Platform device drivers

Mg. Ing. Gonzalo E. Sanchez
MSE - 2022

Implementación de Manejadores de Dispositivos

Platform device drivers

- Introducción
- Platform drivers
- Platform devices
- Dispositivos, drivers y bus matching.
- Platform drivers y OF matching

Introducción

Introducción

- El tipo de dispositivos *plug&play* es bien conocido para todos.
- Son manejados por el kernel tan pronto se conectan.
- Ejemplos:
  - Dispositivos conectados al bus USB o PCI Express.
  - Otros dispositivos con capacidad de ser descubiertos automáticamente.
- Existen dispositivos que no son de este tipo (no son hot plug-in).
- El kernel necesita saber de estos antes de poder

Introducción

- Dispositivos que no son automáticamente descubiertos están conectados a los buses:
  - SPI
  - I2C
  - I2S
  - UART
  - PCI
  - Otros dispositivos conectados a buses sin capacidad de enumeración.

Introducción

- Estos buses son dispositivos de hardware que reciben el nombre de controladores.
- Son parte del SoC, no pueden removerse.
- No pueden ser descubiertos.
- Son llamados **platform devices**.
- NOTA: Se suele decir que los platform devices son on-chip devices (embebidos en el SoC): no es del todo correcto.

Introducción

- En la práctica, la mayoría de los platform devices cumplen el requisito de estar integrados en el SoC.
- Los buses SPI e I2C pueden conectar dispositivos fuera del SoC y tampoco pueden ser descubiertos.
- Esto los hace platform devices también.
- Del mismo modo, puede haber dispositivos USB dentro del SoC pero no son del tipo platform device (auto-discoverable).

Introducción

- Desde el punto de vista del SoC, los dispositivos se conectan internamente a través de buses dedicados (propietarios).
- Desde el punto de vista del kernel, son dispositivos del tipo root que no están conectados a nada.
- Para salvar este problema, se incluye el concepto del bus *pseudo platform*.
- También llamado *platform bus*, se trata de un bus virtual del kernel el cual no representa ningún bus físico.

Introducción

- Trabajar con un platform device requiere de dos pasos:
  - Registrar un driver del tipo platform con un nombre único que manejara los dispositivos deseados.
  - Registrar un platform device con el mismo nombre del driver, con los recursos correspondientes para conocimiento del kernel.
- En nuestro caso, es necesario para trabajar con el bus I2C.
- Se podría definir un bus propietario basado en GPIO (ejemplo LCD alfanumérico, LCD gráficos, Conversor DA del tipo R2R).

Introducción

- **ATENCIÓN:** No todos los platform devices son manejados por platform drivers!!
- Los drivers del tipo platform están dedicados a dispositivos que no se basan en buses convencionales.
- Los dispositivos SPI como los I2C son platform devices, pero no dependen del platform bus.
- Dependen de los correspondientes buses SPI e I2C.
- Absolutamente TODO debe ser hecho a mano en un platform driver.

Platform Drivers

Platform Drivers

- Todo platform device debe implementar una función probe() que el kernel llama en dos situaciones:
  - Cuando se inserta el módulo.
  - Cuando un dispositivo lo reclama.
- Al implementar platform drivers la estructura principal que debe llenarse es **struct platform_driver**.
- Se debe registrar el driver en el platform bus core con funciones dedicadas como se muestra a continuación.

Platform Drivers

```
static struct platform_driver mypdrv = {
    .probe = my_pdrv_probe,
    .remove = my_pdrv_remove,
    .driver = {
        .name = "my_platform_driver",
        .owner = THIS_MODULE,
    },
};
```

Platform Drivers

- **probe():** Se llama cuando un dispositivo reclama este driver luego de un match.

```
static int my_pdrv_probe(struct platform_device *pdev)
```

- **remove():** Se llama para remover el driver cuando no es ya necesario para ningún dispositivo.

```
static int my_pdrv_remove(struct platform_device *pdev)
```

- **Struct device_driver:** Es la descripción propia del driver.

