1. **How to write device tree and device driver for an I2C based pressure sensor connected to a Raspberry Pi 4 Model B board?**

Great! Let's break this down step-by-step. You'll be writing:

1. A **Device Tree Overlay** to tell the Linux kernel about your I2C pressure sensor.
2. A **Device Driver** (likely a kernel module) that knows how to communicate with the sensor over I2C.

We'll assume your sensor uses a standard I2C interface and has a known I2C address (e.g., 0x76 for sensors like the BMP280). Here's how to get started:

### 📦 Part 1: Device Tree Overlay for I2C Pressure Sensor

Raspberry Pi 4 has multiple I2C buses. The common one is /dev/i2c-1 (also known as i2c1).

#### 1.1 Find the Right I2C Bus

Raspberry Pi 4 has:

* I2C0: usually reserved for internal functions (e.g., EEPROM).
* I2C1: user-available on GPIO 2 (SDA) and GPIO 3 (SCL).

So we will focus on i2c1.

#### 1.2 Create an Overlay (my\_pressure\_sensor-overlay.dts)

dts

/dts-v1/;

/plugin/;

/ {

compatible = "brcm,bcm2835";

fragment@0 {

target = <&i2c1>;

\_\_overlay\_\_ {

#address-cells = <1>;

#size-cells = <0>;

pressure\_sensor@76 {

compatible = "myvendor,mypressure"; // You'll match this in your driver

reg = <0x76>; // I2C address

status = "okay";

};

};

};

};

#### 1.3 Compile the Overlay

bash

dtc -@ -I dts -O dtb -o my\_pressure\_sensor.dtbo my\_pressure\_sensor-overlay.dts

Then copy it to /boot/overlays/ and enable it by adding to /boot/config.txt:

bash

dtoverlay=my\_pressure\_sensor

Reboot the Pi afterward.

### 🧠 Part 2: Writing a Simple Linux I2C Device Driver

#### 2.1 Skeleton Driver Structure

Create a kernel module that registers with the I2C core.

c

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/i2c.h>

#define DRIVER\_NAME "mypressure"

static int pressure\_probe(struct i2c\_client \*client, const struct i2c\_device\_id \*id)

{

dev\_info(&client->dev, "Pressure sensor probed at 0x%02x\n", client->addr);

// Initialize sensor here (e.g., read chip ID, setup config)

return 0;

}

static int pressure\_remove(struct i2c\_client \*client)

{

dev\_info(&client->dev, "Pressure sensor removed\n");

return 0;

}

static const struct i2c\_device\_id pressure\_id[] = {

{ DRIVER\_NAME, 0 },

{ }

};

MODULE\_DEVICE\_TABLE(i2c, pressure\_id);

static const struct of\_device\_id pressure\_of\_match[] = {

{ .compatible = "myvendor,mypressure" },

{ }

};

MODULE\_DEVICE\_TABLE(of, pressure\_of\_match);

static struct i2c\_driver pressure\_driver = {

.driver = {

.name = DRIVER\_NAME,

.of\_match\_table = pressure\_of\_match,

},

.probe = pressure\_probe,

.remove = pressure\_remove,

.id\_table = pressure\_id,

};

module\_i2c\_driver(pressure\_driver);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("You");

MODULE\_DESCRIPTION("I2C Pressure Sensor Driver");

#### 2.2 Build the Driver

Create a Makefile:

make

obj-m += pressure\_sensor.o

all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

Build and insert:

bash

make

sudo insmod pressure\_sensor.ko

dmesg | grep pressure

### 🧪 Test and Extend

You can extend the driver to:

* Read pressure/temperature values.
* Expose them via sysfs, input subsystem, or iio.

Would you like an example of reading a register (like a temperature or pressure value) from the sensor in the driver? And do you already know what pressure sensor model you're using (e.g., BMP280, LPS22HB, etc.)?

1. **Please explain:**

#address-cells = <1>;

#size-cells = <0>;

Great question — these two lines are fundamental when defining child nodes in a device tree.

