Linux Firmware-guide Documentation

Release 6.8.0

The kernel development community

Jan 16, 2026

CONTENTS

This section describes the ACPI subsystem in Linux from firmware perspective.

CHAPTER

ONE

ACPI SUPPORT

1.1 ACPI Device Tree - Representation of ACPI Namespace

Author
Lv Zheng <lv.zheng@intel.com>

Credit
Thanks for the help from Zhang Rui <rui.zhang@intel.com> and Rafael J.Wysocki <rafael.j.wysocki@intel.com>.

1.1.1 Abstract

The Linux ACPI subsystem converts ACPI namespace objects into a Linux device tree under the /sys/devices/LNXSYSTEM:00 and updates it upon receiving ACPI hotplug notification events. For each device object in this hierarchy there is a corresponding symbolic link in the /sys/bus/acpi/devices.

This document illustrates the structure of the ACPI device tree.

1.1.2 ACPI Definition Blocks

The ACPI firmware sets up RSDP (Root System Description Pointer) in the system memory address space pointing to the XSDT (Extended System Description Table). The XSDT always points to the FADT (Fixed ACPI Description Table) using its first entry, the data within the FADT includes various fixed-length entries that describe fixed ACPI features of the hardware. The FADT contains a pointer to the DSDT (Differentiated System Description Table). The XSDT also contains entries pointing to possibly multiple SSDTs (Secondary System Description Table).

The DSDT and SSDT data is organized in data structures called definition blocks that contain definitions of various objects, including ACPI control methods, encoded in AML (ACPI Machine Language). The data block of the DSDT along with the contents of SSDTs represents a hierarchical data structure called the ACPI namespace whose topology reflects the structure of the underlying hardware platform.

The relationships between ACPI System Definition Tables described above are illustrated in the following diagram:

```
+---------+    +-------+    +--------+    +------------------------+
|  RSDP   | +->| XSDT  | +->|  FADT  |    |  +-------------------+ |
+---------+ |  +-------+ |  +--------+  +-|->|       DSDT        | |
| Pointer | |  | Entry |-+  | ...... |  | |  +-------------------+ |
+---------+ |  +-------+    | X_DSDT |--+ |  | Definition Blocks | |
| Pointer |-+  | ..... |    | ...... |    |  +-------------------+ |
+---------+    +-------+    +--------+    |  +-------------------+ |
               | Entry |------------------|->|       SSDT        | |
               +- - - -+                  |  +-------------------| |
               | Entry | - - - - - - - -+ |  | Definition Blocks | |
               +- - - -+                | |  +-------------------+ |
                                        | |  +- - - - - - - - - -+ |
                                        +-|->|       SSDT        | |
                                          |  +-------------------+ |
                                          |  | Definition Blocks | |
                                          |  +- - - - - - - - - -+ |
                                          +------------------------+
                                                       |
                                          OSPM Loading |
                                                      \|/
                                                +----------------+
                                                | ACPI Namespace |
                                                +----------------+

               Figure 1. ACPI Definition Blocks
```

Note: RSDP can also contain a pointer to the RSDT (Root System Description Table). Platforms provide RSDT to enable compatibility with ACPI 1.0 operating systems. The OS is expected to use XSDT, if present.

1.1.3 Example ACPI Namespace

All definition blocks are loaded into a single namespace. The namespace is a hierarchy of objects identified by names and paths. The following naming conventions apply to object names in the ACPI namespace:

1. All names are 32 bits long.
2. The first byte of a name must be one of 'A' - 'Z', '_'.
3. Each of the remaining bytes of a name must be one of 'A' - 'Z', '0' - '9', '_'.
4. Names starting with '_' are reserved by the ACPI specification.
5. The '' symbol represents the root of the namespace (i.e. names prepended with '' are relative to the namespace root).
6. The '^' symbol represents the parent of the current namespace node (i.e. names prepended with '^' are relative to the parent of the current namespace node).

The figure below shows an example ACPI namespace:

```
+------+
| \    |                     Root
+------+
  |
  | +------+
  +-| _PR  |                 Scope(_PR): the processor namespace
  | +------+
  |   |
  |   | +------+
  |   +-| CPU0 |             Processor(CPU0): the first processor
  |     +------+
  |
  | +------+
  +-| _SB  |                 Scope(_SB): the system bus namespace
  | +------+
  |   |
  |   | +------+
  |   +-| LID0 |             Device(LID0); the lid device
  |   | +------+
  |   |   |
  |   |   | +------+
  |   |   +-| _HID |         Name(_HID, "PNP0C0D"): the hardware ID
  |   |   | +------+
  |   |   |
  |   |   | +------+
  |   |   +-| _STA |         Method(_STA): the status control method
  |   |     +------+
  |   |
  |   | +------+
  |   +-| PCI0 |             Device(PCI0); the PCI root bridge
  |     +------+
  |       |
  |       | +------+
  |       +-| _HID |         Name(_HID, "PNP0A08"): the hardware ID
  |       | +------+
  |       |
  |       | +------+
  |       +-| _CID |         Name(_CID, "PNP0A03"): the compatible ID
  |       | +------+
  |       |
  |       | +------+
  |       +-| RP03 |         Scope(RP03): the PCI0 power scope
  |       | +------+
  |       |   |
  |       |   | +------+
  |       |   +-| PXP3 |     PowerResource(PXP3): the PCI0 power resource
  |       |     +------+
  |       |
  |       | +------+
  |       +-| GFX0 |         Device(GFX0): the graphics adapter
  |         +------+
  |           |
  |           | +------+
  |           +-| _ADR |     Name(_ADR, 0x00020000): the PCI bus address
  |           | +------+
  |           |
  |           | +------+
  |           +-| DD01 |     Device(DD01): the LCD output device
  |             +------+
  |               |
  |               | +------+
  |               +-| _BCL | Method(_BCL): the backlight control method
  |                 +------+
  |
  | +------+
  +-| _TZ  |                 Scope(_TZ): the thermal zone namespace
  | +------+
  |   |
  |   | +------+
  |   +-| FN00 |             PowerResource(FN00): the FAN0 power resource
  |   | +------+
  |   |
  |   | +------+
  |   +-| FAN0 |             Device(FAN0): the FAN0 cooling device
  |   | +------+
  |   |   |
  |   |   | +------+
  |   |   +-| _HID |         Name(_HID, "PNP0A0B"): the hardware ID
  |   |     +------+
  |   |
  |   | +------+
  |   +-| TZ00 |             ThermalZone(TZ00); the FAN thermal zone
  |     +------+
  |
  | +------+
  +-| _GPE |                 Scope(_GPE): the GPE namespace
    +------+

                  Figure 2. Example ACPI Namespace
```

1.1.4 Linux ACPI Device Objects

The Linux kernel's core ACPI subsystem creates struct acpi_device objects for ACPI namespace objects representing devices, power resources processors, thermal zones. Those objects are exported to user space via sysfs as directories in the subtree under /sys/devices/LNXSYSTM:00. The format of their names is <bus_id:instance>, where 'bus_id' refers to the ACPI namespace representation of the given object and 'instance' is used for distinguishing different object of the same 'bus_id' (it is two-digit decimal representation of an unsigned integer).

The value of 'bus_id' depends on the type of the object whose name it is part of as listed in the table below:

```
+---+----------------+-------+----------+
|   | Object/Feature | Table | bus_id   |
+---+----------------+-------+----------+
| N | Root           | xSDT  | LNXSYSTM |
+---+----------------+-------+----------+
| N | Device         | xSDT  | _HID     |
+---+----------------+-------+----------+
| N | Processor      | xSDT  | LNXCPU   |
+---+----------------+-------+----------+
| N | ThermalZone    | xSDT  | LNXTHERM |
+---+----------------+-------+----------+
| N | PowerResource  | xSDT  | LNXPOWER |
+---+----------------+-------+----------+
| N | Other Devices  | xSDT  | device   |
+---+----------------+-------+----------+
| F | PWR_BUTTON     | FADT  | LNXPWRBN |
+---+----------------+-------+----------+
| F | SLP_BUTTON     | FADT  | LNXSLPBN |
+---+----------------+-------+----------+
| M | Video Extension | xSDT  | LNXVIDEO |
+---+----------------+-------+----------+
| M | ATA Controller  | xSDT  | LNXIOBAY |
+---+----------------+-------+----------+
| M | Docking Station | xSDT  | LNXDOCK  |
+---+----------------+-------+----------+

 Table 1. ACPI Namespace Objects Mapping
```

The following rules apply when creating struct acpi_device objects on the basis of the contents of ACPI System Description Tables (as indicated by the letter in the first column and the notation in the second column of the table above):

N:
: The object's source is an ACPI namespace node (as indicated by the named object's type in the second column). In that case the object's directory in sysfs will contain the 'path' attribute whose value is the full path to the node from the namespace root.

F:
: The struct acpi_device object is created for a fixed hardware feature (as indicated by the fixed feature flag's name in the second column), so its sysfs directory will not contain the 'path' attribute.

M:
: The struct acpi_device object is created for an ACPI namespace node with specific control methods (as indicated by the ACPI defined device's type in the second column). The 'path' attribute containing its namespace path will be present in its sysfs directory. For example, if the _BCL method is present for an ACPI namespace node, a struct acpi_device object with LNXVIDEO 'bus_id' will be created for it.

The third column of the above table indicates which ACPI System Description Tables contain information used for the creation of the struct acpi_device objects represented by the given row (xSDT means DSDT or SSDT).

The fourth column of the above table indicates the 'bus_id' generation rule of the struct acpi_device object:

_HID:
_HID in the last column of the table means that the object's bus_id is derived from the _HID/_CID identification objects present under the corresponding ACPI namespace node. The object's sysfs directory will then contain the 'hid' and 'modalias' attributes that can be used to retrieve the _HID and _CIDs of that object.

LNXxxxxx:
The 'modalias' attribute is also present for struct acpi_device objects having bus_id of the "LNXxxxxx" form (pseudo devices), in which cases it contains the bus_id string itself.

device:
'device' in the last column of the table indicates that the object's bus_id cannot be determined from _HID/_CID of the corresponding ACPI namespace node, although that object represents a device (for example, it may be a PCI device with _ADR defined and without _HID or _CID). In that case the string 'device' will be used as the object's bus_id.

1.1.5 Linux ACPI Physical Device Glue

ACPI device (i.e. struct acpi_device) objects may be linked to other objects in the Linux' device hierarchy that represent "physical" devices (for example, devices on the PCI bus). If that happens, it means that the ACPI device object is a "companion" of a device otherwise represented in a different way and is used (1) to provide configuration information on that device which cannot be obtained by other means and (2) to do specific things to the device with the help of its ACPI control methods. One ACPI device object may be linked this way to multiple "physical" devices.

If an ACPI device object is linked to a "physical" device, its sysfs directory contains the "physical_node" symbolic link to the sysfs directory of the target device object. In turn, the target device's sysfs directory will then contain the "firmware_node" symbolic link to the sysfs directory of the companion ACPI device object. The linking mechanism relies on device identification provided by the ACPI namespace. For example, if there's an ACPI namespace object representing a PCI device (i.e. a device object under an ACPI namespace object representing a PCI bridge) whose _ADR returns 0x00020000 and the bus number of the parent PCI bridge is 0, the sysfs directory representing the struct acpi_device object created for that ACPI namespace object will contain the 'physical_node' symbolic link to the /sys/devices/pci0000:00/0000:00:02:0/ sysfs directory of the corresponding PCI device.

The linking mechanism is generally bus-specific. The core of its implementation is located in the drivers/acpi/glue.c file, but there are complementary parts depending on the bus types in question located elsewhere. For example, the PCI-specific part of it is located in drivers/pci/pci-acpi.c.

1.1.6 Example Linux ACPI Device Tree

The sysfs hierarchy of struct acpi_device objects corresponding to the example ACPI namespace illustrated in Figure 2 with the addition of fixed PWR_BUTTON/SLP_BUTTON devices is shown below:

```
+--------------+---+-----------------+
| LNXSYSTEM:00 | \ | acpi:LNXSYSTEM: |
+--------------+---+-----------------+
  |
  | +-------------+-----+----------------+
  +-| LNXPWRBN:00 | N/A | acpi:LNXPWRBN: |
  | +-------------+-----+----------------+
  |
  | +-------------+-----+----------------+
  +-| LNXSLPBN:00 | N/A | acpi:LNXSLPBN: |
  | +-------------+-----+----------------+
  |
  | +-----------+------------+--------------+
  +-| LNXCPU:00 | \_PR_.CPU0 | acpi:LNXCPU: |
  | +-----------+------------+--------------+
  |
  | +-------------+-------+----------------+
  +-| LNXSYBUS:00 | \_SB_ | acpi:LNXSYBUS: |
  | +-------------+-------+----------------+
  |   |
  |   | +- - - - - - - +- - - - - - +- - - - - - - -+
  |   +-| PNP0C0D:00 | \_SB_.LID0 | acpi:PNP0C0D: |
  |   | +- - - - - - - +- - - - - - +- - - - - - - -+
  |   |
  |   | +------------+------------+-----------------------+
  |   +-| PNP0A08:00 | \_SB_.PCI0 | acpi:PNP0A08:PNP0A03: |
  |     +------------+------------+-----------------------+
  |       |
  |       | +-----------+-----------------+-----+
  |       +-| device:00 | \_SB_.PCI0.RP03 | N/A |
  |       | +-----------+-----------------+-----+
  |       |   |
  |       |   | +-------------+----------------------+----------------+
  |       |   +-| LNXPOWER:00 | \_SB_.PCI0.RP03.PXP3 | acpi:LNXPOWER: |
  |       |     +-------------+----------------------+----------------+
  |       |
  |       | +-------------+-----------------+----------------+
  |       +-| LNXVIDEO:00 | \_SB_.PCI0.GFX0 | acpi:LNXVIDEO: |
  |         +-------------+-----------------+----------------+
  |           |
  |           | +-----------+-----------------+-----+
  |           +-| device:01 | \_SB_.PCI0.DD01 | N/A |
  |             +-----------+-----------------+-----+
  |
  | +-------------+-------+----------------+
  +-| LNXSYBUS:01 | \_TZ_ | acpi:LNXSYBUS: |
    +-------------+-------+----------------+
      |
      | +-------------+------------+----------------+
      +-| LNXPOWER:0a | \_TZ_.FN00 | acpi:LNXPOWER: |
      | +-------------+------------+----------------+
      |
      | +------------+------------+---------------+
      +-| PNP0C0B:00 | \_TZ_.FAN0 | acpi:PNP0C0B: |
      | +------------+------------+---------------+
      |
      | +-------------+------------+----------------+
      +-| LNXTHERM:00 | \_TZ_.TZ00 | acpi:LNXTHERM: |
        +-------------+------------+----------------+

               Figure 3. Example Linux ACPI Device Tree
```

Note: Each node is represented as ”object/path/modalias”, where:

1. 'object' is the name of the object's directory in sysfs.
2. 'path' is the ACPI namespace path of the corresponding ACPI namespace object, as returned by the object's 'path' sysfs attribute.
3. 'modalias' is the value of the object's 'modalias' sysfs attribute (as described earlier in this document).

Note: N/A indicates the device object does not have the 'path' or the 'modalias' attribute.

1.2 Graphs

1.2.1 _DSD

_DSD (Device Specific Data) [dsd-guide] is a predefined ACPI device configuration object that can be used to convey information on hardware features which are not specifically covered by the ACPI specification [acpi]. There are two _DSD extensions that are relevant for graphs: property [dsd-guide] and hierarchical data extensions. The property extension provides generic key-value pairs whereas the hierarchical data extension supports nodes with references to other nodes, forming a tree. The nodes in the tree may contain properties as defined by the property extension. The two extensions together provide a tree-like structure with zero or more properties (key-value pairs) in each node of the tree.

The data structure may be accessed at runtime by using the device_* and fwnode_* functions defined in include/linux/fwnode.h .

Fwnode represents a generic firmware node object. It is independent on the firmware type. In ACPI, fwnodes are _DSD hierarchical data extensions objects. A device's _DSD object is represented by an fwnode.

The data structure may be referenced to elsewhere in the ACPI tables by using a hard reference to the device itself and an index to the hierarchical data extension array on each depth.