Platform Drivers

- Registrar un platform driver es tan simple como llamar a las funciones **platform_driver_register()** o **platform_driver_probe().**
- Esto debe ser hecho dentro de la función **init().**
- Existe una diferencia sutil entre las funciones mencionadas:
  - **platform_driver_register()**: Registra el driver y lo incluye en una lista en el kernel permitiendo la llamada de la función probe() en cualquier momento.
  - **platform_driver_probe()**: El kernel corre el loop de match inmediatamente. Si el dispositivo está presente, corre **probe()**. Caso contrario el kernel ignora el driver.

Platform Drivers

- **NOTA: platform_driver_probe()** se utiliza en casos que se desea minimizar el footprint del driver en memoria.
- Esto se da porque **probe()** se sitúa en una sección **__init** que se libera al finalizar el booteo de kernel.
- Previene la llamada a **probe()** diferida, porque no registra el driver en el sistema.
- Este método solo debe usarse cuando se esta 100% seguro de que el dispositivo está presente
- Ejemplo: Memoria EEPROM parte del PCB.

Introducción al Device Tree

HANDS ON

1. Examinar archivo Ejemplo: simple_platform_driver.c

Platform Drivers

- En el archivo examinado, las funciones init/exit no hacen nada más que registrar/remover el driver en el platform bus core.
- Esto en realidad pasa con la mayoría de los drivers.
- Se puede entonces remover las llamadas a **module_init()** y **module_exit()**.
- En su lugar se utiliza la macro **module_platform_driver**.
- Esta macro se encarga de registrar el módulo en el platform driver core.

Introducción al Device Tree

HANDS ON

1. Ejecutar el siguiente comando en la raíz de las fuentes del kernel: **grep -r "#define module_platform_driver"**
2. Examinar el header localizado.
3. Encontrar la definición de la macro.

Platform Drivers

- **NOTA:** Utilizar esta macro no implica que las secciones **__init** y **__exit** no son necesarias.
- Recordar que estas definiciones tienen que ver con la generación y ubicación del código binario.
- Tampoco implica que dejan de utilizarse **module_init()** y **module_exit()**.
- La utilización de la macro solo evita la cantidad de *boilerplate* necesario. Está escrito, pero no por el desarrollador (nosotros).

Platform Drivers

- Existen macros específicas para cada bus donde se deben registrar drivers:
  - module_platform_driver(struct platform_driver).
  - module_spi_driver(struct spi_driver).
  - module_i2c_driver(struct i2c_driver).
  - module_pci_driver(struct pci_driver).
- No es una lista exhaustiva, solo un ejemplo.
- No solo hay macros para platform devices (existe una para USB por ejemplo).

Platform Devices

Platform Devices

- Una vez que se tiene el platform driver, se debe alimentar al kernel con dispositivos que requieran tal driver.
- Un platform device es representado en el kernel como una instancia de la estructura **struct platform_device**.
- Antes que suceda un match entre driver y device, los campos **name** de ambos deben coincidir.
- El miembro **num_resources** es el tamaño del array **struct resource *resource.**

Platform Devices

```
struct platform_device {
        const char *name;
        u32 id;
    struct device dev;
    u32 num_resources;
    struct resource *resource;
};
```

Platform Devices

- Debemos recordar que los platform devices por definición no son hot-pluggin.
- El sistema no tiene idea alguna de que están conectados al bus correspondiente hasta que trata de accederlos.
- Existen dos métodos de informar al Kernel sobre recursos y datos que el dispositivo necesita:
  - Kernel sin soporte de Device Tree (deprecado - no recomendable para nuevos desarrollos).
  - Device Tree.

Platform Devices

- En la primera opción (deprecada) para hacer un cambio cualquiera se requiere la re-compilación del kernel completo.
- Al incluir código específico para un hardware particular, el tamaño del mismo se incrementa.
- Por esto se introdujo el concepto de Device Tree.
- Ayuda a eliminar del Kernel el código específico y casi nunca testeado.
- El device tree es un archivo de descripción de hardware. Si cambia, solo se compila este.

