Linux Platform Device & Driver Model

- https://docs.kernel.org/driver-api/driver-model/platform.html

1. Platform Bus: The Virtual Bus for Embedded Systems

Concepts

The platform bus is a *virtual bus* in Linux that connects non-discoverable devices (typically SoC peripherals) with their drivers. Unlike PCI or USB, these devices:

- **Cannot self-identify** (no hardware enumeration)
- **Have fixed addresses** (memory/IRQs defined in hardware)
- **Require manual declaration** (via board files or Device Tree)

```
// Kernel representation (simplified)
struct bus_type platform_bus_type = {
    .name     = "platform",
    .match    = platform_match, // Matching logic
    .uevent   = platform_uevent,
};
```

Key Characteristics:

- Parent to all SoC peripherals (e.g., AM335x GPIO, I2C)
- Enables device/driver binding without physical bus
- Central to embedded Linux driver development

2. Platform Device Registration Methods

A. Board Files (Legacy Approach)

When Used: Older kernels (pre-DT) or quick prototyping
Mechanism: Hardcoded in machine-specific C files

```
/* Example: AM335x GPIO controller declaration */
static struct resource am335x_gpio_res[] = {
    [0] = { // Memory region
        .start = 0x4804C000, // Physical base
        .end   = 0x4804CFFF,
        .flags = IORESOURCE_MEM,
    },
    [1] = { // IRQ
        .start = 98,
        .flags = IORESOURCE_IRQ,
    }
};

static struct platform_device am335x_gpio_dev = {
    .name = "am335x-gpio",
    .id = -1,
    .resource = am335x_gpio_res,
    .num_resources = ARRAY_SIZE(am335x_gpio_res),
};
```

Registration:

```
platform_device_register(&am335x_gpio_dev); // Typically in arch/arm/mach-*.c
```

Note: The above board file is for example only. BeagleBone Black NEVER used board files. AM335x was DT-only from the beginning.

Pros/Cons:

- ✅ Direct control

- ❌ Not portable across hardware revisions
- ❌ Hard to maintain

Reference:

- https://github.com/beagleboard/linux/blob/master/arch/arm/mach-omap1/board-nokia770.c
- Refer function: omap_nokia770_init()

B. Device Tree (Modern Standard)

When Used: All contemporary kernels (including BBB)
Mechanism: Hardware description in `.dts` files

```
// am33xx-l4.dtsi fragment
gpio1: gpio@4804C000 {
    compatible = "ti,omap4-gpio";
    gpio-ranges =   <&am33xx_pinmux  0  0  8>,
            <&am33xx_pinmux  8 90  4>,
            <&am33xx_pinmux 12 12 16>,
            <&am33xx_pinmux 28 30  4>;
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
    reg = <0x4804C000 0x1000>;
    interrupts = <98>;
};
```

Flow:

1. Bootloader loads compiled `.dtb`
2. Kernel parses DT during boot
3. Platform devices created automatically

Advantages:

- Hardware-agnostic drivers
- Single source for hardware info
- Runtime modifiable via overlays

C. Dynamic Registration

When Used: Runtime-detected devices or hotpluggable peripherals

```
struct platform_device *pdev;
pdev = platform_device_alloc("dynamic-dev", PLATFORM_DEVID_AUTO);
if (!pdev) return -ENOMEM;

// Add resources post-allocation
platform_device_add_resources(pdev, res, nres);

// Finalize registration
int ret = platform_device_add(pdev);
if (ret) {
    platform_device_put(pdev);
    return ret;
}
```

Use Cases:

- Loadable FPGA configurations
- USB-to-SPI bridges
- Runtime-discovered IP blocks

3. Platform Driver Implementation

Core Structure

```c
#include <linux/platform_device.h>

static int my_probe(struct platform_device *pdev)
{
    // 1. Get device resources
    // 2. Initialize hardware
    // 3. Register interfaces
}

static int my_remove(struct platform_device *pdev)
{
    // Reverse probe operations
}

static struct platform_driver my_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "my-device",
        .of_match_table = of_match_ptr(my_of_match),
    },
};
module_platform_driver(my_driver);
```