1.2.2 Ports and endpoints

The port and endpoint concepts are very similar to those in Devicetree [devicetree, graph-bindings]. A port represents an interface in a device, and an endpoint represents a connection to that interface. Also see [data-node-ref] for generic data node references.

All port nodes are located under the device's ”_DSD” node in the hierarchical data extension tree. The data extension related to each port node must begin with ”port” and must be followed by the ”@” character and the number of the port as its key. The target object it refers to should be called ”PRTX”, where ”X” is the number of the port. An example of such a package would be:

```
Package() { "port@4", "PRT4" }
```

Further on, endpoints are located under the port nodes. The hierarchical data extension key of the endpoint nodes must begin with ”endpoint” and must be followed by the ”@” character and the number of the endpoint. The object it refers to should be called ”EPXY”, where ”X” is the number of the port and ”Y” is the number of the endpoint. An example of such a package would be:

```
Package() { "endpoint@0", "EP40" }
```

Each port node contains a property extension key ”port”, the value of which is the number of the port. Each endpoint is similarly numbered with a property extension key ”reg”, the value of which is the number of the endpoint. Port numbers must be unique within a device and endpoint numbers must be unique within a port. If a device object may only has a single port, then the number of that port shall be zero. Similarly, if a port may only have a single endpoint, the number of that endpoint shall be zero.

The endpoint reference uses property extension with ”remote-endpoint” property name followed by a reference in the same package. Such references consist of the remote device reference, the first package entry of the port data extension reference under the device and finally the first package entry of the endpoint data extension reference under the port. Individual references thus appear as:

```
Package() { device, "port@X", "endpoint@Y" }
```

In the above example, ”X” is the number of the port and ”Y” is the number of the endpoint.

The references to endpoints must be always done both ways, to the remote endpoint and back from the referred remote endpoint node.

A simple example of this is show below:

```
Scope (\_SB.PCI0.I2C2)
{
    Device (CAM0)
    {
        Name (_DSD, Package () {
            ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
            Package () {
                Package () { "compatible", Package () { "nokia,smia" } },
            },
            ToUUID("dbb8e3e6-5886-4ba6-8795-1319f52a966b"),
            Package () {
                Package () { "port@0", "PRT0" },
            }
        })
        Name (PRT0, Package() {
            ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
            Package () {
                Package () { "reg", 0 },
            },
            ToUUID("dbb8e3e6-5886-4ba6-8795-1319f52a966b"),
            Package () {
                Package () { "endpoint@0", "EP00" },
            }
        })
        Name (EP00, Package() {
            ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
            Package () {
                Package () { "reg", 0 },
                Package () { "remote-endpoint", Package() { \_SB.PCI0.ISP,
↪"port@4", "endpoint@0" } },
            }
        })
    }
}

Scope (\_SB.PCI0)
{
    Device (ISP)
    {
        Name (_DSD, Package () {
            ToUUID("dbb8e3e6-5886-4ba6-8795-1319f52a966b"),
            Package () {
                Package () { "port@4", "PRT4" },
            }
        })

        Name (PRT4, Package() {
            ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
            Package () {
                Package () { "reg", 4 }, /* CSI-2 port number */
            },
            ToUUID("dbb8e3e6-5886-4ba6-8795-1319f52a966b"),
            Package () {
                Package () { "endpoint@0", "EP40" },
            }
        })
        Name (EP40, Package() {
            ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
            Package () {
                Package () { "reg", 0 },
                Package () { "remote-endpoint", Package () { \_SB.PCI0.I2C2.
↪CAM0, "port@0", "endpoint@0" } },
            }
        })
    }
}
```

Here, the port 0 of the ”CAM0” device is connected to the port 4 of the ”ISP” device and vice versa.

1.2.3 References

[acpi] Advanced Configuration and Power Interface Specification.
https://uefi.org/specifications/ACPI/6.4/, referenced 2021-11-30.

[data-node-ref] *Referencing hierarchical data nodes*

[devicetree] Devicetree. https://www.devicetree.org, referenced 2016-10-03.

[dsd-guide] DSD Guide.
https://github.com/UEFI/DSD-Guide/blob/main/dsd-guide.adoc, referenced 2021-11-30.

[dsd-rules] _DSD Device Properties Usage Rules.
_DSD Device Properties Usage Rules

[graph-bindings] Common bindings for device graphs (Devicetree).
https://github.com/devicetree-org/dt-schema/blob/main/schemas/graph.yaml, referenced 2021-11-30.

1.3 Referencing hierarchical data nodes

Author
Sakari Ailus <sakari.ailus@linux.intel.com>

ACPI in general allows referring to device objects in the tree only. Hierarchical data extension nodes may not be referred to directly, hence this document defines a scheme to implement such references.

A reference consist of the device object name followed by one or more hierarchical data extension [dsd-guide] keys. Specifically, the hierarchical data extension node which is referred to by the key shall lie directly under the parent object i.e. either the device object or another hierarchical data extension node.

The keys in the hierarchical data nodes shall consist of the name of the node, ”@” character and the number of the node in hexadecimal notation (without pre- or postfixes). The same ACPI object shall include the _DSD property extension with a property ”reg” that shall have the same numerical value as the number of the node.

In case a hierarchical data extensions node has no numerical value, then the ”reg” property shall be omitted from the ACPI object's _DSD properties and the ”@” character and the number shall be omitted from the hierarchical data extension key.

1.3.1 Example

In the ASL snippet below, the ”reference” _DSD property contains a device object reference to DEV0 and under that device object, a hierarchical data extension key ”node@1” referring to the NOD1 object and lastly, a hierarchical data extension key ”anothernode” referring to the ANOD object which is also the final target node of the reference.

```
Device (DEV0)
{
    Name (_DSD, Package () {
        ToUUID("dbb8e3e6-5886-4ba6-8795-1319f52a966b"),
        Package () {
            Package () { "node@0", "NOD0" },
            Package () { "node@1", "NOD1" },
        }
    })
    Name (NOD0, Package() {
        ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
        Package () {
            Package () { "reg", 0 },
            Package () { "random-property", 3 },
        }
    })
    Name (NOD1, Package() {
        ToUUID("dbb8e3e6-5886-4ba6-8795-1319f52a966b"),
        Package () {
            Package () { "reg", 1 },
            Package () { "anothernode", "ANOD" },
        }
    })
    Name (ANOD, Package() {
        ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
        Package () {
            Package () { "random-property", 0 },
        }
    })
}

Device (DEV1)
{
    Name (_DSD, Package () {
        ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
        Package () {
            Package () {
                "reference", Package () {
                    ^DEV0, "node@1", "anothernode"
                }
            },
        }
    })
}
```

Please also see a graph example in *Graphs*.

1.3.2 References

[dsd-guide] DSD Guide.
https://github.com/UEFI/DSD-Guide/blob/main/dsd-guide.adoc, referenced 2021-11-30.

1.4 Describing and referring to LEDs in ACPI

Individual LEDs are described by hierarchical data extension [5] nodes under the device node, the LED driver chip. The ”reg” property in the LED specific nodes tells the numerical ID of each individual LED output to which the LEDs are connected. [leds] The hierarchical data nodes are named ”led@X”, where X is the number of the LED output.

Referring to LEDs in Device tree is documented in [video-interfaces], in ”flash-leds” property documentation. In short, LEDs are directly referred to by using phandles.

While Device tree allows referring to any node in the tree [devicetree], in ACPI references are limited to device nodes only [acpi]. For this reason using the same mechanism on ACPI is not possible. A mechanism to refer to non-device ACPI nodes is documented in [data-node-ref].

ACPI allows (as does DT) using integer arguments after the reference. A combination of the LED driver device reference and an integer argument, referring to the ”reg” property of the relevant LED, is used to identify individual LEDs. The value of the ”reg” property is a contract between the firmware and software, it uniquely identifies the LED driver outputs.

Under the LED driver device, The first hierarchical data extension package list entry shall contain the string ”led@” followed by the number of the LED, followed by the referred object name. That object shall be named ”LED” followed by the number of the LED.

1.4.1 Example

An ASL example of a camera sensor device and a LED driver device for two LEDs is show below. Objects not relevant for LEDs or the references to them have been omitted.

```
Device (LED)
{
        Name (_DSD, Package () {
                ToUUID("dbb8e3e6-5886-4ba6-8795-1319f52a966b"),
                Package () {
                        Package () { "led@0", LED0 },
                        Package () { "led@1", LED1 },
                }
        })
        Name (LED0, Package () {
                ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
                Package () {
                        Package () { "reg", 0 },
                        Package () { "flash-max-microamp", 1000000 },
                        Package () { "flash-timeout-us", 200000 },
                        Package () { "led-max-microamp", 100000 },
                        Package () { "label", "white:flash" },
                }
        })
        Name (LED1, Package () {
                ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
                Package () {
                        Package () { "reg", 1 },
                        Package () { "led-max-microamp", 10000 },
                        Package () { "label", "red:indicator" },
                }
        })
}

Device (SEN)
{
        Name (_DSD, Package () {
                ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
                Package () {
                        Package () {
                                "flash-leds",
                                Package () { ^LED, "led@0", ^LED, "led@1" },
                        }
                }
        })
}
```

where

```
LED     LED driver device
LED0    First LED
LED1    Second LED
SEN     Camera sensor device (or another device the LED is related to)
```

1.4.2 References

[acpi] Advanced Configuration and Power Interface Specification.
https://uefi.org/specifications/ACPI/6.4/, referenced 2021-11-30.

[data-node-ref] *Referencing hierarchical data nodes*

[devicetree] Devicetree. https://www.devicetree.org, referenced 2019-02-21.

[dsd-guide] DSD Guide.
https://github.com/UEFI/DSD-Guide/blob/main/dsd-guide.adoc, referenced 2021-11-30.

[leds] Documentation/devicetree/bindings/leds/common.yaml

[video-interfaces] Documentation/devicetree/bindings/media/video-interfaces.yaml

1.5 MDIO bus and PHYs in ACPI

The PHYs on an MDIO bus [phy] are probed and registered using fwnode_mdiobus_register_phy().

Later, for connecting these PHYs to their respective MACs, the PHYs registered on the MDIO bus have to be referenced.

This document introduces two _DSD properties that are to be used for connecting PHYs on the MDIO bus [dsd-properties-rules] to the MAC layer.

These properties are defined in accordance with the ”Device Properties UUID For _DSD” [dsd-guide] document and the daffd814-6eba-4d8c-8a91-bc9bbf4aa301 UUID must be used in the Device Data Descriptors containing them.

1.5.1 phy-handle

For each MAC node, a device property ”phy-handle” is used to reference the PHY that is registered on an MDIO bus. This is mandatory for network interfaces that have PHYs connected to MAC via MDIO bus.

During the MDIO bus driver initialization, PHYs on this bus are probed using the _ADR object as shown below and are registered on the MDIO bus.

```
Scope(\_SB.MDI0)
{
  Device(PHY1) {
    Name (_ADR, 0x1)
  } // end of PHY1

  Device(PHY2) {
    Name (_ADR, 0x2)
  } // end of PHY2
}
```

Later, during the MAC driver initialization, the registered PHY devices have to be retrieved from the MDIO bus. For this, the MAC driver needs references to the previously registered PHYs which are provided as device object references (e.g. _SB.MDI0.PHY1).

1.5.2 phy-mode

The "phy-mode" _DSD property is used to describe the connection to the PHY. The valid values for "phy-mode" are defined in [ethernet-controller].

1.5.3 managed

Optional property, which specifies the PHY management type. The valid values for "managed" are defined in [ethernet-controller].

1.5.4 fixed-link

The "fixed-link" is described by a data-only subnode of the MAC port, which is linked in the _DSD package via hierarchical data extension (UUID dbb8e3e6-5886-4ba6-8795-1319f52a966b in accordance with [dsd-guide] "_DSD Implementation Guide" document). The subnode should comprise a required property ("speed") and possibly the optional ones - complete list of parameters and their values are specified in [ethernet-controller].

The following ASL example illustrates the usage of these properties.

1.5.5 DSDT entry for MDIO node

The MDIO bus has an SoC component (MDIO controller) and a platform component (PHYs on the MDIO bus).

a) Silicon Component This node describes the MDIO controller, MDI0 --

```
Scope(_SB)
{
  Device(MDI0) {
    Name(_HID, "NXP0006")
    Name(_CCA, 1)
    Name(_UID, 0)
    Name(_CRS, ResourceTemplate() {
      Memory32Fixed(ReadWrite, MDI0_BASE, MDI_LEN)
      Interrupt(ResourceConsumer, Level, ActiveHigh, Shared)
       {
         MDI0_IT
       }
    }) // end of _CRS for MDI0
  } // end of MDI0
}
```

b) Platform Component The PHY1 and PHY2 nodes represent the PHYs connected to MDIO bus MDI0 ---

```
Scope(\_SB.MDI0)
{
  Device(PHY1) {
    Name (_ADR, 0x1)
  } // end of PHY1

  Device(PHY2) {
    Name (_ADR, 0x2)
  } // end of PHY2
}
```

1.5.6 DSDT entries representing MAC nodes

Below are the MAC nodes where PHY nodes are referenced. phy-mode and phy-handle are used as explained earlier. --

```
Scope(\_SB.MCE0.PR17)
{
  Name (_DSD, Package () {
     ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
         Package () {
             Package (2) {"phy-mode", "rgmii-id"},
             Package (2) {"phy-handle", \_SB.MDI0.PHY1}
      }
   })
}

Scope(\_SB.MCE0.PR18)
{
  Name (_DSD, Package () {
    ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
        Package () {
            Package (2) {"phy-mode", "rgmii-id"},
            Package (2) {"phy-handle", \_SB.MDI0.PHY2}}
    }
  })
}
```

1.5.7 MAC node example where ”managed” property is specified.

```
Scope(\_SB.PP21.ETH0)
{
  Name (_DSD, Package () {
     ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
         Package () {
             Package () {"phy-mode", "sgmii"},
             Package () {"managed", "in-band-status"}
         }
   })
}
```

1.5.8 MAC node example with a ”fixed-link” subnode.

```
Scope(\_SB.PP21.ETH1)
{
  Name (_DSD, Package () {
    ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
         Package () {
             Package () {"phy-mode", "sgmii"},
         },
    ToUUID("dbb8e3e6-5886-4ba6-8795-1319f52a966b"),
         Package () {
             Package () {"fixed-link", "LNK0"}
         }
  })
  Name (LNK0, Package(){ // Data-only subnode of port
    ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
         Package () {
             Package () {"speed", 1000},
             Package () {"full-duplex", 1}
         }
  })
}
```

References

[phy] Documentation/networking/phy.rst

[dsd-properties-rules]
_DSD Device Properties Usage Rules

[ethernet-controller]
Documentation/devicetree/bindings/net/ethernet-controller.yaml

[dsd-guide] DSD Guide.
https://github.com/UEFI/DSD-Guide/blob/main/dsd-guide.adoc, referenced 2021-11-30.

1.6 ACPI Based Device Enumeration

ACPI 5 introduced a set of new resources (UartTSerialBus, I2cSerialBus, SpiSerialBus, GpioIo and GpioInt) which can be used in enumerating slave devices behind serial bus controllers.

In addition we are starting to see peripherals integrated in the SoC/Chipset to appear only in ACPI namespace. These are typically devices that are accessed through memory-mapped registers.

In order to support this and re-use the existing drivers as much as possible we decided to do following:

- Devices that have no bus connector resource are represented as platform devices.

- Devices behind real busses where there is a connector resource are represented as struct spi_device or struct i2c_client. Note that standard UARTs are not busses so there is no struct uart_device, although some of them may be represented by struct serdev_device.

As both ACPI and Device Tree represent a tree of devices (and their resources) this implementation follows the Device Tree way as much as possible.

The ACPI implementation enumerates devices behind busses (platform, SPI, I2C, and in some cases UART), creates the physical devices and binds them to their ACPI handle in the ACPI namespace.

This means that when ACPI_HANDLE(dev) returns non-NULL the device was enumerated from ACPI namespace. This handle can be used to extract other device-specific configuration. There is an example of this below.

1.6.1 Platform bus support

Since we are using platform devices to represent devices that are not connected to any physical bus we only need to implement a platform driver for the device and add supported ACPI IDs. If this same IP-block is used on some other non-ACPI platform, the driver might work out of the box or needs some minor changes.