Dispositivos, drivers y bus matching

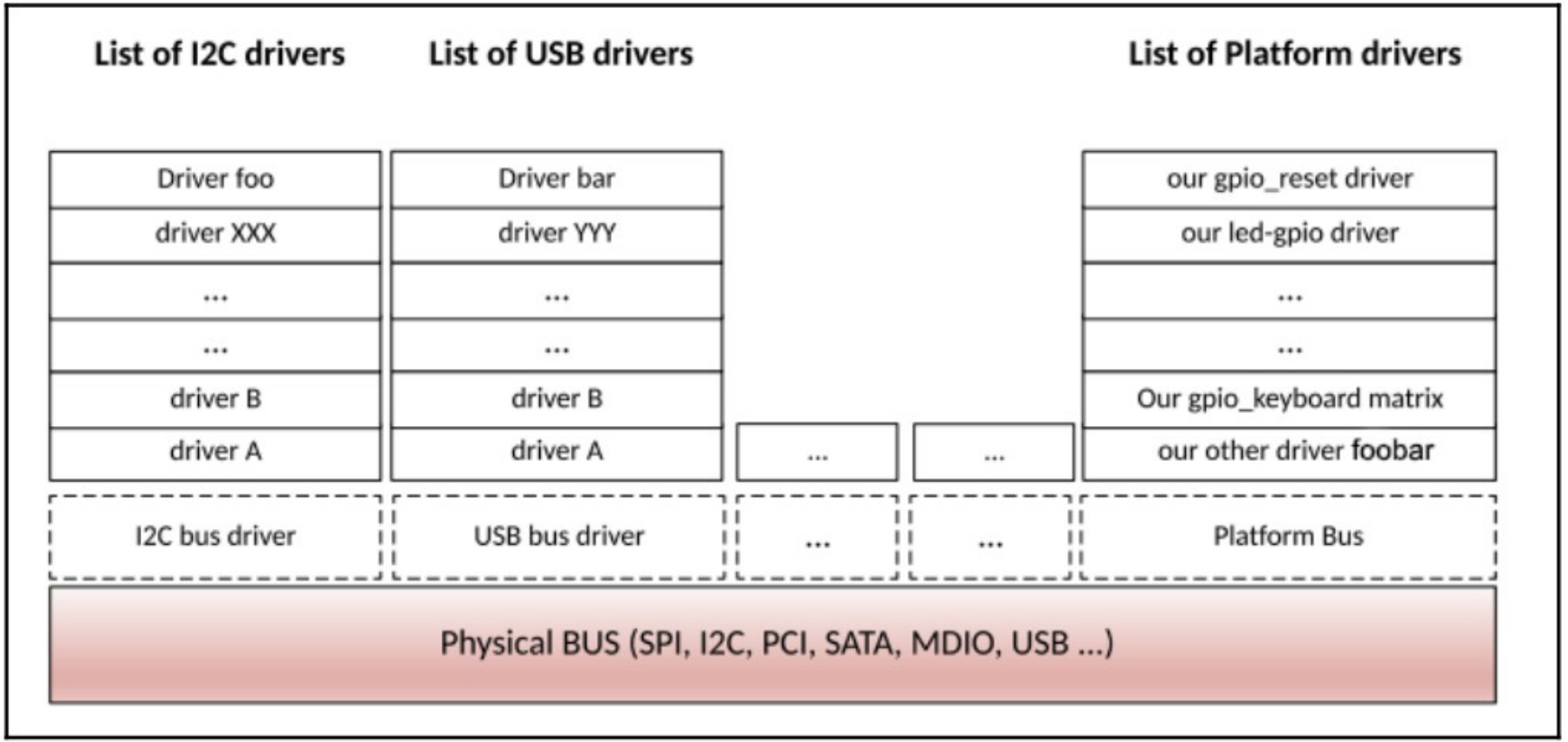
Dispositivos, drivers y bus matching

- Según el modelo de dispositivos de linux, el elemento que llamamos bus es el más importante.
- Cada bus mantiene una lista de drivers y dispositivos conectados a él.
- El responsable de conectar (match) drivers y dispositivos es el bus driver.
- Cada vez que se conecta un dispositivo nuevo al bus, o se agrega un driver, el matching loop comienza a ejecutarse.