Critical Functions:

1. **Probe()**
    - Entry point when device/driver match
    - Allocates resources, registers ops
    - Returns `0` on success, error code otherwise
2. **Remove()**

- Handles module unload/device removal
- Must release all resources

4. Device/Driver Matching Process

Matching Mechanisms

1. Device Tree First (Preferred):

```
static const struct of_device_id my_of_match[] = {
    { .compatible = "vendor,device123" },
    {}
};
```

2. Fallback to ID Table:

```
static struct platform_device_id my_id_table[] = {
    { "legacy-device", 0 },
    {}
};
```

3. Name Matching:

```
// Driver:
.name = "fixed-name"

// Device:
.name = "fixed-name" // Must match exactly
```

5. Resource Management Best Practices

Accessing Resources

```
// Get MMIO region
struct resource *res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
void __iomem *base = devm_ioremap_resource(&pdev->dev, res);

// Get IRQ
int irq = platform_get_irq(pdev, 0);
devm_request_irq(&pdev->dev, irq, handler, 0, "my-irq", NULL);
```

Key APIs:

Function	Purpose	Managed Version
`ioremap()`	Map physical memory	`devm_ioremap_resource()`
`request_irq()`	Register ISR	`devm_request_irq()`
`clk_get()`	Get clock	`devm_clk_get()`

Why Managed (`devm_*`)?
Automatically releases resources on driver detach, preventing leaks.

BBB Practical Example

AM335x GPIO Controller Driver

```
static int am335x_gpio_probe(struct platform_device *pdev)
{
    struct device *dev = &pdev->dev;
```

```
    // 1. Get DT properties
    u32 ngpios;
    of_property_read_u32(dev->of_node, "ngpios", &ngpios);

    // 2. Map registers
    void __iomem *base = devm_platform_ioremap_resource(pdev, 0);

    // 3. Get clock
    struct clk *clk = devm_clk_get(dev, NULL);

    dev_info(dev, "Probed GPIO with %d pins\n", ngpios);
    return 0;
}

static const struct of_device_id am335x_gpio_of_match[] = {
    { .compatible = "ti,am335x-gpio" }, // Matches DT node
    {}
};
```

Testing on BBB:

```
# Check registered platform devices
ls /sys/bus/platform/devices/

# View driver bindings
cat /proc/device-tree/ocp/gpio@4804c000/compatible
# Output: ti,am335x-gpio
```

Device Tree (DT) for Embedded Linux

1. Introduction to Device Tree

Why Device Tree?

- **Hardware Description**: Replaces hardcoded board files (arch/arm/mach-*)
- **Portability**: Single kernel image supports multiple hardware variants
- **Runtime Flexibility**: Overlays enable dynamic hardware changes

Key Components

Component	Purpose	Example
DTS	Source file	`am335x-boneblack.dts`
DTC	Compiler	`dtc` tool
DTB	Binary blob	Loaded by bootloader
DTSI	Include files	`am33xx.dtsi`

2. Device Tree Syntax & Structure

Basic Structure

```
/dts-v1/;
/ {
    node1 {
        property1 = value;
        child-node {
            property2 = [hex values];
        };
    };
};
```

Example: GPIO LED & Button (BBB)

```
/dts-v1/;
#include "am33xx.dtsi"

/ {
    model = "TI AM335x BeagleBone Black";
    compatible = "ti,am335x-bone-black", "ti,am335x-bone";

    leds {
        compatible = "gpio-leds";
        led0 {
            label = "beaglebone:green:usr0";
            gpios = <&gpio1 21 GPIO_ACTIVE_HIGH>;
            linux,default-trigger = "heartbeat";
        };
    };

    gpio_keys {
        compatible = "gpio-keys";
        button0 {
            label = "USER_BUTTON";
            gpios = <&gpio0 7 GPIO_ACTIVE_LOW>;
            linux,code = <0x100>; /* BTN_0 */
        };
    };
};
```