Adding ACPI support for an existing driver should be pretty straightforward. Here is the simplest example:

```
static const struct acpi_device_id mydrv_acpi_match[] = {
        /* ACPI IDs here */
        { }
};
MODULE_DEVICE_TABLE(acpi, mydrv_acpi_match);

static struct platform_driver my_driver = {
        ...
        .driver = {
                .acpi_match_table = mydrv_acpi_match,
        },
};
```

If the driver needs to perform more complex initialization like getting and configuring GPIOs it can get its ACPI handle and extract this information from ACPI tables.

1.6.2 ACPI device objects

Generally speaking, there are two categories of devices in a system in which ACPI is used as an interface between the platform firmware and the OS: Devices that can be discovered and enumerated natively, through a protocol defined for the specific bus that they are on (for example, configuration space in PCI), without the platform firmware assistance, and devices that need to be described by the platform firmware so that they can be discovered. Still, for any device known to the platform firmware, regardless of which category it falls into, there can be a corresponding ACPI device object in the ACPI Namespace in which case the Linux kernel will create a struct acpi_device object based on it for that device.

Those struct acpi_device objects are never used for binding drivers to natively discoverable devices, because they are represented by other types of device objects (for example, struct pci_dev for PCI devices) that are bound to by device drivers (the corresponding struct acpi_device object is then used as an additional source of information on the configuration of the given device). Moreover, the core ACPI device enumeration code creates struct platform_device objects for the majority of devices that are discovered and enumerated with the help of the platform firmware and those platform device objects can be bound to by platform drivers in direct analogy with the natively enumerable devices case. Therefore it is logically inconsistent and so generally invalid to bind drivers to struct acpi_device objects, including drivers for devices that are discovered with the help of the platform firmware.

Historically, ACPI drivers that bound directly to struct acpi_device objects were implemented for some devices enumerated with the help of the platform firmware, but this is not recommended for any new drivers. As explained above, platform device objects are created for those devices as a rule (with a few exceptions that are not relevant here) and so platform drivers should be used for handling them, even though the corresponding ACPI device objects are the only source of device configuration information in that case.

For every device having a corresponding struct acpi_device object, the pointer to it is returned by the ACPI_COMPANION() macro, so it is always possible to get to the device configuration information stored in the ACPI device object this way. Accordingly, struct acpi_device can be regarded as a part of the interface between the kernel and the ACPI Namespace, whereas device objects of other types (for example, struct pci_dev or struct platform_device) are used for interacting with the rest of the system.

1.6.3 DMA support

DMA controllers enumerated via ACPI should be registered in the system to provide generic access to their resources. For example, a driver that would like to be accessible to slave devices via generic API call dma_request_chan() must register itself at the end of the probe function like this:

```
err = devm_acpi_dma_controller_register(dev, xlate_func, dw);
/* Handle the error if it's not a case of !CONFIG_ACPI */
```

and implement custom xlate function if needed (usually acpi_dma_simple_xlate() is enough) which converts the FixedDMA resource provided by struct acpi_dma_spec into the corresponding DMA channel. A piece of code for that case could look like:

```
#ifdef CONFIG_ACPI
struct filter_args {
        /* Provide necessary information for the filter_func */
        ...
};

static bool filter_func(struct dma_chan *chan, void *param)
{
        /* Choose the proper channel */
        ...
}

static struct dma_chan *xlate_func(struct acpi_dma_spec *dma_spec,
                struct acpi_dma *adma)
{
        dma_cap_mask_t cap;
        struct filter_args args;

        /* Prepare arguments for filter_func */
        ...
        return dma_request_channel(cap, filter_func, &args);
}
#else
static struct dma_chan *xlate_func(struct acpi_dma_spec *dma_spec,
                struct acpi_dma *adma)
{
        return NULL;
}
#endif
```

dma_request_chan() will call xlate_func() for each registered DMA controller. In the xlate function the proper channel must be chosen based on information in struct acpi_dma_spec and the properties of the controller provided by struct acpi_dma.

Clients must call dma_request_chan() with the string parameter that corresponds to a specific FixedDMA resource. By default ”tx” means the first entry of the FixedDMA resource array, ”rx” means the second entry. The table below shows a layout:

```
Device (I2C0)
{
        ...
        Method (_CRS, 0, NotSerialized)
        {
                Name (DBUF, ResourceTemplate ()
                {
                        FixedDMA (0x0018, 0x0004, Width32bit, _Y48)
                        FixedDMA (0x0019, 0x0005, Width32bit, )
                })
        ...
        }
}
```

So, the FixedDMA with request line 0x0018 is ”tx” and next one is ”rx” in this example.

In robust cases the client unfortunately needs to call acpi_dma_request_slave_chan_by_index() directly and therefore choose the specific FixedDMA resource by its index.

1.6.4 Named Interrupts

Drivers enumerated via ACPI can have names to interrupts in the ACPI table which can be used to get the IRQ number in the driver.

The interrupt name can be listed in _DSD as 'interrupt-names'. The names should be listed as an array of strings which will map to the Interrupt() resource in the ACPI table corresponding to its index.

The table below shows an example of its usage:

```
Device (DEV0) {
    ...
    Name (_CRS, ResourceTemplate() {
        ...
        Interrupt (ResourceConsumer, Level, ActiveHigh, Exclusive) {
            0x20,
            0x24
        }
    })

    Name (_DSD, Package () {
        ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
        Package () {
            Package () { "interrupt-names", Package () { "default", "alert" } }
↪,
        }
    ...
    })
}
```

The interrupt name 'default' will correspond to 0x20 in Interrupt() resource and 'alert' to 0x24. Note that only the Interrupt() resource is mapped and not GpioInt() or similar.

The driver can call the function - fwnode_irq_get_byname() with the fwnode and interrupt name as arguments to get the corresponding IRQ number.

1.6.5 SPI serial bus support

Slave devices behind SPI bus have SpiSerialBus resource attached to them. This is extracted automatically by the SPI core and the slave devices are enumerated once spi_register_master() is called by the bus driver.

Here is what the ACPI namespace for a SPI slave might look like:

```
Device (EEP0)
{
        Name (_ADR, 1)
        Name (_CID, Package () {
                "ATML0025",
                "AT25",
        })
        ...
        Method (_CRS, 0, NotSerialized)
        {
                SPISerialBus(1, PolarityLow, FourWireMode, 8,
                        ControllerInitiated, 1000000, ClockPolarityLow,
                        ClockPhaseFirst, "\\_SB.PCI0.SPI1",)
        }
        ...
```

The SPI device drivers only need to add ACPI IDs in a similar way to the platform device drivers. Below is an example where we add ACPI support to at25 SPI eeprom driver (this is meant for the above ACPI snippet):

```
static const struct acpi_device_id at25_acpi_match[] = {
        { "AT25", 0 },
        { }
};
MODULE_DEVICE_TABLE(acpi, at25_acpi_match);

static struct spi_driver at25_driver = {
        .driver = {
                ...
                .acpi_match_table = at25_acpi_match,
        },
};
```

Note that this driver actually needs more information like page size of the eeprom, etc. This information can be passed via _DSD method like:

```
Device (EEP0)
{
        ...
        Name (_DSD, Package ()
        {
                ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
                Package ()
                {
                        Package () { "size", 1024 },
                        Package () { "pagesize", 32 },
                        Package () { "address-width", 16 },
                }
        })
}
```

Then the at25 SPI driver can get this configuration by calling device property APIs during ->probe() phase like:

```
err = device_property_read_u32(dev, "size", &size);
if (err)
        ...error handling...

err = device_property_read_u32(dev, "pagesize", &page_size);
```

```
if (err)
        ...error handling...

err = device_property_read_u32(dev, "address-width", &addr_width);
if (err)
        ...error handling...
```

1.6.6 I2C serial bus support

The slaves behind I2C bus controller only need to add the ACPI IDs like with the platform and SPI drivers. The I2C core automatically enumerates any slave devices behind the controller device once the adapter is registered.

Below is an example of how to add ACPI support to the existing mpu3050 input driver:

```
static const struct acpi_device_id mpu3050_acpi_match[] = {
        { "MPU3050", 0 },
        { }
};
MODULE_DEVICE_TABLE(acpi, mpu3050_acpi_match);

static struct i2c_driver mpu3050_i2c_driver = {
        .driver = {
                .name   = "mpu3050",
                .pm     = &mpu3050_pm,
                .of_match_table = mpu3050_of_match,
                .acpi_match_table = mpu3050_acpi_match,
        },
        .probe          = mpu3050_probe,
        .remove         = mpu3050_remove,
        .id_table       = mpu3050_ids,
};
module_i2c_driver(mpu3050_i2c_driver);
```

1.6.7 Reference to PWM device

Sometimes a device can be a consumer of PWM channel. Obviously OS would like to know which one. To provide this mapping the special property has been introduced, i.e.:

```
Device (DEV)
{
    Name (_DSD, Package ()
    {
        ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
        Package () {
            Package () { "compatible", Package () { "pwm-leds" } },
            Package () { "label", "alarm-led" },
            Package () { "pwms",
                Package () {
                    "\\_SB.PCI0.PWM",  // <PWM device reference>
                    0,                 // <PWM index>
                    600000000,         // <PWM period>
                    0,                 // <PWM flags>
                }
            }
        }
    })
    ...
}
```

In the above example the PWM-based LED driver references to the PWM channel 0 of _SB.PCI0.PWM device with initial period setting equal to 600 ms (note that value is given in nanoseconds).

1.6.8 GPIO support

ACPI 5 introduced two new resources to describe GPIO connections: GpioIo and GpioInt. These resources can be used to pass GPIO numbers used by the device to the driver. ACPI 5.1 extended this with _DSD (Device Specific Data) which made it possible to name the GPIOs among other things.

For example:

```
Device (DEV)
{
        Method (_CRS, 0, NotSerialized)
        {
                Name (SBUF, ResourceTemplate()
                {
                        // Used to power on/off the device
                        GpioIo (Exclusive, PullNone, 0, 0,␣
↪IoRestrictionOutputOnly,
                                "\\_SB.PCI0.GPI0", 0, ResourceConsumer) { 85 }

                        // Interrupt for the device
                        GpioInt (Edge, ActiveHigh, ExclusiveAndWake, PullNone,␣
↪0,
                                 "\\_SB.PCI0.GPI0", 0, ResourceConsumer) { 88 }
                }

                Return (SBUF)
        }

        // ACPI 5.1 _DSD used for naming the GPIOs
        Name (_DSD, Package ()
        {
                ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
                Package ()
                {
                        Package () { "power-gpios", Package () { ^DEV, 0, 0, 0␣
↪} },
                        Package () { "irq-gpios", Package () { ^DEV, 1, 0, 0 }␣
↪},
                }
        })
        ...
}
```

These GPIO numbers are controller relative and path "_SB.PCI0.GPI0" specifies the path to the controller. In order to use these GPIOs in Linux we need to translate them to the corresponding Linux GPIO descriptors.

There is a standard GPIO API for that and it is documented in Documentation/admin-guide/gpio/.

In the above example we can get the corresponding two GPIO descriptors with a code like this:

```
#include <linux/gpio/consumer.h>
...

struct gpio_desc *irq_desc, *power_desc;

irq_desc = gpiod_get(dev, "irq");
if (IS_ERR(irq_desc))
        /* handle error */

power_desc = gpiod_get(dev, "power");
if (IS_ERR(power_desc))
        /* handle error */

/* Now we can use the GPIO descriptors */
```

There are also devm_* versions of these functions which release the descriptors once the device is released.

See *_DSD Device Properties Related to GPIO* for more information about the _DSD binding related to GPIOs.

1.6.9 RS-485 support

ACPI _DSD (Device Specific Data) can be used to describe RS-485 capability of UART.

For example:

```
Device (DEV)
{
        ...

        // ACPI 5.1 _DSD used for RS-485 capabilities
        Name (_DSD, Package ()
        {
                ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
                Package ()
                {
                        Package () {"rs485-rts-active-low", Zero},
                        Package () {"rs485-rx-active-high", Zero},
                        Package () {"rs485-rx-during-tx", Zero},
                }
        })
        ...
```

1.6.10 MFD devices

The MFD devices register their children as platform devices. For the child devices there needs to be an ACPI handle that they can use to reference parts of the ACPI namespace that relate to them. In the Linux MFD subsystem we provide two ways:

- The children share the parent ACPI handle.
- The MFD cell can specify the ACPI id of the device.

For the first case, the MFD drivers do not need to do anything. The resulting child platform device will have its ACPI_COMPANION() set to point to the parent device.

If the ACPI namespace has a device that we can match using an ACPI id or ACPI adr, the cell should be set like:

```
static struct mfd_cell_acpi_match my_subdevice_cell_acpi_match = {
        .pnpid = "XYZ0001",
        .adr = 0,
};

static struct mfd_cell my_subdevice_cell = {
        .name = "my_subdevice",
        /* set the resources relative to the parent */
        .acpi_match = &my_subdevice_cell_acpi_match,
};
```

The ACPI id ”XYZ0001” is then used to lookup an ACPI device directly under the MFD device and if found, that ACPI companion device is bound to the resulting child platform device.

1.6.11 Device Tree namespace link device ID

The Device Tree protocol uses device identification based on the ”compatible” property whose value is a string or an array of strings recognized as device identifiers by drivers and the driver core. The set of all those strings may be regarded as a device identification namespace analogous to the ACPI/PNP device ID namespace. Consequently, in principle it should not be necessary to allocate a new (and arguably redundant) ACPI/PNP device ID for a devices with an existing identification string in the Device Tree (DT) namespace, especially if that ID is only needed to indicate that a given device is compatible with another one, presumably having a matching driver in the kernel already.

In ACPI, the device identification object called _CID (Compatible ID) is used to list the IDs of devices the given one is compatible with, but those IDs must belong to one of the namespaces prescribed by the ACPI specification (see Section 6.1.2 of ACPI 6.0 for details) and the DT namespace is not one of them. Moreover, the specification mandates that either a _HID or an

_ADR identification object be present for all ACPI objects representing devices (Section 6.1 of ACPI 6.0). For non-enumerable bus types that object must be _HID and its value must be a device ID from one of the namespaces prescribed by the specification too.

The special DT namespace link device ID, PRP0001, provides a means to use the existing DT-compatible device identification in ACPI and to satisfy the above requirements following from the ACPI specification at the same time. Namely, if PRP0001 is returned by _HID, the ACPI subsystem will look for the "compatible" property in the device object's _DSD and will use the value of that property to identify the corresponding device in analogy with the original DT device identification algorithm. If the "compatible" property is not present or its value is not valid, the device will not be enumerated by the ACPI subsystem. Otherwise, it will be enumerated automatically as a platform device (except when an I2C or SPI link from the device to its parent is present, in which case the ACPI core will leave the device enumeration to the parent's driver) and the identification strings from the "compatible" property value will be used to find a driver for the device along with the device IDs listed by _CID (if present).

Analogously, if PRP0001 is present in the list of device IDs returned by _CID, the identification strings listed by the "compatible" property value (if present and valid) will be used to look for a driver matching the device, but in that case their relative priority with respect to the other device IDs listed by _HID and _CID depends on the position of PRP0001 in the _CID return package. Specifically, the device IDs returned by _HID and preceding PRP0001 in the _CID return package will be checked first. Also in that case the bus type the device will be enumerated to depends on the device ID returned by _HID.

For example, the following ACPI sample might be used to enumerate an lm75-type I2C temperature sensor and match it to the driver using the Device Tree namespace link:

```
Device (TMP0)
{
        Name (_HID, "PRP0001")
        Name (_DSD, Package () {
                ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
                Package () {
                        Package () { "compatible", "ti,tmp75" },
                }
        })
        Method (_CRS, 0, Serialized)
        {
                Name (SBUF, ResourceTemplate ()
                {
                        I2cSerialBusV2 (0x48, ControllerInitiated,
                                400000, AddressingMode7Bit,
                                "\\_SB.PCI0.I2C1", 0x00,
                                ResourceConsumer, , Exclusive,)
                })
                Return (SBUF)
        }
}
```

It is valid to define device objects with a _HID returning PRP0001 and without the "compatible" property in the _DSD or a _CID as long as one of their ancestors provides a _DSD with a valid "compatible" property. Such device objects are then simply regarded as additional "blocks" providing hierarchical configuration information to the driver of the composite ancestor device.