Dispositivos, drivers y bus matching

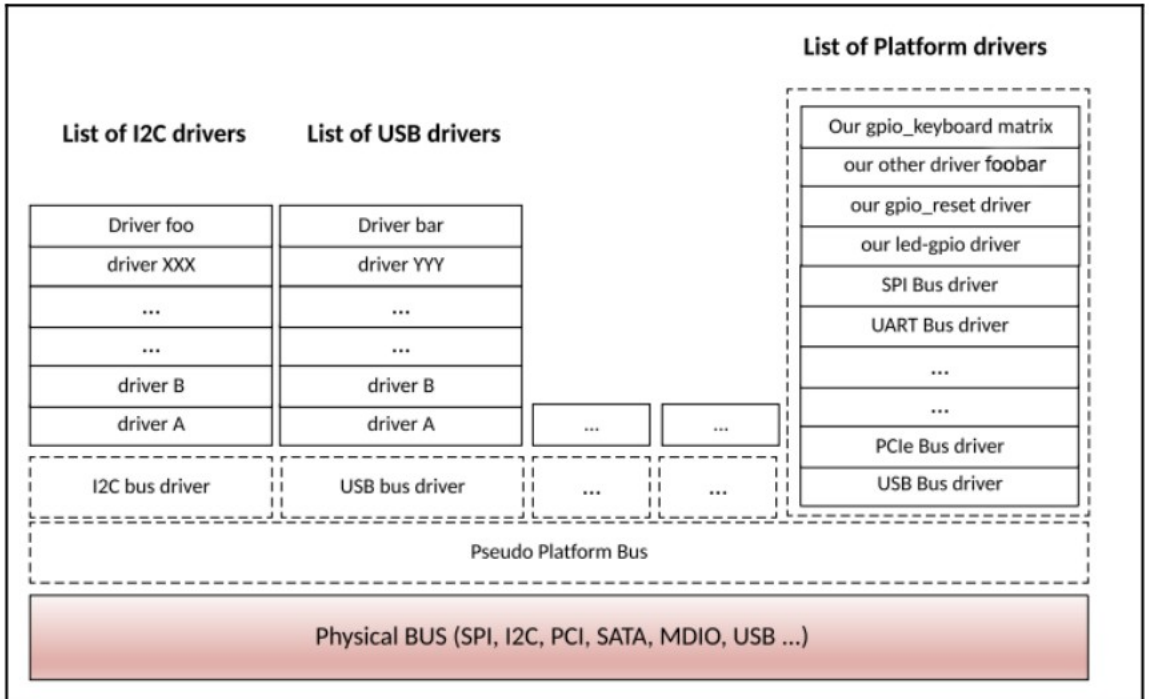
- Suponga que se registra un nuevo dispositivo I2C.
- El kernel lanza el matching loop llamando la función match del core I2C, registrada por el bus I2C.
- El loop verifica si hay algun driver ya registrado que pueda manejar el dispositivo que se acaba de registrar.
- En caso de un match, el kernel notifica al dev manager que carga el driver correspondiente, si es que no está en memoria.
- Al cargarse el driver, su función **probe()** se ejecuta inmediatamente.

Dispositivos, drivers y bus matching



Dispositivos, drivers y bus matching

- Cada driver registrado se monta sobre un bus, lo que genera un árbol. Actualizando la imagen anterior:



Dispositivos, drivers y bus matching

- La forma en que el kernel sabe cuáles dispositivos son manejados por qué drivers es a través de una macro.
- La macro **MODULE_DEVICE_TABLE** permite a un driver exponer su tabla ID.
- La tabla ID muestra los dispositivos soportados.
- Cuando el kernel debe efectuar un match, se recorre esta tabla.
- Para el caso de esta materia, se compara la entrada con el valor cargado bajo **compatible** en el DT.

Dispositivos, drivers y bus matching

- La macro **MODULE_DEVICE_TABLE** esta definida en el archivo **linux/module.h**

```
#define MODULE_DEVICE_TABLE(type, name)
```

- **type**: Puede ser i2c, spi, acpi, of, platform, usb, pci, o cualquier otro bus definido en **include/linux/mod_devicetable.h**
- **name:** es un puntero en el array **XXX_device_id** utilizado para device matching (ej: i2c_device_id).

Dispositivos, drivers y bus matching

- **IMPORTANTE**: En la actualidad no se utilizan estructuras de este estilo para dispositivos no detectables.
- Se utiliza el mecanismo de match Open Firmware utilizando el Device Tree (estructura **of_device_id).**
- Sin embargo, se siguen declarando tanto las estructuras **XX_device_id** como **of_device_id** para retro-compatibilidad.
- Ejemplo: El kernel 4.9 requiere ambas definiciones. Se actualizó en versiones posteriores.

Gracias.