### 📘 #address-cells and #size-cells — What Do They Mean?

These properties **define the format of the** reg **property** for child nodes.

They **tell the parser how to interpret the** reg = <...> property of child devices. This is especially important for buses like I2C, SPI, or memory-mapped devices.

### ✅ #address-cells = <1>;

This means that each child node will specify **1 cell (32 bits)** to describe its **address**.

* For an I2C bus, the **address** is just the I2C device address (like 0x76).
* So reg = <0x76>; is valid because you're only expected to provide **1 cell**.

🧠 Tip: 1 cell = 32 bits. But I2C addresses are 7 bits (or sometimes 10 bits), so 1 cell is more than enough.

### ✅ #size-cells = <0>;

This tells the kernel that **child nodes don’t need to specify a size** in the reg property.

Why? Because I2C devices **aren’t memory-mapped**, and they don’t expose regions of memory to the CPU like RAM or MMIO peripherals do. So, **no size information is needed**.

### 🌟 Summary (in your context)

dts

#address-cells = <1>; // Expect one 32-bit cell for each child’s address

#size-cells = <0>; // No size information needed for I2C devices

This setup allows you to cleanly write:

dts

pressure\_sensor@76 {

reg = <0x76>; // 7-bit I2C address

};

Let me know if you want a visual diagram of how these work across buses like I2C vs memory-mapped devices — it's super useful once you see it!

1. **Please explain:**

static const struct i2c\_device\_id pressure\_id[] = { { DRIVER\_NAME, 0 }, { } }; MODULE\_DEVICE\_TABLE(i2c, pressure\_id); static const struct of\_device\_id pressure\_of\_match[] = { { .compatible = "myvendor,mypressure" }, { } }; MODULE\_DEVICE\_TABLE(of, pressure\_of\_match);

Absolutely — this is a key part of how your Linux I2C driver gets matched to devices declared in the **Device Tree** or registered manually by platform code. Let’s break it down:

## 🔍 i2c\_device\_id and of\_device\_id — What are they for?

These two tables are how your driver says:

"Hey kernel, if you find a device that matches one of these identifiers, I'm the driver for it!"

### 🔧 i2c\_device\_id pressure\_id[]

c

static const struct i2c\_device\_id pressure\_id[] = {

{ DRIVER\_NAME, 0 },

{ }

};

MODULE\_DEVICE\_TABLE(i2c, pressure\_id);

* This table is used when a device is **registered without Device Tree**, usually via board files (board-\*.c) in older platforms.
* Each entry has:
  + .name: a string that matches the I2C device name.
  + .driver\_data: optional data you can access later (like chip variants).
* DRIVER\_NAME is "mypressure" — so if something calls i2c\_new\_device() with that name, this driver will match it.

➡️ This is important for non-Device-Tree setups, or for module autoloading.

### 🌳 of\_device\_id pressure\_of\_match[]

c

static const struct of\_device\_id pressure\_of\_match[] = {

{ .compatible = "myvendor,mypressure" },

{ }

};

MODULE\_DEVICE\_TABLE(of, pressure\_of\_match);

* This table is used when a device is described in the **Device Tree**, like your overlay.
* compatible = "myvendor,mypressure" must match the device tree node:

dts

pressure\_sensor@76 {

compatible = "myvendor,mypressure";

};

* When the kernel parses this node, it checks all registered drivers’ .of\_match\_table entries to find a match.

➡️ This is how your driver gets **bound automatically** to the I2C device at boot.

### 🧠 Why Do We Use MODULE\_DEVICE\_TABLE(...)?

This macro tells depmod (used when installing modules) to include those IDs in a special section so the kernel can **autoload the driver** when it sees a matching device.

Without this, the kernel might not know that your module supports "myvendor,mypressure".

### 🔁 How It All Ties Together

1. You describe your sensor in the Device Tree:

dts

compatible = "myvendor,mypressure";

reg = <0x76>;

1. The kernel finds that device on I2C at boot.
2. It checks all drivers with .of\_match\_table.
3. Your driver matches, probe() is called.