However, PRP0001 can only be returned from either _HID or _CID of a device object if all of the properties returned by the _DSD associated with it (either the _DSD of the device object itself or the _DSD of its ancestor in the "composite device" case described above) can be used in the ACPI environment. Otherwise, the _DSD itself is regarded as invalid and therefore the "compatible" property returned by it is meaningless.

Refer to *_DSD Device Properties Usage Rules* for more information.

1.6.12 PCI hierarchy representation

Sometimes it could be useful to enumerate a PCI device, knowing its position on the PCI bus.

For example, some systems use PCI devices soldered directly on the mother board, in a fixed position (ethernet, Wi-Fi, serial ports, etc.). In this conditions it is possible to refer to these PCI devices knowing their position on the PCI bus topology.

To identify a PCI device, a complete hierarchical description is required, from the chipset root port to the final device, through all the intermediate bridges/switches of the board.

For example, let's assume we have a system with a PCIe serial port, an Exar XR17V3521, soldered on the main board. This UART chip also includes 16 GPIOs and we want to add the property `gpio-line-names` [1] to these pins. In this case, the `lspci` output for this component is:

```
07:00.0 Serial controller: Exar Corp. XR17V3521 Dual PCIe UART (rev 03)
```

The complete `lspci` output (manually reduced in length) is:

```
00:00.0 Host bridge: Intel Corp... Host Bridge (rev 0d)
...
00:13.0 PCI bridge: Intel Corp... PCI Express Port A #1 (rev fd)
00:13.1 PCI bridge: Intel Corp... PCI Express Port A #2 (rev fd)
00:13.2 PCI bridge: Intel Corp... PCI Express Port A #3 (rev fd)
00:14.0 PCI bridge: Intel Corp... PCI Express Port B #1 (rev fd)
00:14.1 PCI bridge: Intel Corp... PCI Express Port B #2 (rev fd)
...
05:00.0 PCI bridge: Pericom Semiconductor Device 2404 (rev 05)
06:01.0 PCI bridge: Pericom Semiconductor Device 2404 (rev 05)
06:02.0 PCI bridge: Pericom Semiconductor Device 2404 (rev 05)
06:03.0 PCI bridge: Pericom Semiconductor Device 2404 (rev 05)
07:00.0 Serial controller: Exar Corp. XR17V3521 Dual PCIe UART (rev 03) <--␣
↪Exar
...
```

The bus topology is:

```
-[0000:00]-+-00.0
           ...
           +-13.0-[01]----00.0
           +-13.1-[02]----00.0
           +-13.2-[03]--
           +-14.0-[04]----00.0
           +-14.1-[05-09]----00.0-[06-09]--+-01.0-[07]----00.0 <-- Exar
           |                               +-02.0-[08]----00.0
           |                               \-03.0-[09]--
           ...
           \-1f.1
```

To describe this Exar device on the PCI bus, we must start from the ACPI name of the chipset bridge (also called ”root port”) with address:

```
Bus: 0 - Device: 14 - Function: 1
```

To find this information, it is necessary to disassemble the BIOS ACPI tables, in particular the DSDT (see also [2]):

```
mkdir ~/tables/
cd ~/tables/
acpidump > acpidump
acpixtract -a acpidump
iasl -e ssdt?.* -d dsdt.dat
```

Now, in the dsdt.dsl, we have to search the device whose address is related to 0x14 (device) and 0x01 (function). In this case we can find the following device:

```
Scope (_SB.PCI0)
{
... other definitions follow ...
        Device (RP02)
        {
                Method (_ADR, 0, NotSerialized)  // _ADR: Address
                {
                        If ((RPA2 != Zero))
                        {
                                Return (RPA2) /* \RPA2 */
                        }
                        Else
                        {
                                Return (0x00140001)
                        }
                }
... other definitions follow ...
```

and the _ADR method [3] returns exactly the device/function couple that we are looking for. With this information and analyzing the above `lspci` output (both the devices list and the devices tree), we can write the following ACPI description for the Exar PCIe UART, also adding the list of its GPIO line names:

```
Scope (_SB.PCI0.RP02)
{
        Device (BRG1) //Bridge
        {
                Name (_ADR, 0x0000)

                Device (BRG2) //Bridge
                {
                        Name (_ADR, 0x00010000)

                        Device (EXAR)
                        {
                                Name (_ADR, 0x0000)

                                Name (_DSD, Package ()
                                {
                                        ToUUID("daffd814-6eba-4d8c-8a91-
↪bc9bbf4aa301"),
                                        Package ()
                                        {
                                                Package ()
                                                {
                                                        "gpio-line-names",
                                                        Package ()
                                                        {
                                                                "mode_232",
                                                                "mode_422",
                                                                "mode_485",
                                                                "misc_1",
                                                                "misc_2",
                                                                "misc_3",
                                                                "",
                                                                "",
                                                                "aux_1",
                                                                "aux_2",
                                                                "aux_3",
                                                        }
                                                }
                                        }
                                })
                        }
                }
        }
}
```

The location "_SB.PCI0.RP02" is obtained by the above investigation in the dsdt.dsl table, whereas the device names "BRG1", "BRG2" and "EXAR" are created analyzing the position of the Exar UART in the PCI bus topology.

1.6.13 References

[1] *_DSD Device Properties Related to GPIO*

[2] Documentation/admin-guide/acpi/initrd_table_override.rst

[3] ACPI Specifications, Version 6.3 - Paragraph 6.1.1 _ADR Address)
https://uefi.org/sites/default/files/resources/ACPI_6_3_May16.pdf, referenced 2020-11-18

1.7 ACPI _OSI and _REV methods

An ACPI BIOS can use the "Operating System Interfaces" method (_OSI) to find out what the operating system supports. Eg. If BIOS AML code includes _OSI("XYZ"), the kernel's AML interpreter can evaluate that method, look to see if it supports 'XYZ' and answer YES or NO to the BIOS.

The ACPI _REV method returns the "Revision of the ACPI specification that OSPM supports"

This document explains how and why the BIOS and Linux should use these methods. It also explains how and why they are widely misused.

1.7.1 How to use _OSI

Linux runs on two groups of machines -- those that are tested by the OEM to be compatible with Linux, and those that were never tested with Linux, but where Linux was installed to replace the original OS (Windows or OSX).

The larger group is the systems tested to run only Windows. Not only that, but many were tested to run with just one specific version of Windows. So even though the BIOS may use _OSI to query what version of Windows is running, only a single path through the BIOS has actually been tested. Experience shows that taking untested paths through the BIOS exposes Linux to an entire category of BIOS bugs. For this reason, Linux _OSI defaults must continue to claim compatibility with all versions of Windows.

But Linux isn't actually compatible with Windows, and the Linux community has also been hurt with regressions when Linux adds the latest version of Windows to its list of _OSI strings. So it is possible that additional strings will be more thoroughly vetted before shipping upstream in the future. But it is likely that they will all eventually be added.

What should an OEM do if they want to support Linux and Windows using the same BIOS image? Often they need to do something different for Linux to deal with how Linux is different from Windows.

In this case, the OEM should create custom ASL to be executed by the Linux kernel and changes to Linux kernel drivers to execute this custom ASL. The easiest way to accomplish this is to introduce a device specific method (_DSM) that is called from the Linux kernel.

In the past the kernel used to support something like: _OSI("Linux-OEM-my_interface_name") where 'OEM' is needed if this is an OEM-specific hook, and 'my_interface_name' describes the hook, which could be a quirk, a bug, or a bug-fix.

However this was discovered to be abused by other BIOS vendors to change completely unrelated code on completely unrelated systems. This prompted an evaluation of all of its uses. This uncovered that they aren't needed for any of the original reasons. As such, the kernel will not respond to any custom Linux-* strings by default.

That was easy. Read on, to find out how to do it wrong.

1.7.2 Before _OSI, there was _OS

ACPI 1.0 specified "_OS" as an "object that evaluates to a string that identifies the operating system."

The ACPI BIOS flow would include an evaluation of _OS, and the AML interpreter in the kernel would return to it a string identifying the OS:

Windows 98, SE: "Microsoft Windows" Windows ME: "Microsoft WindowsME:Millennium Edition" Windows NT: "Microsoft Windows NT"

The idea was on a platform tasked with running multiple OS's, the BIOS could use _OS to enable devices that an OS might support, or enable quirks or bug workarounds necessary to make the platform compatible with that pre-existing OS.

But _OS had fundamental problems. First, the BIOS needed to know the name of every possible version of the OS that would run on it, and needed to know all the quirks of those OS's. Certainly it would make more sense for the BIOS to ask *specific* things of the OS, such "do you support a specific interface", and thus in ACPI 3.0, _OSI was born to replace _OS.

_OS was abandoned, though even today, many BIOS look for _OS "Microsoft Windows NT", though it seems somewhat far-fetched that anybody would install those old operating systems over what came with the machine.

Linux answers "Microsoft Windows NT" to please that BIOS idiom. That is the *only* viable strategy, as that is what modern Windows does, and so doing otherwise could steer the BIOS down an untested path.

1.7.3 _OSI is born, and immediately misused

With _OSI, the *BIOS* provides the string describing an interface, and asks the OS: "YES/NO, are you compatible with this interface?"

eg. _OSI("3.0 Thermal Model") would return TRUE if the OS knows how to deal with the thermal extensions made to the ACPI 3.0 specification. An old OS that doesn't know about those extensions would answer FALSE, and a new OS may be able to return TRUE.

For an OS-specific interface, the ACPI spec said that the BIOS and the OS were to agree on a string of the form such as "Windows-interface_name".

But two bad things happened. First, the Windows ecosystem used _OSI not as designed, but as a direct replacement for _OS -- identifying the OS version, rather than an OS supported interface. Indeed, right from the start, the ACPI 3.0 spec itself codified this misuse in example code using _OSI("Windows 2001").

This misuse was adopted and continues today.

Linux had no choice but to also return TRUE to _OSI("Windows 2001") and its successors. To do otherwise would virtually guarantee breaking a BIOS that has been tested only with that _OSI returning TRUE.

This strategy is problematic, as Linux is never completely compatible with the latest version of Windows, and sometimes it takes more than a year to iron out incompatibilities.

Not to be out-done, the Linux community made things worse by returning TRUE to _OSI(”Linux”). Doing so is even worse than the Windows misuse of _OSI, as ”Linux” does not even contain any version information. _OSI(”Linux”) led to some BIOS' malfunctioning due to BIOS writer's using it in untested BIOS flows. But some OEM's used _OSI(”Linux”) in tested flows to support real Linux features. In 2009, Linux removed _OSI(”Linux”), and added a cmdline parameter to restore it for legacy systems still needed it. Further a BIOS_BUG warning prints for all BIOS's that invoke it.

No BIOS should use _OSI(”Linux”).

The result is a strategy for Linux to maximize compatibility with ACPI BIOS that are tested on Windows machines. There is a real risk of over-stating that compatibility; but the alternative has often been catastrophic failure resulting from the BIOS taking paths that were never validated under *any* OS.

1.7.4 Do not use _REV

Since _OSI(”Linux”) went away, some BIOS writers used _REV to support Linux and Windows differences in the same BIOS.

_REV was defined in ACPI 1.0 to return the version of ACPI supported by the OS and the OS AML interpreter.

Modern Windows returns _REV = 2. Linux used ACPI_CA_SUPPORT_LEVEL, which would increment, based on the version of the spec supported.

Unfortunately, _REV was also misused. eg. some BIOS would check for _REV = 3, and do something for Linux, but when Linux returned _REV = 4, that support broke.

In response to this problem, Linux returns _REV = 2 always, from mid-2015 onward. The ACPI specification will also be updated to reflect that _REV is deprecated, and always returns 2.

1.7.5 Apple Mac and _OSI(”Darwin”)

On Apple's Mac platforms, the ACPI BIOS invokes _OSI(”Darwin”) to determine if the machine is running Apple OSX.

Like Linux's _OSI(”*Windows*”) strategy, Linux defaults to answering YES to _OSI(”Darwin”) to enable full access to the hardware and validated BIOS paths seen by OSX. Just like on Windows-tested platforms, this strategy has risks.

Starting in Linux-3.18, the kernel answered YES to _OSI(”Darwin”) for the purpose of enabling Mac Thunderbolt support. Further, if the kernel noticed _OSI(”Darwin”) being invoked, it additionally disabled all _OSI(”*Windows*”) to keep poorly written Mac BIOS from going down untested combinations of paths.

The Linux-3.18 change in default caused power regressions on Mac laptops, and the 3.18 implementation did not allow changing the default via cmdline ”acpi_osi=!Darwin”. Linux-4.7 fixed the ability to use acpi_osi=!Darwin as a workaround, and we hope to see Mac Thunderbolt power management support in Linux-4.11.

1.8 Linux ACPI Custom Control Method How To

Author
Zhang Rui <rui.zhang@intel.com>

Linux supports customizing ACPI control methods at runtime.

Users can use this to:

1. override an existing method which may not work correctly, or just for debugging purposes.
2. insert a completely new method in order to create a missing method such as _OFF, _ON, _STA, _INI, etc.

For these cases, it is far simpler to dynamically install a single control method rather than override the entire DSDT, because kernel rebuild/reboot is not needed and test result can be got in minutes.

Note:

- Only ACPI METHOD can be overridden, any other object types like ”Device”, ”OperationRegion”, are not recognized. Methods declared inside scope operators are also not supported.
- The same ACPI control method can be overridden for many times, and it's always the latest one that used by Linux/kernel.
- To get the ACPI debug object output (Store (AAAA, Debug)), please run:

```
echo 1 > /sys/module/acpi/parameters/aml_debug_output
```

1.8.1 1. override an existing method

a) get the ACPI table via ACPI sysfs I/F. e.g. to get the DSDT, just run ”cat /sys/firmware/acpi/tables/DSDT > /tmp/dsdt.dat”

b) disassemble the table by running ”iasl -d dsdt.dat”.

c) rewrite the ASL code of the method and save it in a new file,

d) package the new file (psr.asl) to an ACPI table format. Here is an example of a customized _SB._AC._PSR method:

```
DefinitionBlock ("", "SSDT", 1, "", "", 0x20080715)
{
   Method (\_SB_.AC._PSR, 0, NotSerialized)
   {
      Store ("In AC _PSR", Debug)
      Return (ACON)
   }
}
```

Note that the full pathname of the method in ACPI namespace should be used.

e) assemble the file to generate the AML code of the method. e.g. ”iasl -vw 6084 psr.asl” (psr.aml is generated as a result) If parameter ”-vw 6084” is not supported by your iASL compiler, please try a newer version.

f) mount debugfs by ”mount -t debugfs none /sys/kernel/debug”

g) override the old method via the debugfs by running ”cat /tmp/psr.aml > /sys/kernel/debug/acpi/custom_method”

1.8.2 2. insert a new method

This is easier than overriding an existing method. We just need to create the ASL code of the method we want to insert and then follow the step c) ~ g) in section 1.

1.8.3 3. undo your changes

The ”undo” operation is not supported for a new inserted method right now, i.e. we can not remove a method currently. For an overridden method, in order to undo your changes, please save a copy of the method original ASL code in step c) section 1, and redo step c) ~ g) to override the method with the original one.

Note: We can use a kernel with multiple custom ACPI method running, But each individual write to debugfs can implement a SINGLE method override. i.e. if we want to insert/override multiple ACPI methods, we need to redo step c) ~ g) for multiple times.

Note: Be aware that root can mis-use this driver to modify arbitrary memory and gain additional rights, if root's privileges got restricted (for example if root is not allowed to load additional modules after boot).

1.9 ACPICA Trace Facility

Copyright
© 2015, Intel Corporation

Author
Lv Zheng <lv.zheng@intel.com>

1.9.1 Abstract

This document describes the functions and the interfaces of the method tracing facility.

1.9.2 Functionalities and usage examples

ACPICA provides method tracing capability. And two functions are currently implemented using this capability.

