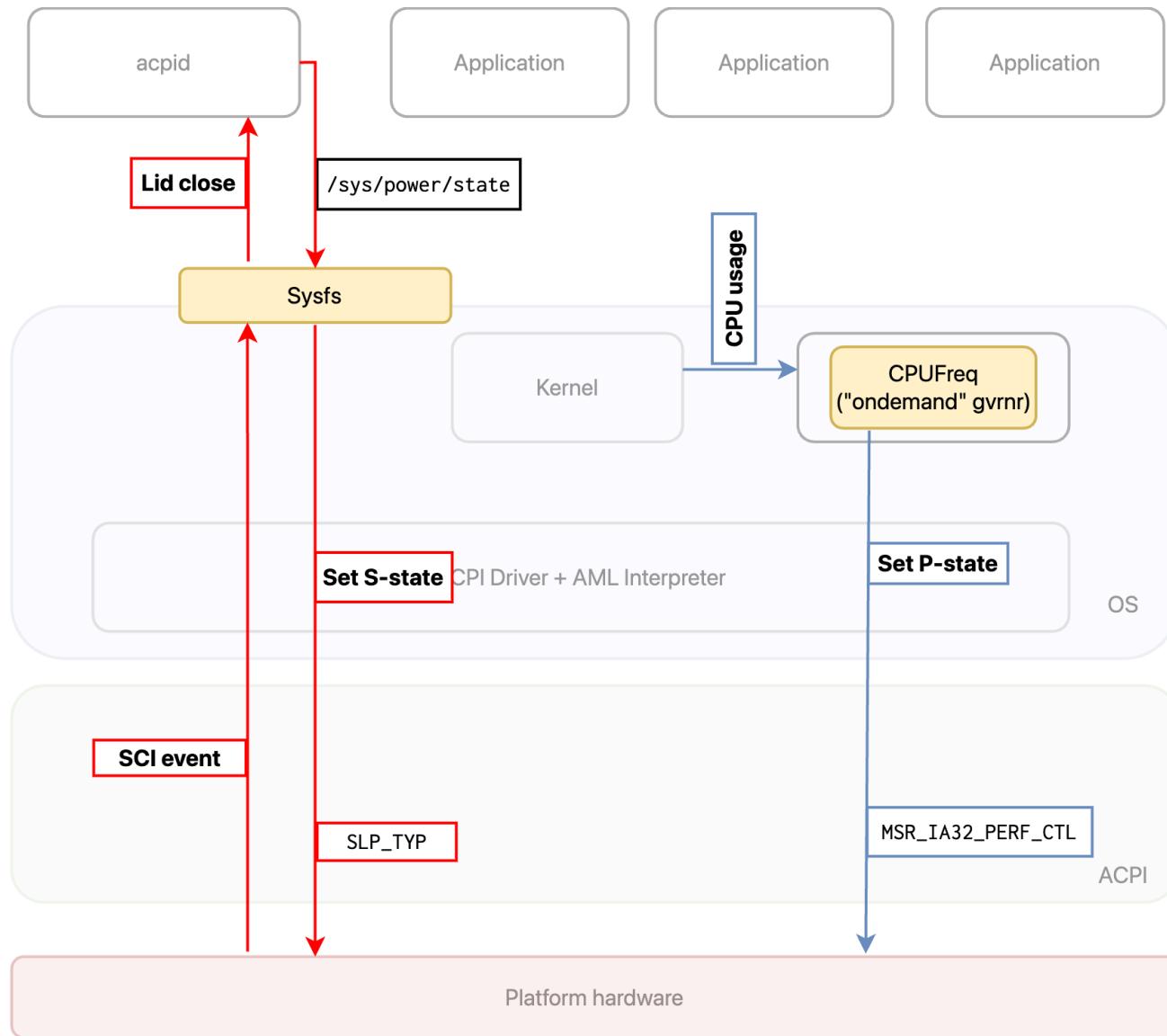
ACPI Power states » CPU states & device states



ACPI Power states » DVFS states



The Linux ACPI power management stack



Platform configuration » Device trees

Discoverability

Introduction

What it is

- **Data structure** describing hardware
- Device trees have both a binary format for operating systems to use and a textual format for convenient editing and management.

