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_device. Note that standard UARTs are not busses so there is no struct uart_device, although some of them may be represented by sturct 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.

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.

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.

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 (2) {"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.

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 than with 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...
```

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);
```

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).

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 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 Documentation/firmware-guide/acpi/gpio-properties.rst for more information about the _DSD binding related to GPIOs.

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.

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 Documentation/firmware-guide/acpi/DSD-properties-rules.rst for more information.

PCI hierarchy representation

Sometimes 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 us assume to 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 is necessary 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.

References

[1] Documentation/firmware-guide/acpi/gpio-properties.rst

[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