Log reducer

ACPICA subsystem provides debugging outputs when CONFIG_ACPI_DEBUG is enabled. The debugging messages which are deployed via ACPI_DEBUG_PRINT() macro can be reduced at 2 levels - per-component level (known as debug layer, configured via /sys/module/acpi/parameters/debug_layer) and per-type level (known as debug level, configured via /sys/module/acpi/parameters/debug_level).

But when the particular layer/level is applied to the control method evaluations, the quantity of the debugging outputs may still be too large to be put into the kernel log buffer. The idea thus is worked out to only enable the particular debug layer/level (normally more detailed) logs when the control method evaluation is started, and disable the detailed logging when the control method evaluation is stopped.

The following command examples illustrate the usage of the ”log reducer” functionality:

a. Filter out the debug layer/level matched logs when control methods are being evaluated:

```
# cd /sys/module/acpi/parameters
# echo "0xXXXXXXXX" > trace_debug_layer
# echo "0xYYYYYYYY" > trace_debug_level
# echo "enable" > trace_state
```

b. Filter out the debug layer/level matched logs when the specified control method is being evaluated:

```
# cd /sys/module/acpi/parameters
# echo "0xXXXXXXXX" > trace_debug_layer
# echo "0xYYYYYYYY" > trace_debug_level
# echo "\PPPP.AAAA.TTTT.HHHH" > trace_method_name
# echo "method" > /sys/module/acpi/parameters/trace_state
```

c. Filter out the debug layer/level matched logs when the specified control method is being evaluated for the first time:

```
# cd /sys/module/acpi/parameters
# echo "0xXXXXXXXX" > trace_debug_layer
# echo "0xYYYYYYYY" > trace_debug_level
# echo "\PPPP.AAAA.TTTT.HHHH" > trace_method_name
# echo "method-once" > /sys/module/acpi/parameters/trace_state
```

Where:

0xXXXXXXXX/0xYYYYYYYY
Refer to *ACPI CA Debug Output* for possible debug layer/level masking values.

PPPP.AAAA.TTTT.HHHH
Full path of a control method that can be found in the ACPI namespace. It needn't be an entry of a control method evaluation.

AML tracer

There are special log entries added by the method tracing facility at the "trace points" the AML interpreter starts/stops to execute a control method, or an AML opcode. Note that the format of the log entries are subject to change:

```
[    0.186427]   exdebug-0398 ex_trace_point        : Method Begin␣
↪[0xf58394d8:\_SB.PCI0.LPCB.ECOK] execution.
[    0.186630]   exdebug-0398 ex_trace_point        : Opcode Begin␣
↪[0xf5905c88:If] execution.
[    0.186820]   exdebug-0398 ex_trace_point        : Opcode Begin␣
↪[0xf5905cc0:LEqual] execution.
[    0.187010]   exdebug-0398 ex_trace_point        : Opcode Begin␣
↪[0xf5905a20:-NamePath-] execution.
[    0.187214]   exdebug-0398 ex_trace_point        : Opcode End [0xf5905a20:-
↪NamePath-] execution.
[    0.187407]   exdebug-0398 ex_trace_point        : Opcode Begin␣
↪[0xf5905f60:One] execution.
[    0.187594]   exdebug-0398 ex_trace_point        : Opcode End␣
↪[0xf5905f60:One] execution.
[    0.187789]   exdebug-0398 ex_trace_point        : Opcode End␣
↪[0xf5905cc0:LEqual] execution.
[    0.187980]   exdebug-0398 ex_trace_point        : Opcode Begin␣
↪[0xf5905cc0:Return] execution.
[    0.188146]   exdebug-0398 ex_trace_point        : Opcode Begin␣
↪[0xf5905f60:One] execution.
[    0.188334]   exdebug-0398 ex_trace_point        : Opcode End␣
↪[0xf5905f60:One] execution.
[    0.188524]   exdebug-0398 ex_trace_point        : Opcode End␣
↪[0xf5905cc0:Return] execution.
[    0.188712]   exdebug-0398 ex_trace_point        : Opcode End␣
↪[0xf5905c88:If] execution.
[    0.188903]   exdebug-0398 ex_trace_point        : Method End [0xf58394d8:\_
↪SB.PCI0.LPCB.ECOK] execution.
```

Developers can utilize these special log entries to track the AML interpretation, thus can aid issue debugging and performance tuning. Note that, as the "AML tracer" logs are implemented via ACPI_DEBUG_PRINT() macro, CONFIG_ACPI_DEBUG is also required to be enabled for enabling "AML tracer" logs.

The following command examples illustrate the usage of the "AML tracer" functionality:

a. Filter out the method start/stop "AML tracer" logs when control methods are being evaluated:

   ```
   # cd /sys/module/acpi/parameters
   # echo "0x80" > trace_debug_layer
   # echo "0x10" > trace_debug_level
   # echo "enable" > trace_state
   ```

b. Filter out the method start/stop "AML tracer" when the specified control method is being evaluated:

```
# cd /sys/module/acpi/parameters
# echo "0x80" > trace_debug_layer
# echo "0x10" > trace_debug_level
# echo "\PPPP.AAAA.TTTT.HHHH" > trace_method_name
# echo "method" > trace_state
```

c. Filter out the method start/stop ”AML tracer” logs when the specified control method is being evaluated for the first time:

```
# cd /sys/module/acpi/parameters
# echo "0x80" > trace_debug_layer
# echo "0x10" > trace_debug_level
# echo "\PPPP.AAAA.TTTT.HHHH" > trace_method_name
# echo "method-once" > trace_state
```

d. Filter out the method/opcode start/stop ”AML tracer” when the specified control method is being evaluated:

```
# cd /sys/module/acpi/parameters
# echo "0x80" > trace_debug_layer
# echo "0x10" > trace_debug_level
# echo "\PPPP.AAAA.TTTT.HHHH" > trace_method_name
# echo "opcode" > trace_state
```

e. Filter out the method/opcode start/stop ”AML tracer” when the specified control method is being evaluated for the first time:

```
# cd /sys/module/acpi/parameters
# echo "0x80" > trace_debug_layer
# echo "0x10" > trace_debug_level
# echo "\PPPP.AAAA.TTTT.HHHH" > trace_method_name
# echo "opcode-opcode" > trace_state
```

Note that all above method tracing facility related module parameters can be used as the boot parameters, for example:

```
acpi.trace_debug_layer=0x80 acpi.trace_debug_level=0x10 \
acpi.trace_method_name=\_SB.LID0._LID acpi.trace_state=opcode-once
```

1.9.3 Interface descriptions

All method tracing functions can be configured via ACPI module parameters that are accessible at /sys/module/acpi/parameters/:

trace_method_name
The full path of the AML method that the user wants to trace.

Note that the full path shouldn't contain the trailing ”_”s in its name segments but may contain ”” to form an absolute path.

trace_debug_layer
The temporary debug_layer used when the tracing feature is enabled.

Using ACPI_EXECUTER (0x80) by default, which is the debug_layer used to match all ”AML tracer” logs.

trace_debug_level
The temporary debug_level used when the tracing feature is enabled.

Using ACPI_LV_TRACE_POINT (0x10) by default, which is the debug_level used to match all ”AML tracer” logs.

trace_state
The status of the tracing feature.

Users can enable/disable this debug tracing feature by executing the following command:

```
# echo string > /sys/module/acpi/parameters/trace_state
```

Where ”string” should be one of the following:

”disable”
Disable the method tracing feature.

”enable”
Enable the method tracing feature.

ACPICA debugging messages matching ”trace_debug_layer/trace_debug_level” during any method execution will be logged.

”method”
Enable the method tracing feature.

ACPICA debugging messages matching ”trace_debug_layer/trace_debug_level” during method execution of ”trace_method_name” will be logged.

”method-once”
Enable the method tracing feature.

ACPICA debugging messages matching ”trace_debug_layer/trace_debug_level” during method execution of ”trace_method_name” will be logged only once.

”opcode”
Enable the method tracing feature.

ACPICA debugging messages matching ”trace_debug_layer/trace_debug_level” during method/opcode execution of ”trace_method_name” will be logged.

”opcode-once”
Enable the method tracing feature.

ACPICA debugging messages matching ”trace_debug_layer/trace_debug_level” during method/opcode execution of ”trace_method_name” will be logged only once.

Note that, the difference between the ”enable” and other feature enabling options are:

1. When ”enable” is specified, since ”trace_debug_layer/trace_debug_level” shall apply to all control method evaluations, after configuring ”trace_state” to ”enable”, ”trace_method_name” will be reset to NULL.
2. When ”method/opcode” is specified, if ”trace_method_name” is NULL when ”trace_state” is configured to these options, the ”trace_debug_layer/trace_debug_level” will apply to all control method evaluations.

1.10 _DSD Device Properties Usage Rules

1.10.1 Properties, Property Sets and Property Subsets

The _DSD (Device Specific Data) configuration object, introduced in ACPI 5.1, allows any type of device configuration data to be provided via the ACPI namespace. In principle, the format of the data may be arbitrary, but it has to be identified by a UUID which must be recognized by the driver processing the _DSD output. However, there are generic UUIDs defined for _DSD recognized by the ACPI subsystem in the Linux kernel which automatically processes the data packages associated with them and makes those data available to device drivers as "device properties".

A device property is a data item consisting of a string key and a value (of a specific type) associated with it.

In the ACPI _DSD context it is an element of the sub-package following the generic Device Properties UUID in the _DSD return package as specified in the section titled "Well-Known _DSD UUIDs and Data Structure Formats" sub-section "Device Properties UUID" in _DSD (Device Specific Data) Implementation Guide document[1].

It also may be regarded as the definition of a key and the associated data type that can be returned by _DSD in the Device Properties UUID sub-package for a given device.

A property set is a collection of properties applicable to a hardware entity like a device. In the ACPI _DSD context it is the set of all properties that can be returned in the Device Properties UUID sub-package for the device in question.

Property subsets are nested collections of properties. Each of them is associated with an additional key (name) allowing the subset to be referred to as a whole (and to be treated as a separate entity). The canonical representation of property subsets is via the mechanism specified in the section titled "Well-Known _DSD UUIDs and Data Structure Formats" sub-section "Hierarchical Data Extension UUID" in _DSD (Device Specific Data) Implementation Guide document[1].

Property sets may be hierarchical. That is, a property set may contain multiple property subsets that each may contain property subsets of its own and so on.

1.10.2 General Validity Rule for Property Sets

Valid property sets must follow the guidance given by the Device Properties UUID definition document [1].

_DSD properties are intended to be used in addition to, and not instead of, the existing mechanisms defined by the ACPI specification. Therefore, as a rule, they should only be used if the ACPI specification does not make direct provisions for handling the underlying use case. It generally is invalid to return property sets which do not follow that rule from _DSD in data packages associated with the Device Properties UUID.

[1] https://github.com/UEFI/DSD-Guide

Additional Considerations

There are cases in which, even if the general rule given above is followed in principle, the property set may still not be regarded as a valid one.

For example, that applies to device properties which may cause kernel code (either a device driver or a library/subsystem) to access hardware in a way possibly leading to a conflict with AML methods in the ACPI namespace. In particular, that may happen if the kernel code uses device properties to manipulate hardware normally controlled by ACPI methods related to power management, like _PSx and _DSW (for device objects) or _ON and _OFF (for power resource objects), or by ACPI device disabling/enabling methods, like _DIS and _SRS.

In all cases in which kernel code may do something that will confuse AML as a result of using device properties, the device properties in question are not suitable for the ACPI environment and consequently they cannot belong to a valid property set.

1.10.3 Property Sets and Device Tree Bindings

It often is useful to make _DSD return property sets that follow Device Tree bindings.

In those cases, however, the above validity considerations must be taken into account in the first place and returning invalid property sets from _DSD must be avoided. For this reason, it may not be possible to make _DSD return a property set following the given DT binding literally and completely. Still, for the sake of code re-use, it may make sense to provide as much of the configuration data as possible in the form of device properties and complement that with an ACPI-specific mechanism suitable for the use case at hand.

In any case, property sets following DT bindings literally should not be expected to automatically work in the ACPI environment regardless of their contents.

1.10.4 References

1.11 ACPI CA Debug Output

The ACPI CA can generate debug output. This document describes how to use this facility.

1.11.1 Compile-time configuration

The ACPI CA debug output is globally enabled by CONFIG_ACPI_DEBUG. If this config option is not set, the debug messages are not even built into the kernel.

1.11.2 Boot- and run-time configuration

When CONFIG_ACPI_DEBUG=y, you can select the component and level of messages you're interested in. At boot-time, use the acpi.debug_layer and acpi.debug_level kernel command line options. After boot, you can use the debug_layer and debug_level files in /sys/module/acpi/parameters/ to control the debug messages.

1.11.3 debug_layer (component)

The "debug_layer" is a mask that selects components of interest, e.g., a specific part of the ACPI interpreter. To build the debug_layer bitmask, look for the "#define _COMPONENT" in an ACPI source file.

You can set the debug_layer mask at boot-time using the acpi.debug_layer command line argument, and you can change it after boot by writing values to /sys/module/acpi/parameters/debug_layer.

The possible components are defined in include/acpi/acoutput.h.

Reading /sys/module/acpi/parameters/debug_layer shows the supported mask values:

```
ACPI_UTILITIES                  0x00000001
ACPI_HARDWARE                   0x00000002
ACPI_EVENTS                     0x00000004
ACPI_TABLES                     0x00000008
ACPI_NAMESPACE                  0x00000010
ACPI_PARSER                     0x00000020
ACPI_DISPATCHER                 0x00000040
ACPI_EXECUTER                   0x00000080
ACPI_RESOURCES                  0x00000100
ACPI_CA_DEBUGGER                0x00000200
ACPI_OS_SERVICES                0x00000400
ACPI_CA_DISASSEMBLER            0x00000800
ACPI_COMPILER                   0x00001000
ACPI_TOOLS                      0x00002000
```

1.11.4 debug_level

The "debug_level" is a mask that selects different types of messages, e.g., those related to initialization, method execution, informational messages, etc. To build debug_level, look at the level specified in an ACPI_DEBUG_PRINT() statement.

The ACPI interpreter uses several different levels, but the Linux ACPI core and ACPI drivers generally only use ACPI_LV_INFO.

You can set the debug_level mask at boot-time using the acpi.debug_level command line argument, and you can change it after boot by writing values to /sys/module/acpi/parameters/debug_level.

The possible levels are defined in include/acpi/acoutput.h. Reading /sys/module/acpi/parameters/debug_level shows the supported mask values, currently these:

```
ACPI_LV_INIT                0x00000001
ACPI_LV_DEBUG_OBJECT        0x00000002
ACPI_LV_INFO                0x00000004
ACPI_LV_INIT_NAMES          0x00000020
ACPI_LV_PARSE               0x00000040
ACPI_LV_LOAD                0x00000080
ACPI_LV_DISPATCH            0x00000100
ACPI_LV_EXEC                0x00000200
ACPI_LV_NAMES               0x00000400
ACPI_LV_OPREGION            0x00000800
ACPI_LV_BFIELD              0x00001000
ACPI_LV_TABLES              0x00002000
ACPI_LV_VALUES              0x00004000
ACPI_LV_OBJECTS             0x00008000
ACPI_LV_RESOURCES           0x00010000
ACPI_LV_USER_REQUESTS       0x00020000
ACPI_LV_PACKAGE             0x00040000
ACPI_LV_ALLOCATIONS         0x00100000
ACPI_LV_FUNCTIONS           0x00200000
ACPI_LV_OPTIMIZATIONS       0x00400000
ACPI_LV_MUTEX               0x01000000
ACPI_LV_THREADS             0x02000000
ACPI_LV_IO                  0x04000000
ACPI_LV_INTERRUPTS          0x08000000
ACPI_LV_AML_DISASSEMBLE     0x10000000
ACPI_LV_VERBOSE_INFO        0x20000000
ACPI_LV_FULL_TABLES         0x40000000
ACPI_LV_EVENTS              0x80000000
```

1.11.5 Examples

For example, drivers/acpi/acpica/evxfevnt.c contains this:

```
#define _COMPONENT          ACPI_EVENTS
...
ACPI_DEBUG_PRINT((ACPI_DB_INIT, "ACPI mode disabled\n"));
```

To turn on this message, set the ACPI_EVENTS bit in acpi.debug_layer and the ACPI_LV_INIT bit in acpi.debug_level. (The ACPI_DEBUG_PRINT statement uses ACPI_DB_INIT, which is a macro based on the ACPI_LV_INIT definition.)