Let me know if you'd like to go into how driver\_data can be used for supporting multiple variants of a chip in the same driver!

1. **Please tell me more about "of" in MODULE\_DEVICE\_TABLE(of, pressure\_of\_match);**

Great — let’s zoom in on the "of" in this line:

c

MODULE\_DEVICE\_TABLE(of, pressure\_of\_match);

### 🔎 What is "of" in this context?

The "of" refers to **Open Firmware** (OF), which is the base concept behind the **Device Tree** system used in Linux.

So when you see of in Linux kernel code, especially in things like:

* of\_device\_id
* of\_match\_table
* MODULE\_DEVICE\_TABLE(of, ...)

…it’s referring to **Device Tree matching**.

### 🧠 MODULE\_DEVICE\_TABLE(of, pressure\_of\_match); — What Does It Do?

This macro does **two things**:

1. **Creates a special ELF section in your kernel module (**.modinfo**)** that contains a list of all compatible strings your driver supports.
2. **Lets the kernel auto-load your module** when it sees a Device Tree node with a matching compatible string.

So, in your case:

* Your driver supports "myvendor,mypressure".
* The kernel scans all loaded modules and finds that yours supports it — thanks to this macro.
* If your module isn't loaded yet, the kernel can use this information to **load it automatically**.

Without this line, modprobe **or** udev **won't know your driver supports that device**, and it won’t autoload even if everything else is right.

### 📋 Summary

| Term | Meaning |
| --- | --- |
| of | Refers to **Open Firmware** / Device Tree matching |
| MODULE\_DEVICE\_TABLE(of, ...) | Makes your driver compatible with Device Tree auto-loading |
| pressure\_of\_match[] | Table of compatible = "..."; strings your driver supports |

Let me know if you’d like to see how this plays with modinfo, or how you can test autoloading with a real .dtbo and driver!

1. **If I am using Device Tree. Does my driver require:**

static const struct i2c\_device\_id pressure\_id[] = { { DRIVER\_NAME, 0 }, { } }; MODULE\_DEVICE\_TABLE(i2c, pressure\_id);

Excellent question — and you're thinking in the right direction.

### ✅ Short Answer:

**No, it's not strictly required if you're only using Device Tree (**OF**)** — **but it is still a good idea to include it.** Here's why 👇

### 📦 Why It’s Technically Not Required with Device Tree

When you're using a Device Tree (DT), the kernel uses your driver's:

c

static const struct of\_device\_id pressure\_of\_match[] = { ... };

and matches it with the compatible = "..." property from the .dts. If you have:

c

.driver = {

.of\_match\_table = pressure\_of\_match,

},

Then **the probe function will be called based on the match from the DT** — no i2c\_device\_id table needed.

### 🤔 So Why Include i2c\_device\_id Then?

1. **Support for non-DT platforms**  
   Some boards still register I2C devices using legacy platform data or i2c\_board\_info, which requires this table. Including it makes your driver more portable and reusable.
2. **Module autoloading (optional, but helpful)**  
   In some older systems, MODULE\_DEVICE\_TABLE(i2c, ...) is used to help udev/modprobe recognize drivers when a new I2C device is added via sysfs or hotplug. Not strictly needed in DT-only systems, but may help in hybrid setups.
3. **Kernel conventions & consistency**  
   Many upstream kernel drivers still include both tables (even if one is unused) for completeness and to ease future maintenance.

### 🧪 Real-world Example

If you **omit the** i2c\_device\_id **table**, but your of\_match\_table is present and correct:

* ✅ Works fine for DT boot
* ❌ May not work if someone adds the device manually or uses the driver in a different system

### ✅ Best Practice (Recommended)

Even if you're using DT, include both tables:

c

static const struct i2c\_device\_id pressure\_id[] = {

{ "mypressure", 0 },

{ }

};

MODULE\_DEVICE\_TABLE(i2c, pressure\_id);

That way, you're future-proofed.

