

Platform Devices & Device Tree

ECEN 427

BYU Electrical & Computer
Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

Why does Linux need a device tree?

- Discovering devices:
 - USB and PCI devices announce themselves, so the kernel can find them by asking the bus
 - Memory-mapped SoC peripherals, like everything in our FPGA, can't be found that way: there is nothing to ask
- **platform devices:** devices that cannot be discovered; need to be described ahead of time.
 - Something has to tell the kernel that each one exists, where its registers are, and which interrupt it uses
 - That something is the **device tree**: a data file describing the hardware, kept separate from the kernel binary

<https://byu-cpe.github.io/ecen427/documentation/platform-device-tree/>

Device tree files in our system

https://github.com/byu-cpe/ecen427_student/tree/main/device_tree

- Source .dts/.dtsi → dtc compiler → binary .dtb/.dtbo
- pynq.dts: the base tree, loaded by the bootloader at boot. We never change it
- ecen427.dtsi → ecen427.dtbo
 - an **overlay** added to the live tree at runtime, along with the bitstream (configure_hardware.sh, through configs)
 - Our overlay adds its nodes under amba, next to the ARM's built-in peripherals
 - In Lab 6 you'll add a node for your own hardware

Anatomy of a node

```
ecen427_user_intc {  
    compatible = "generic-uio";  
    reg = <0x41800000 0x10000>;  
    interrupt-parent = <&intc>;  
    interrupts = <0x0 0x1E 0x04>;  
};
```

- **compatible:** a "vendor,device" string that decides which driver handles this node
- **reg:** base address and size, the same values as Vivado's Address Editor
- **interrupt-parent:** which interrupt controller the line goes to (the ARM GIC)
- **interrupts:** three numbers the GIC defines: type (0 = shared peripheral), number (0x1E = 30), flags (4 = level-triggered, active high)

Connecting device tree node to driver

```
static struct of_device_id my_of_match[] = {
    { .compatible = "vendor,my-device" },
    { /* NULL entry in array indicates end of array */ }
};
```

```
MODULE_DEVICE_TABLE(of, my_of_match);
```

```
static struct platform_driver my_driver = {
    .driver = { .name = "my_driver",
                .of_match_table = my_of_match },
    .probe   = my_probe,
    .remove  = my_remove,
};
```

- The kernel creates a `platform_device` for each node in the tree
- Register your driver as one that handles platform devices:

```
int platform_driver_register(struct platform_driver *drv);
```
- Each match calls `probe(pdev)` once, for that one device

`my_init() {`

`my_probe()`

`my_remove()`

`my_exit() {`

Querying the Device Tree

- When your probe is called, you are given a pointer to a struct platform_device

```
int my_probe(struct platform_device *pdev);
```

- You can query for the memory addresses and IRQ values:

```
struct resource *platform_get_resource(struct platform_device *dev,  
                                     unsigned int type, unsigned int num);
```

```
struct resource {  
    resource_size_t start;  
    resource_size_t end;  
    ...  
};
```

Use IORESOURCE_MEM

If you want the first <reg>
entry, pass in a 0

```
int platform_get_irq(struct platform_device *dev, unsigned int num);
```

- The IRQ return value is Linux's **virtual** IRQ number, not the 0x1E in the device tree.

Creating a device file in /dev

- First you must create a “device class”
 - a group of devices of the same kind (tty, input, uio, ...)

```
struct class *class_create(const char *name);
```

- Creating a device file in /dev

```
struct device *device_create(const struct class *cls, struct device *parent,  
                             dev_t devt, void *drvdata, const char *fmt, ...);
```

- **parent**: the device this one sits under in the kernel's device hierarchy (sysfs).
 - NULL puts it at the top level (/sys/devices/virtual/<class>/).
 - Passing &pdev->dev nests it under the platform device in sysfs instead, linking the /dev file to the hardware node it came from.
 - Either one works for the lab.
- **devt**: major/minor number (make sure this matches the one you gave to cdev_add)
- **drvdata**: Any pointer you choose to store in the device. NULL is fine.
- **fmt, ...**: the name, formatted like printf. It becomes /dev/<name> and /sys/class/<class>/<name>.
 - With several devices, use something like "my_device%d", minor

Driver vs Device

When	Runs	Do this
<code>module_init</code>	once per module load (per driver)	Get major/minor numbers, register as platform driver, create device class
<code>probe</code>	once per matching node (per device)	Create character device, create /dev file, reserve and map memory, setup interrupts
<code>remove</code>	once per device	undo probe, in reverse order
<code>module_exit</code>	once per unload	undo init, in reverse order