Enable all AML "Debug" output (stores to the Debug object while interpreting AML) during boot:

```
acpi.debug_layer=0xffffffff acpi.debug_level=0x2
```

Enable all ACPI hardware-related messages:

```
acpi.debug_layer=0x2 acpi.debug_level=0xffffffff
```

Enable all ACPI_DB_INFO messages after boot:

```
# echo 0x4 > /sys/module/acpi/parameters/debug_level
```

Show all valid component values:

```
# cat /sys/module/acpi/parameters/debug_layer
```

1.12 The AML Debugger

Author
Lv Zheng <lv.zheng@intel.com>

This document describes the usage of the AML debugger embedded in the Linux kernel.

1.12.1 1. Build the debugger

The following kernel configuration items are required to enable the AML debugger interface from the Linux kernel:

```
CONFIG_ACPI_DEBUGGER=y
CONFIG_ACPI_DEBUGGER_USER=m
```

The userspace utilities can be built from the kernel source tree using the following commands:

```
$ cd tools
$ make acpi
```

The resultant userspace tool binary is then located at:

```
tools/power/acpi/acpidbg
```

It can be installed to system directories by running ”make install” (as a sufficiently privileged user).

1.12.2 2. Start the userspace debugger interface

After booting the kernel with the debugger built-in, the debugger can be started by using the following commands:

```
# mount -t debugfs none /sys/kernel/debug
# modprobe acpi_dbg
# tools/power/acpi/acpidbg
```

That spawns the interactive AML debugger environment where you can execute debugger commands.

The commands are documented in the ”ACPICA Overview and Programmer Reference” that can be downloaded from https://acpica.org/documentation

The detailed debugger commands reference is located in Chapter 12 ”ACPICA Debugger Reference”. The ”help” command can be used for a quick reference.

1.12.3 3. Stop the userspace debugger interface

The interactive debugger interface can be closed by pressing Ctrl+C or using the ”quit” or ”exit” commands. When finished, unload the module with:

```
# rmmod acpi_dbg
```

The module unloading may fail if there is an acpidbg instance running.

1.12.4 4. Run the debugger in a script

It may be useful to run the AML debugger in a test script. ”acpidbg” supports this in a special ”batch” mode. For example, the following command outputs the entire ACPI namespace:

```
# acpidbg -b "namespace"
```

1.13 APEI output format

APEI uses printk as hardware error reporting interface, the output format is as follow:

```
<error record> :=
APEI generic hardware error status
severity: <integer>, <severity string>
section: <integer>, severity: <integer>, <severity string>
flags: <integer>
<section flags strings>
fru_id: <uuid string>
fru_text: <string>
section_type: <section type string>
<section data>

<severity string>* := recoverable | fatal | corrected | info

<section flags strings># :=
[primary][, containment warning][, reset][, threshold exceeded]\
[, resource not accessible][, latent error]

<section type string> := generic processor error | memory error | \
PCIe error | unknown, <uuid string>

<section data> :=
<generic processor section data> | <memory section data> | \
<pcie section data> | <null>
```

```
<generic processor section data> :=
[processor_type: <integer>, <proc type string>]
[processor_isa: <integer>, <proc isa string>]
[error_type: <integer>
<proc error type strings>]
[operation: <integer>, <proc operation string>]
[flags: <integer>
<proc flags strings>]
[level: <integer>]
[version_info: <integer>]
[processor_id: <integer>]
[target_address: <integer>]
[requestor_id: <integer>]
[responder_id: <integer>]
[IP: <integer>]

<proc type string>* := IA32/X64 | IA64

<proc isa string>* := IA32 | IA64 | X64

<processor error type strings># :=
[cache error][, TLB error][, bus error][, micro-architectural error]

<proc operation string>* := unknown or generic | data read | data write | \
instruction execution

<proc flags strings># :=
[restartable][, precise IP][, overflow][, corrected]

<memory section data> :=
[error_status: <integer>]
[physical_address: <integer>]
[physical_address_mask: <integer>]
[node: <integer>]
[card: <integer>]
[module: <integer>]
[bank: <integer>]
[device: <integer>]
[row: <integer>]
[column: <integer>]
[bit_position: <integer>]
[requestor_id: <integer>]
[responder_id: <integer>]
[target_id: <integer>]
[error_type: <integer>, <mem error type string>]

<mem error type string>* :=
unknown | no error | single-bit ECC | multi-bit ECC | \
single-symbol chipkill ECC | multi-symbol chipkill ECC | master abort | \
target abort | parity error | watchdog timeout | invalid address | \
mirror Broken | memory sparing | scrub corrected error | \
scrub uncorrected error

<pcie section data> :=
[port_type: <integer>, <pcie port type string>]
[version: <integer>.<integer>]
[command: <integer>, status: <integer>]
[device_id: <integer>:<integer>:<integer>.<integer>
slot: <integer>
secondary_bus: <integer>
vendor_id: <integer>, device_id: <integer>
class_code: <integer>]
[serial number: <integer>, <integer>]
[bridge: secondary_status: <integer>, control: <integer>]
[aer_status: <integer>, aer_mask: <integer>
<aer status string>
[aer_uncor_severity: <integer>]
aer_layer=<aer layer string>, aer_agent=<aer agent string>
aer_tlp_header: <integer> <integer> <integer> <integer>]

<pcie port type string>* := PCIe end point | legacy PCI end point | \
unknown | unknown | root port | upstream switch port | \
downstream switch port | PCIe to PCI/PCI-X bridge | \
PCI/PCI-X to PCIe bridge | root complex integrated endpoint device | \
root complex event collector

if section severity is fatal or recoverable
<aer status string># :=
unknown | unknown | unknown | unknown | Data Link Protocol | \
unknown | unknown | unknown | unknown | unknown | unknown | unknown | \
Poisoned TLP | Flow Control Protocol | Completion Timeout | \
Completer Abort | Unexpected Completion | Receiver Overflow | \
Malformed TLP | ECRC | Unsupported Request
else
<aer status string># :=
Receiver Error | unknown | unknown | unknown | unknown | unknown | \
Bad TLP | Bad DLLP | RELAY_NUM Rollover | unknown | unknown | unknown | \
Replay Timer Timeout | Advisory Non-Fatal
fi

<aer layer string> :=
Physical Layer | Data Link Layer | Transaction Layer

<aer agent string> :=
Receiver ID | Requester ID | Completer ID | Transmitter ID
```

Where, [] designate corresponding content is optional

All <field string> description with * has the following format:

```
field: <integer>, <field string>
```

Where value of <integer> should be the position of "string" in <field string> description. Otherwise, <field string> will be "unknown".

All <field strings> description with # has the following format:

```
field: <integer>
<field strings>
```

Where each string in <fields strings> corresponding to one set bit of <integer>. The bit position is the position of "string" in <field strings> description.

For more detailed explanation of every field, please refer to UEFI specification version 2.3 or later, section Appendix N: Common Platform Error Record.

1.14 APEI Error INJection

EINJ provides a hardware error injection mechanism. It is very useful for debugging and testing APEI and RAS features in general.

You need to check whether your BIOS supports EINJ first. For that, look for early boot messages similar to this one:

```
ACPI: EINJ 0x000000007370A000 000150 (v01 INTEL           00000001 INTL␣
↪00000001)
```

which shows that the BIOS is exposing an EINJ table - it is the mechanism through which the injection is done.

Alternatively, look in /sys/firmware/acpi/tables for an "EINJ" file, which is a different representation of the same thing.

It doesn't necessarily mean that EINJ is not supported if those above don't exist: before you give up, go into BIOS setup to see if the BIOS has an option to enable error injection. Look for something called WHEA or similar. Often, you need to enable an ACPI5 support option prior, in order to see the APEI,EINJ,... functionality supported and exposed by the BIOS menu.

To use EINJ, make sure the following are options enabled in your kernel configuration:

```
CONFIG_DEBUG_FS
CONFIG_ACPI_APEI
CONFIG_ACPI_APEI_EINJ
```

The EINJ user interface is in <debugfs mount point>/apei/einj.

The following files belong to it:

- available_error_type

  This file shows which error types are supported:

Error Type Value	Error Description
0x00000001	Processor Correctable
0x00000002	Processor Uncorrectable non-fatal
0x00000004	Processor Uncorrectable fatal
0x00000008	Memory Correctable
0x00000010	Memory Uncorrectable non-fatal
0x00000020	Memory Uncorrectable fatal
0x00000040	PCI Express Correctable
0x00000080	PCI Express Uncorrectable non-fatal
0x00000100	PCI Express Uncorrectable fatal
0x00000200	Platform Correctable
0x00000400	Platform Uncorrectable non-fatal
0x00000800	Platform Uncorrectable fatal

The format of the file contents are as above, except present are only the available error types.

- error_type

  Set the value of the error type being injected. Possible error types are defined in the file available_error_type above.

- error_inject

  Write any integer to this file to trigger the error injection. Make sure you have specified all necessary error parameters, i.e. this write should be the last step when injecting errors.

- flags

  Present for kernel versions 3.13 and above. Used to specify which of param{1..4} are valid and should be used by the firmware during injection. Value is a bitmask as specified in ACPI5.0 spec for the SET_ERROR_TYPE_WITH_ADDRESS data structure:

  Bit 0
  Processor APIC field valid (see param3 below).

  Bit 1
  Memory address and mask valid (param1 and param2).

  Bit 2
  PCIe (seg,bus,dev,fn) valid (see param4 below).

  If set to zero, legacy behavior is mimicked where the type of injection specifies just one bit set, and param1 is multiplexed.

- param1

  This file is used to set the first error parameter value. Its effect depends on the error type specified in error_type. For example, if error type is memory related type, the param1 should be a valid physical memory address. [Unless "flag" is set - see above]

- param2

  Same use as param1 above. For example, if error type is of memory related type, then param2 should be a physical memory address mask. Linux requires page or narrower granularity, say, 0xfffffffffffff000.

- param3

  Used when the 0x1 bit is set in ”flags” to specify the APIC id
- param4 Used when the 0x4 bit is set in ”flags” to specify target PCIe device
- notrigger

  The error injection mechanism is a two-step process. First inject the error, then perform some actions to trigger it. Setting ”notrigger” to 1 skips the trigger phase, which *may* allow the user to cause the error in some other context by a simple access to the CPU, memory location, or device that is the target of the error injection. Whether this actually works depends on what operations the BIOS actually includes in the trigger phase.

BIOS versions based on the ACPI 4.0 specification have limited options in controlling where the errors are injected. Your BIOS may support an extension (enabled with the param_extension=1 module parameter, or boot command line einj.param_extension=1). This allows the address and mask for memory injections to be specified by the param1 and param2 files in apei/einj.

BIOS versions based on the ACPI 5.0 specification have more control over the target of the injection. For processor-related errors (type 0x1, 0x2 and 0x4), you can set flags to 0x3 (param3 for bit 0, and param1 and param2 for bit 1) so that you have more information added to the error signature being injected. The actual data passed is this:

```
memory_address = param1;
memory_address_range = param2;
apicid = param3;
pcie_sbdf = param4;
```

For memory errors (type 0x8, 0x10 and 0x20) the address is set using param1 with a mask in param2 (0x0 is equivalent to all ones). For PCI express errors (type 0x40, 0x80 and 0x100) the segment, bus, device and function are specified using param1:

```
 31     24 23    16 15    11 10      8  7        0
+-------------------------------------------------+
| segment |   bus  | device | function | reserved |
+-------------------------------------------------+
```

Anyway, you get the idea, if there's doubt just take a look at the code in drivers/acpi/apei/einj.c.

An ACPI 5.0 BIOS may also allow vendor-specific errors to be injected. In this case a file named vendor will contain identifying information from the BIOS that hopefully will allow an application wishing to use the vendor-specific extension to tell that they are running on a BIOS that supports it. All vendor extensions have the 0x80000000 bit set in error_type. A file vendor_flags controls the interpretation of param1 and param2 (1 = PROCESSOR, 2 = MEMORY, 4 = PCI). See your BIOS vendor documentation for details (and expect changes to this API if vendors creativity in using this feature expands beyond our expectations).

An error injection example:

```
# cd /sys/kernel/debug/apei/einj
# cat available_error_type            # See which errors can be injected
0x00000002    Processor Uncorrectable non-fatal
0x00000008    Memory Correctable
0x00000010    Memory Uncorrectable non-fatal
# echo 0x12345000 > param1            # Set memory address for injection
# echo 0xfffffffffffff000 > param2          # Mask - anywhere in this page
# echo 0x8 > error_type                     # Choose correctable memory error
# echo 1 > error_inject                     # Inject now
```

You should see something like this in dmesg:

```
[22715.830801] EDAC sbridge MC3: HANDLING MCE MEMORY ERROR
[22715.834759] EDAC sbridge MC3: CPU 0: Machine Check Event: 0 Bank 7:␣
↪8c00004000010090
[22715.834759] EDAC sbridge MC3: TSC 0
[22715.834759] EDAC sbridge MC3: ADDR 12345000 EDAC sbridge MC3: MISC 144780c86
[22715.834759] EDAC sbridge MC3: PROCESSOR 0:306e7 TIME 1422553404 SOCKET 0␣
↪APIC 0
[22716.616173] EDAC MC3: 1 CE memory read error on CPU_SrcID#0_Channel#0_DIMM
↪#0 (channel:0 slot:0 page:0x12345 offset:0x0 grain:32 syndrome:0x0 - ␣
↪area:DRAM err_code:0001:0090 socket:0 channel_mask:1 rank:0)
```

Special notes for injection into SGX enclaves:

There may be a separate BIOS setup option to enable SGX injection.

The injection process consists of setting some special memory controller trigger that will inject the error on the next write to the target address. But the h/w prevents any software outside of an SGX enclave from accessing enclave pages (even BIOS SMM mode).

The following sequence can be used:

1) Determine physical address of enclave page
2) Use "notrigger=1" mode to inject (this will setup the injection address, but will not actually inject)
3) Enter the enclave
4) Store data to the virtual address matching physical address from step 1
5) Execute CLFLUSH for that virtual address
6) Spin delay for 250ms
7) Read from the virtual address. This will trigger the error

For more information about EINJ, please refer to ACPI specification version 4.0, section 17.5 and ACPI 5.0, section 18.6.

1.15 _DSD Device Properties Related to GPIO

With the release of ACPI 5.1, the _DSD configuration object finally allows names to be given to GPIOs (and other things as well) returned by _CRS. Previously, we were only able to use an integer index to find the corresponding GPIO, which is pretty error prone (it depends on the _CRS output ordering, for example).

With _DSD we can now query GPIOs using a name instead of an integer index, like the ASL example below shows:

```
// Bluetooth device with reset and shutdown GPIOs
Device (BTH)
{
    Name (_HID, ...)

    Name (_CRS, ResourceTemplate ()
    {
        GpioIo (Exclusive, PullUp, 0, 0, IoRestrictionOutputOnly,
                "\\_SB.GPO0", 0, ResourceConsumer) { 15 }
        GpioIo (Exclusive, PullUp, 0, 0, IoRestrictionOutputOnly,
                "\\_SB.GPO0", 0, ResourceConsumer) { 27, 31 }
    })

    Name (_DSD, Package ()
    {
        ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
        Package ()
        {
            Package () { "reset-gpios", Package () { ^BTH, 1, 1, 0 } },
            Package () { "shutdown-gpios", Package () { ^BTH, 0, 0, 0 } },
        }
    })
}
```

The format of the supported GPIO property is:

```
Package () { "name", Package () { ref, index, pin, active_low }}
```

ref
The device that has _CRS containing GpioIo()/GpioInt() resources, typically this is the device itself (BTH in our case).

index
Index of the GpioIo()/GpioInt() resource in _CRS starting from zero.

pin
Pin in the GpioIo()/GpioInt() resource. Typically this is zero.

active_low
If 1, the GPIO is marked as active_low.

Since ACPI GpioIo() resource does not have a field saying whether it is active low or high, the ”active_low” argument can be used here. Setting it to 1 marks the GPIO as active low.

Note, active_low in _DSD does not make sense for GpioInt() resource and must be 0. GpioInt() resource has its own means of defining it.

