# _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 | x | AS IS (assumed firmware configured for us) |
| Explicit | x (no _DSD) | as Pull Bias (Up == High, Down == Low), assuming non-active (Polarity = !Pull Bias) |
| Down | Low | as low, assuming active |
| Down | High | as low, assuming non-active |
| Up | Low | as high, assuming non-active |
| Up | High | as high, 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.

## 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.

# 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.

# 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.

## 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.