3. Node Types & Properties

Common Property Types

Type	Syntax	Example
String	`prop = "value";`	`compatible = "ti,am335x-gpio";`

Type	Syntax	Example
Cell	`prop = <value>;`	`interrupts = <98>;`
Binary	`prop = [hex];`	`reg = <0x4804c000 0x1000>;`
Phandle	`prop = <&label>;`	`gpios = <&gpio1 12 0>;`

Example: I2C LCD Display

```
&i2c1 {  // BBB I2C1 bus
    status = "okay";
    pinctrl-names = "default";
    pinctrl-0 = <&i2c1_pins>;

    lcd: lcd@27 {
        compatible = "newhaven,nhd-0216";
        reg = <0x27>;
        backlight-gpios = <&gpio1 18 GPIO_ACTIVE_HIGH>;
    };
};
```

4. Device Tree Compilation Process

Compilation Commands

```
# Single file
dtc -I dts -O dtb -o am335x-boneblack.dtb am335x-boneblack.dts

# For BBB kernel (with includes)
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- dtbs
```

Output Locations

- Compiled DTBs: `/boot/dtbs/$(uname -r)/`
- Source in kernel: `arch/arm/boot/dts/`

5. Loading Device Tree on BBB

Bootloader (U-Boot) Commands

```
# List available DTBs
ls mmc 0:1

# Load and boot
load mmc 0:1 ${fdtaddr} /boot/dtbs/am335x-boneblack.dtb
setenv fdtfile am335x-boneblack.dtb
boot
```

Runtime Overlays

- Runtime Hardware Configuration: Modify hardware mapping without rebooting
- Cape Emulation: Dynamically add/remove capes (expansion boards)
- Pin Multiplexing: Change pin functions on-the-fly (e.g., GPIO → I2C)

```
# Config-pin for pinmux
config-pin P9.24 i2c  # Set pin to I2C mode

# Load overlay
echo BB-I2C1 > /sys/devices/platform/bone_capemgr/slots
```

6. Advanced Concepts

Address Mapping

```
ocp {  // On-Chip Peripherals
    uart0: serial@44e09000 {
        compatible = "ti,am3352-uart";
        reg = <0x44e09000 0x2000>;
        interrupts = <72>;
    };
};
```

Interrupt Handling

```
gpio0: gpio@44e07000 {
    compatible = "ti,am335x-gpio";
    interrupts = <96>;  // IRQ number
    interrupt-controller;
    #interrupt-cells = <2>;
};
```

Clock Specification

```
clocks {
    sys_clk: clock@0 {
        #clock-cells = <0>;
        compatible = "fixed-clock";
        clock-frequency = <24000000>;
    };
};

uart0: serial@44e09000 {
```

```
    clocks = <&sys_clk>;
};
```

7. Debugging Device Tree

Inspect Loaded DT

```
# View full DT
dtc -I fs /sys/firmware/devicetree/base

# Check specific properties
cat /proc/device-tree/model
```

Runtime Validation

```
# Check pinmux
cat /sys/kernel/debug/pinctrl/44e10800.pinmux/pins

# Verify I2C devices
i2cdetect -y 1
```

8. Practical BBB Examples

SPI Device Example

```
&spi0 {
    status = "okay";
    pinctrl-names = "default";
    pinctrl-0 = <&spi0_pins>;
```

```
    adc@0 {
        compatible = "microchip,mcp3008";
        reg = <0>;
        spi-max-frequency = <1000000>;
        vref-supply = <&vdd_3v3>;
    };
};
```

PWM Configuration

```
&epwmss0 {
    status = "okay";

    pwm: pwm@48300200 {
        status = "okay";
        pinctrl-names = "default";
        pinctrl-0 = <&pwm_pins>;
    };
};
```

Further References:

- https://github.com/beagleboard/linux/blob/master/arch/arm/boot/dts/ti/omap/am335x-boneblack.dts
- https://github.com/beagleboard/linux/blob/master/arch/arm/boot/dts/ti/omap/am335x-boneblack-common.dtsi
- https://github.com/beagleboard/linux/blob/master/arch/arm/boot/dts/ti/omap/am33xx.dtsi

---

I2C Device Drivers in Linux

1. I2C Protocol Basics

Core Characteristics

- **Two-wire communication** (SDA + SCL)
- **Master-slave architecture** (Multi-master possible)
- **7/10-bit addressing** (0x08-0x77 for 7-bit)
- **Speed modes**:
  - Standard (100 kHz)
  - Fast (400 kHz)
  - Fast+ (1 MHz)
  - High-speed (3.4 MHz)

Transaction Structure

```
Start → [Address+R/W] → ACK → [Data] → ACK/NACK → ... → Stop
```

Key Points:

- Clock stretching allowed
- No error detection (unlike SPI)
- Open-drain signaling (requires pull-ups)

2. I2C vs SMBus

Feature	I2C	SMBus
Voltage Levels	1.8V-5V	2.7V-5V
Clock Speed	Up to 3.4 MHz	Max 100 kHz
Timeout	None	35ms mandatory
Packet Error Chk	No	PEC (optional)

Feature	I2C	SMBus
Standard Commands	No	Yes (0x00-0x1F)
Linux Support	`i2c.h`	`i2c-smbus.h`

Practical Implications:

- SMBus devices work on I2C buses
- I2C devices may fail on strict SMBus controllers
- Always prefer `i2c_smbus_*` APIs for compatibility

3. Linux I2C Subsystem

Key Components

1. **I2C Adapter** (Controller)
    - Represented by `struct i2c_adapter`
    - Registered by SoC-specific drivers (e.g., `i2c-omap` for BBB)
2. **I2C Client** (Device)
    - `struct i2c_client` represents a slave device
    - Bound to driver via `i2c_driver` structure
3. **I2C Core**
    - Handles bus transactions
    - Manages driver/device matching

4. I2C Client Driver APIs

Essential Headers

```
#include <linux/i2c.h>          // Core I2C
#include <linux/i2c-dev.h>      // User-space interface
#include <linux/i2c-smbus.h>    // SMBus extensions
```

Driver Registration

```
static struct i2c_driver my_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "my_i2c_drv",
        .of_match_table = my_of_match,
    },
    .id_table = my_id_table,
};
module_i2c_driver(my_driver);
```

Communication APIs

SMBus-Compatible Functions (Preferred)

```
// Read 8-bit register
s32 i2c_smbus_read_byte_data(const struct i2c_client *client, u8 reg);

// Write 16-bit value
s32 i2c_smbus_write_word_data(const struct i2c_client *client, u8 reg, u16 value);

// Block read
s32 i2c_smbus_read_block_data(const struct i2c_client *client, u8 reg, u8 *vals);
```

Raw I2C Transfers

```
struct i2c_msg {
    __u16 addr;     // Slave address
    __u16 flags;    // I2C_M_RD, etc.
    __u16 len;      // Message length
    __u8 *buf;      // Data buffer
};

int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num);
```

Device Tree Binding Example

```
&i2c1 {  // BBB I2C1 bus
    status = "okay";
    pinctrl-names = "default";
    pinctrl-0 = <&i2c1_pins>;

    temp_sensor: tmp102@48 {
        compatible = "ti,tmp102";
        reg = <0x48>;
        vcc-supply = <&vdd_3v3>;
    };
};
```

5. Key Implementation Details

Probe Function Essentials

```c
static int my_probe(struct i2c_client *client)
{
    struct device *dev = &client->dev;

    // Verify functionality
    if (!i2c_check_functionality(client->adapter,
                                 I2C_FUNC_SMBUS_BYTE_DATA)) {
        dev_err(dev, "SMBus byte ops not supported\n");
        return -EIO;
    }

    // Read device ID (example)
    u8 dev_id = i2c_smbus_read_byte_data(client, REG_WHO_AM_I);
    if (dev_id != EXPECTED_ID) {
        dev_err(dev, "Invalid device ID: 0x%02x\n", dev_id);
        return -ENODEV;
    }

    // Initialize device
    // ...
    return 0;
}
```