In our Bluetooth example the ”reset-gpios” refers to the second GpioIo() resource, second pin in that resource with the GPIO number of 31.

The GpioIo() resource unfortunately doesn't explicitly provide an initial state of the output pin which driver should use during its initialization.

Linux tries to use common sense here and derives the state from the bias and polarity settings. The table below shows the expectations:

Pull Bias	Polarity	Requested...
Implicit		
Default	x	AS IS (assumed firmware configured it for us)
Explicit		
None	x	AS IS (assumed firmware configured it for us) with no Pull Bias
Up	x (no _DSD)	as high, assuming non-active
	Low	
	High	as high, assuming active
Down	x (no _DSD)	as low, assuming non-active
	High	
	Low	as low, assuming active

That said, for our above example the both GPIOs, since the bias setting is explicit and _DSD is present, will be treated as active with a high polarity and Linux will configure the pins in this state until a driver reprograms them differently.

It is possible to leave holes in the array of GPIOs. This is useful in cases like with SPI host controllers where some chip selects may be implemented as GPIOs and some as native signals. For example a SPI host controller can have chip selects 0 and 2 implemented as GPIOs and 1 as native:

```
Package () {
    "cs-gpios",
    Package () {
        ^GPIO, 19, 0, 0, // chip select 0: GPIO
        0,               // chip select 1: native signal
        ^GPIO, 20, 0, 0, // chip select 2: GPIO
    }
}
```

Note, that historically ACPI has no means of the GPIO polarity and thus the SPISerialBus() resource defines it on the per-chip basis. In order to avoid a chain of negations, the GPIO polarity is considered being Active High. Even for the cases when _DSD() is involved (see the example above) the GPIO CS polarity must be defined Active High to avoid ambiguity.

1.15.1 Other supported properties

Following Device Tree compatible device properties are also supported by _DSD device properties for GPIO controllers:

- gpio-hog
- output-high
- output-low
- input
- line-name

Example:

```
Name (_DSD, Package () {
    // _DSD Hierarchical Properties Extension UUID
    ToUUID("dbb8e3e6-5886-4ba6-8795-1319f52a966b"),
    Package () {
        Package () { "hog-gpio8", "G8PU" }
    }
})

Name (G8PU, Package () {
    ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
    Package () {
        Package () { "gpio-hog", 1 },
        Package () { "gpios", Package () { 8, 0 } },
        Package () { "output-high", 1 },
        Package () { "line-name", "gpio8-pullup" },
    }
})
```

- gpio-line-names

The `gpio-line-names` declaration is a list of strings (”names”), which describes each line/pin of a GPIO controller/expander. This list, contained in a package, must be inserted inside the GPIO controller declaration of an ACPI table (typically inside the DSDT). The `gpio-line-names` list must respect the following rules (see also the examples):

- the first name in the list corresponds with the first line/pin of the GPIO controller/expander
- the names inside the list must be consecutive (no ”holes” are permitted)
- the list can be incomplete and can end before the last GPIO line: in other words, it is not mandatory to fill all the GPIO lines
- empty names are allowed (two quotation marks `""` correspond to an empty name)
- names inside one GPIO controller/expander must be unique

Example of a GPIO controller of 16 lines, with an incomplete list with two empty names:

```
Package () {
    "gpio-line-names",
    Package () {
        "pin_0",
        "pin_1",
        "",
        "",
        "pin_3",
        "pin_4_push_button",
    }
}
```

At runtime, the above declaration produces the following result (using the ”libgpiod” tools):

```
root@debian:~# gpioinfo gpiochip4
gpiochip4 - 16 lines:
        line   0:      "pin_0"       unused   input  active-high
        line   1:      "pin_1"       unused   input  active-high
        line   2:      unnamed       unused   input  active-high
        line   3:      unnamed       unused   input  active-high
        line   4:      "pin_3"       unused   input  active-high
        line   5: "pin_4_push_button" unused input active-high
        line   6:      unnamed       unused   input  active-high
        line   7       unnamed       unused   input  active-high
        line   8:      unnamed       unused   input  active-high
        line   9:      unnamed       unused   input  active-high
        line  10:      unnamed       unused   input  active-high
        line  11:      unnamed       unused   input  active-high
        line  12:      unnamed       unused   input  active-high
        line  13:      unnamed       unused   input  active-high
        line  14:      unnamed       unused   input  active-high
        line  15:      unnamed       unused   input  active-high
root@debian:~# gpiofind pin_4_push_button
gpiochip4 5
root@debian:~#
```

Another example:

```
Package () {
    "gpio-line-names",
    Package () {
        "SPI0_CS_N", "EXP2_INT", "MUX6_IO", "UART0_RXD",
        "MUX7_IO", "LVL_C_A1", "MUX0_IO", "SPI1_MISO",
    }
}
```

See Documentation/devicetree/bindings/gpio/gpio.txt for more information about these properties.

1.15.2 ACPI GPIO Mappings Provided by Drivers

There are systems in which the ACPI tables do not contain _DSD but provide _CRS with GpioIo()/GpioInt() resources and device drivers still need to work with them.

In those cases ACPI device identification objects, _HID, _CID, _CLS, _SUB, _HRV, available to the driver can be used to identify the device and that is supposed to be sufficient to determine the meaning and purpose of all of the GPIO lines listed by the GpioIo()/GpioInt() resources returned by _CRS. In other words, the driver is supposed to know what to use the GpioIo()/GpioInt() resources for once it has identified the device. Having done that, it can simply assign names to the GPIO lines it is going to use and provide the GPIO subsystem with a mapping between those names and the ACPI GPIO resources corresponding to them.

To do that, the driver needs to define a mapping table as a NULL-terminated array of struct acpi_gpio_mapping objects that each contains a name, a pointer to an array of line data (struct acpi_gpio_params) objects and the size of that array. Each struct acpi_gpio_params object consists of three fields, crs_entry_index, line_index, active_low, representing the index of the target GpioIo()/GpioInt() resource in _CRS starting from zero, the index of the target line in that resource starting from zero, and the active-low flag for that line, respectively, in analogy with the _DSD GPIO property format specified above.

For the example Bluetooth device discussed previously the data structures in question would look like this:

```
static const struct acpi_gpio_params reset_gpio = { 1, 1, false };
static const struct acpi_gpio_params shutdown_gpio = { 0, 0, false };

static const struct acpi_gpio_mapping bluetooth_acpi_gpios[] = {
  { "reset-gpios", &reset_gpio, 1 },
  { "shutdown-gpios", &shutdown_gpio, 1 },
  { }
};
```

Next, the mapping table needs to be passed as the second argument to acpi_dev_add_driver_gpios() or its managed analogue that will register it with the ACPI device object pointed to by its first argument. That should be done in the driver's .probe() routine. On removal, the driver should unregister its GPIO mapping table by calling acpi_dev_remove_driver_gpios() on the ACPI device object where that table was previously registered.

1.15.3 Using the _CRS fallback

If a device does not have _DSD or the driver does not create ACPI GPIO mapping, the Linux GPIO framework refuses to return any GPIOs. This is because the driver does not know what it actually gets. For example if we have a device like below:

```
Device (BTH)
{
    Name (_HID, ...)

    Name (_CRS, ResourceTemplate () {
        GpioIo (Exclusive, PullNone, 0, 0, IoRestrictionNone,
                "\\_SB.GPO0", 0, ResourceConsumer) { 15 }
        GpioIo (Exclusive, PullNone, 0, 0, IoRestrictionNone,
                "\\_SB.GPO0", 0, ResourceConsumer) { 27 }
    })
}
```

The driver might expect to get the right GPIO when it does:

```
desc = gpiod_get(dev, "reset", GPIOD_OUT_LOW);
if (IS_ERR(desc))
      ...error handling...
```

but since there is no way to know the mapping between ”reset” and the GpioIo() in _CRS desc will hold ERR_PTR(-ENOENT).

The driver author can solve this by passing the mapping explicitly (this is the recommended way and it's documented in the above chapter).

The ACPI GPIO mapping tables should not contaminate drivers that are not knowing about which exact device they are servicing on. It implies that the ACPI GPIO mapping tables are hardly linked to an ACPI ID and certain objects, as listed in the above chapter, of the device in question.

1.15.4 Getting GPIO descriptor

There are two main approaches to get GPIO resource from ACPI:

```
desc = gpiod_get(dev, connection_id, flags);
desc = gpiod_get_index(dev, connection_id, index, flags);
```

We may consider two different cases here, i.e. when connection ID is provided and otherwise.

Case 1:

```
desc = gpiod_get(dev, "non-null-connection-id", flags);
desc = gpiod_get_index(dev, "non-null-connection-id", index, flags);
```

Case 2:

```
desc = gpiod_get(dev, NULL, flags);
desc = gpiod_get_index(dev, NULL, index, flags);
```

Case 1 assumes that corresponding ACPI device description must have defined device properties and will prevent to getting any GPIO resources otherwise.

Case 2 explicitly tells GPIO core to look for resources in _CRS.

Be aware that gpiod_get_index() in cases 1 and 2, assuming that there are two versions of ACPI device description provided and no mapping is present in the driver, will return different resources. That's why a certain driver has to handle them carefully as explained in the previous chapter.

1.16 ACPI I2C Muxes

Describing an I2C device hierarchy that includes I2C muxes requires an ACPI Device () scope per mux channel.

Consider this topology:

```
+------+   +------+
| SMB1 |-->| MUX0 |--CH00--> i2c client A (0x50)
|      |   | 0x70 |--CH01--> i2c client B (0x50)
+------+   +------+
```

which corresponds to the following ASL:

```
Device (SMB1)
{
    Name (_HID, ...)
    Device (MUX0)
    {
        Name (_HID, ...)
        Name (_CRS, ResourceTemplate () {
            I2cSerialBus (0x70, ControllerInitiated, I2C_SPEED,
                        AddressingMode7Bit, "^SMB1", 0x00,
                        ResourceConsumer,,)
        }

        Device (CH00)
        {
            Name (_ADR, 0)

            Device (CLIA)
            {
                Name (_HID, ...)
                Name (_CRS, ResourceTemplate () {
                    I2cSerialBus (0x50, ControllerInitiated, I2C_SPEED,
                                AddressingMode7Bit, "^CH00", 0x00,
                                ResourceConsumer,,)
                }
            }
        }

        Device (CH01)
        {
            Name (_ADR, 1)

            Device (CLIB)
            {
                Name (_HID, ...)
                Name (_CRS, ResourceTemplate () {
                    I2cSerialBus (0x50, ControllerInitiated, I2C_SPEED,
                                AddressingMode7Bit, "^CH01", 0x00,
                                ResourceConsumer,,)
                }
            }
        }
    }
}
```

1.17 Special Usage Model of the ACPI Control Method Lid Device

Copyright
© 2016, Intel Corporation

Author
Lv Zheng <lv.zheng@intel.com>

1.17.1 Abstract

Platforms containing lids convey lid state (open/close) to OSPMs using a control method lid device. To implement this, the AML tables issue Notify(lid_device, 0x80) to notify the OSPMs whenever the lid state has changed. The _LID control method for the lid device must be implemented to report the "current" state of the lid as either "opened" or "closed".

For most platforms, both the _LID method and the lid notifications are reliable. However, there are exceptions. In order to work with these exceptional buggy platforms, special restrictions and exceptions should be taken into account. This document describes the restrictions and the exceptions of the Linux ACPI lid device driver.

1.17.2 Restrictions of the returning value of the _LID control method

The _LID control method is described to return the "current" lid state. However the word of "current" has ambiguity, some buggy AML tables return the lid state upon the last lid notification instead of returning the lid state upon the last _LID evaluation. There won't be difference when the _LID control method is evaluated during the runtime, the problem is its initial returning value. When the AML tables implement this control method with cached value, the initial returning value is likely not reliable. There are platforms always return "closed" as initial lid state.

1.17.3 Restrictions of the lid state change notifications

There are buggy AML tables never notifying when the lid device state is changed to "opened". Thus the "opened" notification is not guaranteed. But it is guaranteed that the AML tables always notify "closed" when the lid state is changed to "closed". The "closed" notification is normally used to trigger some system power saving operations on Windows. Since it is fully tested, it is reliable from all AML tables.

1.17.4 Exceptions for the userspace users of the ACPI lid device driver

The ACPI button driver exports the lid state to the userspace via the following file:

```
/proc/acpi/button/lid/LID0/state
```

This file actually calls the _LID control method described above. And given the previous explanation, it is not reliable enough on some platforms. So it is advised for the userspace program to not to solely rely on this file to determine the actual lid state.

The ACPI button driver emits the following input event to the userspace:

- SW_LID

The ACPI lid device driver is implemented to try to deliver the platform triggered events to the userspace. However, given the fact that the buggy firmware cannot make sure "opened"/"closed" events are paired, the ACPI button driver uses the following 3 modes in order not to trigger issues.

If the userspace hasn't been prepared to ignore the unreliable "opened" events and the unreliable initial state notification, Linux users can use the following kernel parameters to handle the possible issues:

A. button.lid_init_state=method: When this option is specified, the ACPI button driver reports the initial lid state using the returning value of the _LID control method and whether the ”opened”/”closed” events are paired fully relies on the firmware implementation.

   This option can be used to fix some platforms where the returning value of the _LID control method is reliable but the initial lid state notification is missing.

   This option is the default behavior during the period the userspace isn't ready to handle the buggy AML tables.

B. button.lid_init_state=open: When this option is specified, the ACPI button driver always reports the initial lid state as ”opened” and whether the ”opened”/”closed” events are paired fully relies on the firmware implementation.

   This may fix some platforms where the returning value of the _LID control method is not reliable and the initial lid state notification is missing.

If the userspace has been prepared to ignore the unreliable ”opened” events and the unreliable initial state notification, Linux users should always use the following kernel parameter:

C. button.lid_init_state=ignore: When this option is specified, the ACPI button driver never reports the initial lid state and there is a compensation mechanism implemented to ensure that the reliable ”closed” notifications can always be delivered to the userspace by always pairing ”closed” input events with complement ”opened” input events. But there is still no guarantee that the ”opened” notifications can be delivered to the userspace when the lid is actually opens given that some AML tables do not send ”opened” notifications reliably.

   In this mode, if everything is correctly implemented by the platform firmware, the old userspace programs should still work. Otherwise, the new userspace programs are required to work with the ACPI button driver. This option will be the default behavior after the userspace is ready to handle the buggy AML tables.

1.18 Low Power Idle Table (LPIT)

To enumerate platform Low Power Idle states, Intel platforms are using “Low Power Idle Table” (LPIT). More details about this table can be downloaded from: https://www.uefi.org/sites/default/files/resources/Intel_ACPI_Low_Power_S0_Idle.pdf

Residencies for each low power state can be read via FFH (Function fixed hardware) or a memory mapped interface.

On platforms supporting S0ix sleep states, there can be two types of residencies:

- CPU PKG C10 (Read via FFH interface)
- Platform Controller Hub (PCH) SLP_S0 (Read via memory mapped interface)

The following attributes are added dynamically to the cpuidle sysfs attribute group:

```
/sys/devices/system/cpu/cpuidle/low_power_idle_cpu_residency_us
/sys/devices/system/cpu/cpuidle/low_power_idle_system_residency_us
```

The ”low_power_idle_cpu_residency_us” attribute shows time spent by the CPU package in PKG C10

The ”low_power_idle_system_residency_us” attribute shows SLP_S0 residency, or system time spent with the SLP_S0# signal asserted. This is the lowest possible system power state, achieved only when CPU is in PKG C10 and all functional blocks in PCH are in a low power state.

1.19 ACPI video extensions

This driver implement the ACPI Extensions For Display Adapters for integrated graphics devices on motherboard, as specified in ACPI 2.0 Specification, Appendix B, allowing to perform some basic control like defining the video POST device, retrieving EDID information or to setup a video output, etc. Note that this is an ref. implementation only. It may or may not work for your integrated video device.

The ACPI video driver does 3 things regarding backlight control.

1.19.1 Export a sysfs interface for user space to control backlight level

If the ACPI table has a video device, and acpi_backlight=vendor kernel command line is not present, the driver will register a backlight device and set the required backlight operation structure for it for the sysfs interface control. For every registered class device, there will be a directory named acpi_videoX under /sys/class/backlight.