What is it for

- Usually passed to OS (e.g., Linux) to **provide information about HW topology where it cannot be detected/probed**
- Move the hardware description out of the kernel binary
 - no more hard-coded initialization functions
 - a single kernel can run on more than one board

Syntax

- Defined by Power.org Standard for Embedded Power Architecture Platform Requirements
- Nodes are organized in a hierarchy as a collection of property and value tokens.
- Typically compiled from .DTS (source) to .DTB (binary) and sent to the kernel on boot

```
/dts-v1/;  
/ {  
    compatible =  
        "acme,coyotes-revenge";  
    cpus {  
        cpu@0 {  
            compatible = "arm,cortex-a9";  
        };  
        ...  
    };  
    ...  
    serial@101F0000 {  
        compatible = "arm,pl011";  
    };  
    ...  
    flash@2,0 {  
        compatible = "samsung,k8f1315ebm",  
                    "cfi-flash";  
    };  
};
```

Syntax » Interrupt controller

- Interrupt signals are expressed as links between nodes independent of the tree.
- `#interrupt-cells:` how many cells are in an interrupt specifier for this interrupt controller

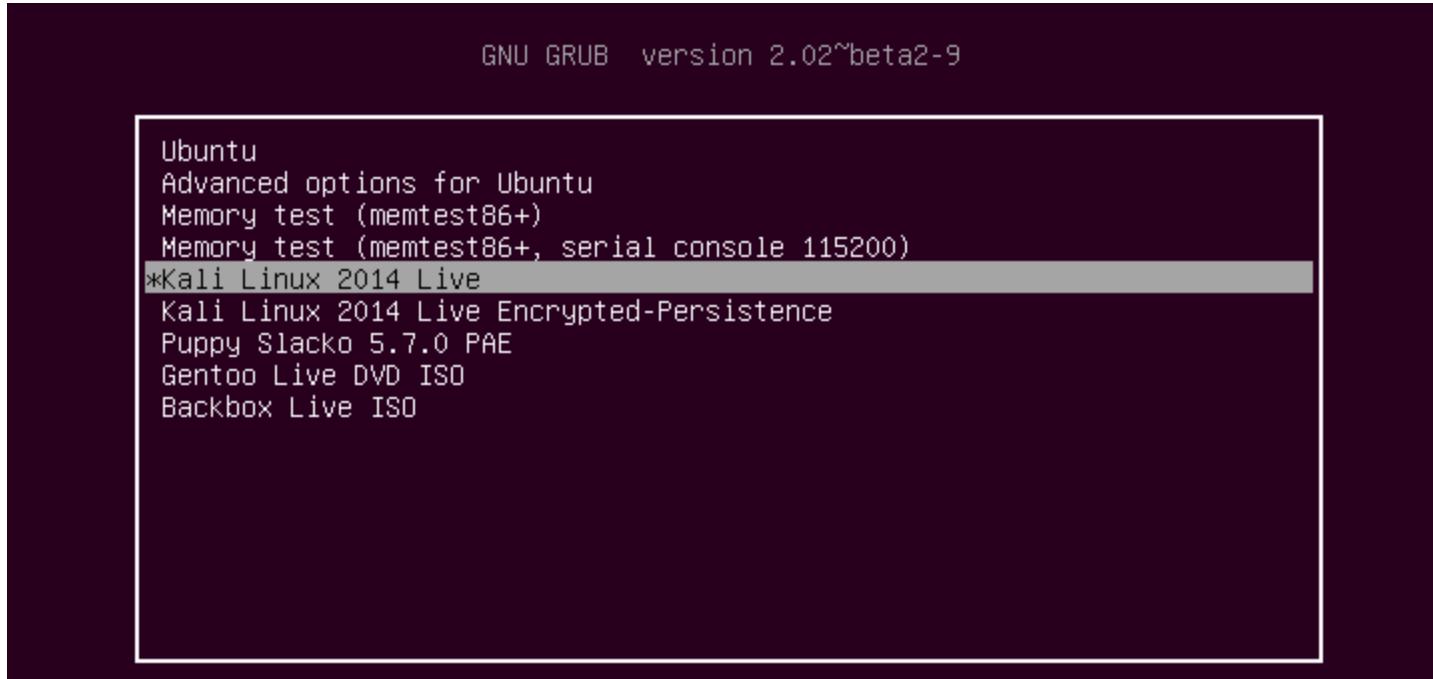
```
/ {
    #address-cells = <1>;
    #size-cells = <1>;
    interrupt-parent = <&intc>; // receiving interrupts of all the childs
    ...
    intc: interrupt-controller@10140000 {
        compatible = "arm,pl190";
        reg = <0x10140000 0x1000 >;
        interrupt-controller; // specifies this is the I.C.
        #interrupt-cells = <2>;
    };

    serial@101f0000 {
        compatible = "arm,pl011";
        reg = <0x101f0000 0x1000 >;
        interrupts = < 1 0 >; // (intno, flags, e.g., active high or low)
    };
    ...
}
```