Power Management

```c
static int my_suspend(struct device *dev)
{
    struct i2c_client *client = to_i2c_client(dev);
    i2c_smbus_write_byte_data(client, REG_POWER, POWER_DOWN);
    return 0;
}

static const struct dev_pm_ops my_pm_ops = {
```

```
    SET_SYSTEM_SLEEP_PM_OPS(my_suspend, my_resume)
};
```

6. BBB-Specific Considerations

I2C Controllers in AM335x

- **I2C0**: not accessible through the expansion headers on bbb
- **I2C1**: often used for on-board EEPROM access and internal power management.
- **I2C2**: available for external I2C device

Debugging Tools

```
# Scan I2C bus (BBB I2C1)
i2cdetect -y -r 1

# Dump registers
i2cdump -y 1 0x48 b

# Read byte
i2cget -y 1 0x48 0x00 b
```

7. I2C LCD Character Driver - Example

1. Driver Architecture

```
[User Space] --> [Char Device] --> [I2C Transfers] --> [I2C Device]
```

2. Full Driver Implementation

```c
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/i2c.h>
#include <linux/slab.h>

#define I2C_BUS_NUM     2       // BBB I2C2 (P9.19/20)
#define LCD_ADDR        0x27    // PCF8574 I2C LCD address
#define DEVICE_NAME     "i2c_lcd"
#define MAX_TEXT        40      // Max characters per write

static struct i2c_adapter *lcd_i2c_adapter;
static struct i2c_client *lcd_i2c_client;
static dev_t dev_num;
static struct cdev lcd_cdev;
static struct class *lcd_class;

// Private device structure
struct lcd_data {
    struct i2c_client *client;
    char buffer[MAX_TEXT];
};

// LCD Low-Level Functions
static void lcd_send_cmd(struct i2c_client *client, u8 cmd)
{
    u8 data[4];
    // ...
    i2c_master_send(client, data, 4);
    msleep(2); // Wait for command execution
}

static void lcd_send_data(struct i2c_client *client, u8 dat)
{
    u8 data[4];
```

```
    // ...
    i2c_master_send(client, data, 4);
    udelay(100);
}

// Initialize LCD
static void lcd_init(struct i2c_client *client)
{
    msleep(50); // Power-on delay

    // Initialize 4-bit mode ...
}

// File Operations
static int lcd_open(struct inode *inode, struct file *file)
{
    struct lcd_data *data;

    data = kmalloc(sizeof(*data), GFP_KERNEL);
    if (!data)
        return -ENOMEM;

    data->client = lcd_i2c_client;
    file->private_data = data;

    return 0;
}

static ssize_t lcd_write(struct file *file, const char __user *buf,
                         size_t len, loff_t *ppos)
{
    struct lcd_data *data = file->private_data;
    // send data and commands as appropriate
    lcd_send_cmd(data->client, ...);
    lcd_send_data(data->client, ...);
    return len;
```

```
}

static int lcd_release(struct inode *inode, struct file *file)
{
    kfree(file->private_data);
    return 0;
}

static struct file_operations lcd_fops = {
    .owner = THIS_MODULE,
    .open = lcd_open,
    .write = lcd_write,
    .release = lcd_release,
    // ...
};

// I2C Driver Functions
static int lcd_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
    int ret;
    // Initialize LCD hardware
    lcd_init(client);
    // return error if init failed

    // Initialize character device
    ret = alloc_chrdev_region(&dev_num, 0, 1, DEVICE_NAME);
    if (ret < 0) {
        pr_err("Failed to allocate device number\n");
        return ret;
    }

    cdev_init(&lcd_cdev, &lcd_fops);
    ret = cdev_add(&lcd_cdev, dev_num, 1);
    if (ret < 0) {
        unregister_chrdev_region(dev_num, 1);
        return ret;
```

```
    }

    // Create device class and node
    lcd_class = class_create(THIS_MODULE, DEVICE_NAME);
    device_create(lcd_class, NULL, dev_num, NULL, DEVICE_NAME);

    pr_info("I2C LCD driver probed successfully\n");
    return 0;
}

static int lcd_remove(struct i2c_client *client)
{
    // Cleanup character device
    device_destroy(lcd_class, dev_num);
    class_destroy(lcd_class);
    cdev_del(&lcd_cdev);
    unregister_chrdev_region(dev_num, 1);

    pr_info("I2C LCD driver removed\n");
    return 0;
}

static const struct i2c_device_id lcd_id[] = {
    { DEVICE_NAME, 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, lcd_id);

static struct i2c_driver lcd_driver = {
    .driver = {
        .name = DEVICE_NAME,
        .owner = THIS_MODULE,
    },
    .probe = lcd_probe,
    .remove = lcd_remove,
    .id_table = lcd_id,
```

```
};

// Board info for manual device creation
static struct i2c_board_info lcd_board_info = {
    I2C_BOARD_INFO(DEVICE_NAME, LCD_ADDR)
};

static int __init lcd_init_module(void)
{
    int ret;

    // Get I2C adapter
    lcd_i2c_adapter = i2c_get_adapter(I2C_BUS_NUM);
    if (!lcd_i2c_adapter) {
        pr_err("I2C adapter not found\n");
        return -ENODEV;
    }

    // Create I2C device
    lcd_i2c_client = i2c_new_client_device(lcd_i2c_adapter, &lcd_board_info);
    if (!lcd_i2c_client) {
        pr_err("Failed to create I2C device\n");
        i2c_put_adapter(lcd_i2c_adapter);
        return -ENODEV;
    }

    // Register driver
    ret = i2c_add_driver(&lcd_driver);
    if (ret) {
        i2c_unregister_device(lcd_i2c_client);
        i2c_put_adapter(lcd_i2c_adapter);
        return ret;
    }

    return 0;
}
```

```c
static void __exit lcd_exit_module(void)
{
    i2c_del_driver(&lcd_driver);
    if (lcd_i2c_client)
        i2c_unregister_device(lcd_i2c_client);
    if (lcd_i2c_adapter)
        i2c_put_adapter(lcd_i2c_adapter);
}

module_init(lcd_init_module);
module_exit(lcd_exit_module);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("I2C LCD Character Driver for BBB");
```