The backlight sysfs interface has a standard definition here: Documentation/ABI/stable/sysfs-class-backlight.

And what ACPI video driver does is:

actual_brightness:
on read, control method _BQC will be evaluated to get the brightness level the firmware thinks it is at;

bl_power:
not implemented, will set the current brightness instead;

brightness:
on write, control method _BCM will run to set the requested brightness level;

max_brightness:
Derived from the _BCL package(see below);

type:
firmware

Note that ACPI video backlight driver will always use index for brightness, actual_brightness and max_brightness. So if we have the following _BCL package:

```
Method (_BCL, 0, NotSerialized)
{
        Return (Package (0x0C)
        {
                0x64,
                0x32,
                0x0A,
                0x14,
                0x1E,
                0x28,
                0x32,
                0x3C,
                0x46,
                0x50,
                0x5A,
                0x64
        })
}
```

The first two levels are for when laptop are on AC or on battery and are not used by Linux currently. The remaining 10 levels are supported levels that we can choose from. The applicable index values are from 0 (that corresponds to the 0x0A brightness value) to 9 (that corresponds to the 0x64 brightness value) inclusive. Each of those index values is regarded as a "brightness level" indicator. Thus from the user space perspective the range of available brightness levels is from 0 to 9 (max_brightness) inclusive.

1.19.2 Notify user space about hotkey event

There are generally two cases for hotkey event reporting:

i) For some laptops, when user presses the hotkey, a scancode will be generated and sent to user space through the input device created by the keyboard driver as a key type input event, with proper remap, the following key code will appear to user space:

```
EV_KEY, KEY_BRIGHTNESSUP
EV_KEY, KEY_BRIGHTNESSDOWN
etc.
```

For this case, ACPI video driver does not need to do anything(actually, it doesn't even know this happened).

ii) For some laptops, the press of the hotkey will not generate the scancode, instead, firmware will notify the video device ACPI node about the event. The event value is defined in the ACPI spec. ACPI video driver will generate an key type input event according to the notify value it received and send the event to user space through the input device it created:

event	keycode
0x86	KEY_BRIGHTNESSUP
0x87	KEY_BRIGHTNESSDOWN
etc.	

so this would lead to the same effect as case i) now.

Once user space tool receives this event, it can modify the backlight level through the sysfs interface.

1.19.3 Change backlight level in the kernel

This works for machines covered by case ii) in Section 2. Once the driver received a notification, it will set the backlight level accordingly. This does not affect the sending of event to user space, they are always sent to user space regardless of whether or not the video module controls the backlight level directly. This behaviour can be controlled through the brightness_switch_enabled module parameter as documented in admin-guide/kernel-parameters.rst. It is recommended to disable this behaviour once a GUI environment starts up and wants to have full control of the backlight level.

1.20 Probing devices in other D states than 0

1.20.1 Introduction

In some cases it may be preferred to leave certain devices powered off for the entire system bootup if powering on these devices has adverse side effects, beyond just powering on the said device.

1.20.2 How it works

The _DSC (Device State for Configuration) object that evaluates to an integer may be used to tell Linux the highest allowed D state for a device during probe. The support for _DSC requires support from the kernel bus type if the bus driver normally sets the device in D0 state for probe.

The downside of using _DSC is that as the device is not powered on, even if there's a problem with the device, the driver likely probes just fine but the first user will find out the device doesn't work, instead of a failure at probe time. This feature should thus be used sparingly.

I^{2}C

If an I^{2}C driver indicates its support for this by setting the I2C_DRV_ACPI_WAIVE_D0_PROBE flag in struct i2c_driver.flags field and the _DSC object evaluates to integer higher than the D state of the device, the device will not be powered on (put in D0 state) for probe.

D states

The D states and thus also the allowed values for _DSC are listed below. Refer to [1] for more information on device power states.

```
Number  State   Description
0       D0      Device fully powered on
1       D1
2       D2
3       D3hot
4       D3cold  Off
```

1.20.3 References

[1] https://uefi.org/specifications/ACPI/6.4/02_Definition_of_Terms/Definition_of_Terms.html#device-power-state-definitions

1.20.4 Example

An ASL example describing an ACPI device using _DSC object to tell Operating System the device should remain powered off during probe looks like this. Some objects not relevant from the example point of view have been omitted.

```
Device (CAM0)
{
        Name (_HID, "SONY319A")
        Name (_UID, Zero)
        Name (_CRS, ResourceTemplate ()
        {
                I2cSerialBus(0x0020, ControllerInitiated, 0x00061A80,
                             AddressingMode7Bit, "\\_SB.PCI0.I2C0",
                             0x00, ResourceConsumer)
        })
        Method (_DSC, 0, NotSerialized)
        {
                Return (0x4)
        }
}
```

1.21 Intel INT3496 ACPI device extcon driver documentation

The Intel INT3496 ACPI device extcon driver is a driver for ACPI devices with an acpi-id of INT3496, such as found for example on Intel Baytrail and Cherrytrail tablets.

This ACPI device describes how the OS can read the id-pin of the devices' USB-otg port, as well as how it optionally can enable Vbus output on the otg port and how it can optionally control the muxing of the data pins between an USB host and an USB peripheral controller.

The ACPI devices exposes this functionality by returning an array with up to 3 gpio descriptors from its ACPI _CRS (Current Resource Settings) call:

Index 0	The input gpio for the id-pin, this is always present and valid
Index 1	The output gpio for enabling Vbus output from the device to the otg port, write 1 to enable the Vbus output (this gpio descriptor may be absent or invalid)
Index 2	The output gpio for muxing of the data pins between the USB host and the USB peripheral controller, write 1 to mux to the peripheral controller

There is a mapping between indices and GPIO connection IDs as follows

id	index 0
vbus	index 1
mux	index 2

1.22 Intel North Mux-Agent

1.22.1 Introduction

North Mux-Agent is a function of the Intel PMC firmware that is supported on most Intel based platforms that have the PMC microcontroller. It's used for configuring the various USB Multiplexer/DeMultiplexers on the system. The platforms that allow the mux-agent to be configured from the operating system have an ACPI device object (node) with HID ”INTC105C” that represents it.

The North Mux-Agent (aka. Intel PMC Mux Control, or just mux-agent) driver communicates with the PMC microcontroller by using the PMC IPC method (drivers/platform/x86/intel_scu_ipc.c). The driver registers with the USB Type-C Mux Class which allows the USB Type-C Controller and Interface drivers to configure the cable plug orientation and mode (with Alternate Modes). The driver also registers with the USB Role Class in order to support both USB Host and Device modes. The driver is located here: drivers/usb/typec/mux/intel_pmc_mux.c.

1.22.2 Port nodes

General

For every USB Type-C connector under the mux-agent control on the system, there is a separate child node under the PMC mux-agent device node. Those nodes do not represent the actual connectors, but instead the ”channels” in the mux-agent that are associated with the connectors:

```
Scope (_SB.PCI0.PMC.MUX)
{
    Device (CH0)
    {
        Name (_ADR, 0)
    }

    Device (CH1)
    {
        Name (_ADR, 1)
    }
}
```

_PLD (Physical Location of Device)

The optional _PLD object can be used with the port (the channel) nodes. If _PLD is supplied, it should match the connector node _PLD:

```
Scope (_SB.PCI0.PMC.MUX)
{
    Device (CH0)
    {
        Name (_ADR, 0)
        Method (_PLD, 0, NotSerialized)
        {
            /* Consider this as pseudocode. */
            Return (\_SB.USBC.CON0._PLD())
        }
    }
}
```

Mux-agent specific _DSD Device Properties

Port Numbers

In order to configure the muxes behind a USB Type-C connector, the PMC firmware needs to know the USB2 port and the USB3 port that is associated with the connector. The driver extracts the correct port numbers by reading specific _DSD device properties named ”usb2-port-number” and ”usb3-port-number”. These properties have integer value that means the port index. The port index number is 1's based, and value 0 is illegal. The driver uses the numbers extracted from these device properties as-is when sending the mux-agent specific messages to the PMC:

```
Name (_DSD, Package () {
    ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
    Package() {
        Package () {"usb2-port-number", 6},
        Package () {"usb3-port-number", 3},
    },
})
```

Orientation

Depending on the platform, the data and SBU lines coming from the connector may be ”fixed” from the mux-agent's point of view, which means the mux-agent driver should not configure them according to the cable plug orientation. This can happen for example if a retimer on the platform handles the cable plug orientation. The driver uses a specific device properties ”sbu-orientation” (SBU) and ”hsl-orientation” (data) to know if those lines are ”fixed”, and to which orientation. The value that these properties have is a string value, and it can be one that is defined for the USB Type-C connector orientation: ”normal” or ”reversed”:

```
Name (_DSD, Package () {
    ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
    Package() {
        Package () {"sbu-orientation", "normal"},
        Package () {"hsl-orientation", "normal"},
    },
})
```

1.22.3 Example ASL

The following ASL is an example that shows the mux-agent node, and two connectors under its control:

```
Scope (_SB.PCI0.PMC)
{
    Device (MUX)
    {
        Name (_HID, "INTC105C")

        Device (CH0)
        {
            Name (_ADR, 0)

            Name (_DSD, Package () {
                ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
                Package() {
                    Package () {"usb2-port-number", 6},
                    Package () {"usb3-port-number", 3},
                    Package () {"sbu-orientation", "normal"},
                    Package () {"hsl-orientation", "normal"},
                },
            })
        }

        Device (CH1)
        {
            Name (_ADR, 1)

            Name (_DSD, Package () {
                ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
                Package() {
                    Package () {"usb2-port-number", 5},
                    Package () {"usb3-port-number", 2},
                    Package () {"sbu-orientation", "normal"},
                    Package () {"hsl-orientation", "normal"},
                },
            })
        }
    }
}
```

1.23 Chrome OS ACPI Device

Hardware functionality specific to Chrome OS is exposed through a Chrome OS ACPI device. The plug and play ID of a Chrome OS ACPI device is GGL0001 and the hardware ID is GOOG0016. The following ACPI objects are supported:

Table 1: Supported ACPI Objects

Object	Description
CHSW	Chrome OS switch positions
HWID	Chrome OS hardware ID
FWID	Chrome OS firmware version
FRID	Chrome OS read-only firmware version
BINF	Chrome OS boot information
GPIO	Chrome OS GPIO assignments
VBNV	Chrome OS NVRAM locations
VDTA	Chrome OS verified boot data
FMAP	Chrome OS flashmap base address
MLST	Chrome OS method list

1.23.1 CHSW (Chrome OS switch positions)

This control method returns the switch positions for Chrome OS specific hardware switches.

Arguments:

None

Result code:

An integer containing the switch positions as bitfields:

0x00000002	Recovery button was pressed when x86 firmware booted.
0x00000004	Recovery button was pressed when EC firmware booted. (required if EC EEPROM is rewritable; otherwise optional)
0x00000020	Developer switch was enabled when x86 firmware booted.
0x00000200	Firmware write protection was disabled when x86 firmware booted. (required if firmware write protection is controlled through x86 BIOS; otherwise optional)

All other bits are reserved and should be set to 0.

1.23.2 HWID (Chrome OS hardware ID)

This control method returns the hardware ID for the Chromebook.

Arguments:

None

Result code:

A null-terminated ASCII string containing the hardware ID from the Model-Specific Data area of EEPROM.

Note that the hardware ID can be up to 256 characters long, including the terminating null.

1.23.3 FWID (Chrome OS firmware version)

This control method returns the firmware version for the rewritable portion of the main processor firmware.

Arguments:

None

Result code:

A null-terminated ASCII string containing the complete firmware version for the rewritable portion of the main processor firmware.

1.23.4 FRID (Chrome OS read-only firmware version)

This control method returns the firmware version for the read-only portion of the main processor firmware.

Arguments:

None

Result code:

A null-terminated ASCII string containing the complete firmware version for the read-only (bootstrap + recovery) portion of the main processor firmware.

1.23.5 BINF (Chrome OS boot information)

This control method returns information about the current boot.

Arguments:

None

Result code:

```
Package {
        Reserved1
        Reserved2
        Active EC Firmware
        Active Main Firmware Type
        Reserved5
}
```

Field	Format	Description
Reserved1	DWORD	Set to 256 (0x100). This indicates this field is no longer used.
Reserved2	DWORD	Set to 256 (0x100). This indicates this field is no longer used.
Active EC firmware	DWORD	The EC firmware which was used during boot. • 0 - Read-only (recovery) firmware • 1 - Rewritable firmware. Set to 0 if EC firmware is always read-only.
Active Main Firmware Type	DWORD	The main firmware type which was used during boot. • 0 - Recovery • 1 - Normal • 2 - Developer • 3 - netboot (factory installation only) Other values are reserved.
Reserved5	DWORD	Set to 256 (0x100). This indicates this field is no longer used.

1.23.6 GPIO (Chrome OS GPIO assignments)

This control method returns information about Chrome OS specific GPIO assignments for Chrome OS hardware, so the kernel can directly control that hardware.

Arguments:

None

Result code:

```
Package {
        Package {
                // First GPIO assignment
                Signal Type        //DWORD
                Attributes         //DWORD
                Controller Offset  //DWORD
                Controller Name    //ASCIIZ
        },
        ...
        Package {
                // Last GPIO assignment
                Signal Type        //DWORD
                Attributes         //DWORD
                Controller Offset  //DWORD
                Controller Name    //ASCIIZ
        }
}
```

Where ASCIIZ means a null-terminated ASCII string.

Field	Format	Description
Signal Type	DWORD	Type of GPIO signal • 0x00000001 - Recovery button • 0x00000002 - Developer mode switch • 0x00000003 - Firmware write protection switch • 0x00000100 - Debug header GPIO 0 • … • 0x000001FF - Debug header GPIO 255 Other values are reserved.
Attributes	DWORD	Signal attributes as bitfields: • 0x00000001 - Signal is active-high (for button, a GPIO value of 1 means the button is pressed; for switches, a GPIO value of 1 means the switch is enabled). If this bit is 0, the signal is active low. Set to 0 for debug header GPIOs.
Controller Offset	DWORD	GPIO number on the specified controller.
Controller Name	ASCIIZ	Name of the controller for the GPIO. Currently supported names: ”NM10” - Intel NM10 chip

1.23.7 VBNV (Chrome OS NVRAM locations)

This control method returns information about the NVRAM (CMOS) locations used to communicate with the BIOS.

Arguments:

None

Result code:

```
Package {
        NV Storage Block Offset  //DWORD
        NV Storage Block Size    //DWORD
}
```

Field	Format	Description
NV Storage Block Offset	DWORD	Offset in CMOS bank 0 of the verified boot non-volatile storage block, counting from the first writable CMOS byte (that is, offset=0 is the byte following the 14 bytes of clock data).
NV Storage Block Size	DWORD	Size in bytes of the verified boot non-volatile storage block.

1.23.8 FMAP (Chrome OS flashmap address)

This control method returns the physical memory address of the start of the main processor firmware flashmap.

Arguments:

None

NoneResult code:

A DWORD containing the physical memory address of the start of the main processor firmware flashmap.

1.23.9 VDTA (Chrome OS verified boot data)

This control method returns the verified boot data block shared between the firmware verification step and the kernel verification step.

Arguments:

None

Result code:

A buffer containing the verified boot data block.

1.23.10 MECK (Management Engine Checksum)

This control method returns the SHA-1 or SHA-256 hash that is read out of the Management Engine extended registers during boot. The hash is exported via ACPI so the OS can verify that the ME firmware has not changed. If Management Engine is not present, or if the firmware was unable to read the extended registers, this buffer can be zero.

Arguments:

None

Result code:

A buffer containing the ME hash.

1.23.11 MLST (Chrome OS method list)

This control method returns a list of the other control methods supported by the Chrome OS hardware device.

Arguments:

None

Result code:

A package containing a list of null-terminated ASCII strings, one for each control method supported by the Chrome OS hardware device, not including the MLST method itself. For this version of the specification, the result is:

```
Package {
        "CHSW",
        "FWID",
        "HWID",
        "FRID",
        "BINF",
        "GPIO",
        "VBNV",
        "FMAP",
        "VDTA",
        "MECK"
}
```