Booting

Introduction

Booting



- What happens when we start up the PC?
- How does magically happens that the OS is loaded and started up?

Booting techniques

- General purpose PCs
 - BIOS (old)
 - UEFI (new)
- Embedded
 - UBOOT, still for Linux based platforms
 - other for bare metal OSes (see Terraneo's lecture)

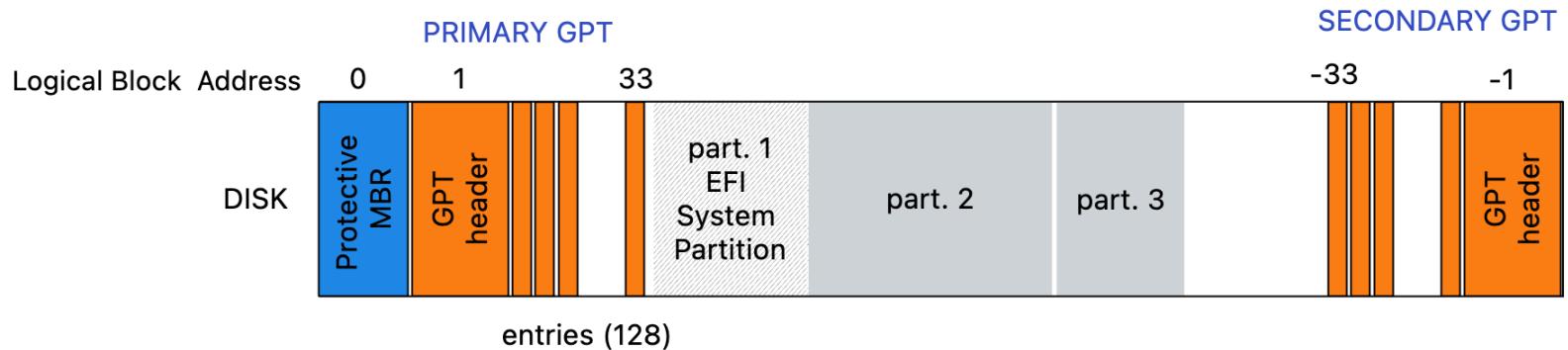
Booting a general purpose Linux

UEFI

- Replacement of **BIOS**
- Modular (you can extend it with drivers)
- Runs on various platforms and written in C
- Takes control right after the system is powered on and loads the firmware settings into RAM from **nvRAM**
- Startup files:
 - Stored in a **dedicated FAT32 partition**
 - Use standard pathnames, e.g., Linux: **elilo.efi**, OS X: **boot.efi**

GUID partition table

- No need for MBR code (ignore block 0)
- Uses GUID Partition Table (GPT)
 - Describes layout of the partition table on a disk
 - Occupies blocks 1-33
- UEFI understands Microsoft FAT file systems
 - Apple's UEFI knows HFS+ in addition



GUID partition table » Example disk structure