Let me know if you want an example of a board file using i2c\_register\_board\_info() and how it relies on the i2c\_device\_id table!

1. **Does the I2C sensor in this example mean it is a Port based input-output?**

Great question again! The I2C sensor in this example is **not** considered **port-based I/O** in the traditional sense. Let's clarify the terminology and what's really happening.

### 🧠 What is ****Port-Based I/O****?

**Port-based I/O (PIO)** refers to how some processors (especially x86) have **a separate I/O address space**. Devices are accessed using **special CPU instructions** like in and out.

* Found in x86 systems (e.g., using I/O ports like 0x3F8 for COM1).
* Uses a separate I/O address space, different from RAM.
* Accessed with inb(), outb(), etc.

🛑 **This is not how I2C devices work**.

### 🔌 What Kind of I/O is I2C?

I2C is a **bus-based communication protocol**, and it's typically treated as **memory-mapped or register-based I/O over a serial bus**.

* It's not PIO, and it doesn't use inb()/outb() instructions.
* Instead, you send and receive data over SDA (data) and SCL (clock) lines using I2C controller hardware.
* You access the I2C sensor’s internal registers using the kernel's **I2C subsystem**, not CPU ports.

Example:

c

i2c\_smbus\_read\_byte\_data(client, 0xF7); // Read register 0xF7 from the sensor

This communicates with the I2C peripheral through the controller driver.

### ⚙️ So What Is It Then?

I2C sensors are:

* **Peripheral devices** connected over a **serial bus** (not via I/O ports).
* Controlled through a **bus controller** (I2C master) in the SoC.
* Accessed through **high-level kernel APIs**, e.g., i2c\_smbus\_\*() or regmap.

So it’s better to think of I2C as:

**Bus-based peripheral I/O**, not port-based I/O.

### 🧩 Summary

| Feature | I2C Sensor (like in your example) | Port-Based I/O (e.g., x86 PIO) |
| --- | --- | --- |
| I/O type | Bus-based (serial) | Port-based (separate I/O space) |
| Accessed via | Kernel I2C APIs | inb(), outb() instructions |
| Platform | Embedded, ARM, etc. | Mostly x86 |
| Communication | Over SDA/SCL lines (via I2C master) | CPU directly accesses I/O ports |

Let me know if you’d like a diagram showing how an I2C sensor connects to the SoC and how the driver interacts with it — it helps a lot to visualize!

1. **How would be the device tree if there is an interrupt involved?**

Ah, great follow-up! If your I2C-based sensor supports an **interrupt line** (usually for things like data-ready, threshold alert, or FIFO full), then you'll need to tell the kernel **how to handle that interrupt** via the Device Tree.

Let’s walk through how that looks.

### 🧭 General Idea

You add an interrupts property to your device node, like this:

dts

pressure\_sensor@76 {

compatible = "myvendor,mypressure";

reg = <0x76>;

interrupt-parent = <&gpio>;

interrupts = <17 IRQ\_TYPE\_EDGE\_RISING>; // GPIO17, rising edge trigger

};

This tells the kernel:

“This I2C sensor uses GPIO17 as an interrupt line, and it triggers on a rising edge.”

### 🧩 Explanation of Each Property

| Property | Meaning |
| --- | --- |
| reg = <0x76>; | The I2C address of the sensor. |
| interrupt-parent = <&gpio>; | The GPIO controller that owns the pin (almost always <&gpio> on the Pi). |
| interrupts = <PIN FLAGS>; | Which GPIO pin and what kind of trigger (edge/level). |

### 🔧 Interrupt Flags (IRQ\_TYPE\_…)

| Macro | Value | Meaning |
| --- | --- | --- |
| IRQ\_TYPE\_NONE | 0 | No specific type |
| IRQ\_TYPE\_EDGE\_RISING | 1 | Trigger on rising edge |
| IRQ\_TYPE\_EDGE\_FALLING | 2 | Trigger on falling edge |
| IRQ\_TYPE\_EDGE\_BOTH | 3 | Trigger on both edges |
| IRQ\_TYPE\_LEVEL\_HIGH | 4 | High level trigger |
| IRQ\_TYPE\_LEVEL\_LOW | 8 | Low level trigger |

These values come from include/dt-bindings/interrupt-controller/irq.h.