3. Key Components Explained

1. I2C Communication

- **Manual Device Creation**: Using `i2c_new_client_device()` instead of DT
- **4-bit Mode Handling**: Required for PCF8574-based LCDs
- **Timing Control**: Proper delays between commands

2. Character Device Operations

- **Standard File Ops**: `open()`, `write()`, `release()`
- **Userspace Buffer Handling**: `copy_from_user()` for safe data transfer
- **Private Data**: Stores I2C client reference per file handle

4. Testing the Driver

Build & Load

```
make -C /lib/modules/$(uname -r)/build M=$PWD modules
sudo insmod i2c_lcd.ko
```

Verify Operation

```
# Check device file
ls -l /dev/i2c_lcd

# Write to display
echo "Hello\nBBB" > /dev/i2c_lcd

# Check kernel messages
dmesg | tail
```

Unload

```
sudo rmmod i2c_lcd
```

5. Important Notes

1. **Hardware Connection**:
    - P9.17 (I2C1_SCL) → LCD SCL
    - P9.18 (I2C1_SDA) → LCD SDA
    - Proper pull-ups required (4.7kΩ to 3.3V)
2. **LCD Compatibility**:

- Designed for HD44780-compatible LCDs with PCF8574 I2C backpack
- Address may vary (0x27 or 0x3F)

3. **Error Handling**:

- Verify I2C bus with `i2cdetect -y 1`
- Check connections if probe fails