### ⚡ Full Example Device Node with Interrupt

dts

pressure\_sensor@76 {

compatible = "myvendor,mypressure";

reg = <0x76>;

interrupt-parent = <&gpio>;

interrupts = <17 IRQ\_TYPE\_EDGE\_RISING>;

status = "okay";

};

### 🧠 What the Driver Needs to Do

In your driver, in the probe() function, you’ll retrieve the IRQ number and request it:

c

int irq = client->irq;

if (irq > 0) {

ret = devm\_request\_threaded\_irq(&client->dev, irq,

NULL, // optional top-half

pressure\_irq\_handler,

IRQF\_ONESHOT,

"mypressure", client);

}

And then define the threaded IRQ handler:

c

static irqreturn\_t pressure\_irq\_handler(int irq, void \*dev\_id)

{

struct i2c\_client \*client = dev\_id;

dev\_info(&client->dev, "Interrupt received!\n");

// Read data, clear interrupt, etc.

return IRQ\_HANDLED;

}

### ✅ Notes for Raspberry Pi

* GPIO numbers are **BCM GPIO numbers**. For example, GPIO17 = physical pin 11 on Pi header.
* Make sure your sensor’s interrupt pin is pulled up/down as needed.
* Ensure GPIO is not being used by another overlay (e.g., UART, SPI, etc.).

Would you like a full .dts file with a GPIO interrupt and maybe a matching test kernel module?

1. **Is devm\_request\_threaded\_irq() something you made-up or is it in some header files?**

Awesome that you're looking closely — and nope, I didn’t make it up 😄

### ✅ devm\_request\_threaded\_irq() is ****a real Linux kernel API****

It’s defined in the Linux kernel and provided by the **device-managed resource framework** (devm\_\*()), which helps you automatically free resources when a device is removed or the driver is unloaded.

### 📁 Where is it defined?

You'll find it in:

c

#include <linux/interrupt.h>

Specifically, it’s declared like this:

c

int devm\_request\_threaded\_irq(struct device \*dev,

unsigned int irq,

irq\_handler\_t handler,

irq\_handler\_t thread\_fn,

unsigned long irqflags,

const char \*devname,

void \*dev\_id);

### 🔍 What Does It Do?

It’s similar to request\_threaded\_irq(), but it's **device-managed**:

* No need to manually call free\_irq() in remove().
* The kernel will automatically release the IRQ when your driver is removed.

### 🧠 Parameters Breakdown

| Parameter | Description |
| --- | --- |
| dev | Device pointer, usually &client->dev for I2C |
| irq | IRQ number (e.g., from client->irq) |
| handler | Top-half interrupt handler (can be NULL if you only use bottom-half) |
| thread\_fn | Bottom-half threaded IRQ handler (runs in kernel thread) |
| irqflags | Flags like IRQF\_ONESHOT, IRQF\_TRIGGER\_RISING, etc. |
| devname | Name used in /proc/interrupts |
| dev\_id | Typically client, used to match when freeing IRQs |

### 🧪 Example Use in Real Drivers

c

ret = devm\_request\_threaded\_irq(&client->dev, client->irq,

NULL, pressure\_irq\_handler,

IRQF\_ONESHOT | IRQF\_TRIGGER\_RISING,

"mypressure", client);

This sets up an interrupt for client->irq, no top-half, and runs pressure\_irq\_handler() in a kernel thread.

### 🧼 Clean-up? Already Handled!

No need to do:

c

free\_irq(irq, client);

Because devm\_\*() will do it **automatically** when the device goes away. Super clean.

Let me know if you want to see how devm\_request\_threaded\_irq() compares to the classic request\_irq() or request\_threaded\_irq() in terms of lifecycle and error handling!