Linux Platform device drivers

Individual Study Report ECGR 6890-030

Advisor: Dr. Hamed Tabhki

Sourabh Betigeri (*Author*)
ECE Department
William States Lee College of Engineering
sbetiger@uncc.edu

Sam Rogers (*Co-Author*)
ECE Department
William States Lee College of Engineering
srogers48@uncc.edu

Abstract—

Keywords—Linux Device drivers, platform drivers, hardware accelerators, ARM, AMBA Bus, GEM5

I. Introduction

In the very early days, Linux users often had to tell the kernel where specific devices were to be found before their systems would work. In the absence of this information, the driver could not know which I/O ports and interrupt line(s) the device was configured to use. Happily, we now live in the days of busses like PCI which have discoverability built into them; any device sitting on a PCI bus can tell the system what sort of device it is and where its resources are. So the kernel can, at boot time, enumerate the devices available and everything just works. Alas, life is not so simple; there are plenty of devices which are still not discoverable by the CPU. In the embedded and system-on-chip world, non-discoverable devices are, if anything are increasing in number. So the kernel still needs to provide ways to be told about the hardware that is actually present. "Platform devices" have long been used in this role in the kernel.

The Linux kernel runs on a wide range of architectures and hardware platforms, and therefore needs to maximize the reusability of code between platforms.

- For example, we want the same USB device driver to be usable on a x86 PC, or an ARM platform, even though the USB controllers used on these platforms are different.
- This requires a clean organization of the code, with the device drivers separated from the controller drivers , the hardware description separated from the drivers themselves, etc.
- This is what the Linux kernel Device Model allows, in addition to other advantages covered in this section.

II. Device Model in Linux kernel

In Linux, a driver is always interfacing with:

1. A framework that allows the driver to expose the hardware features in a generic way.
2. A bus infrastructure , part of the device model, to detect/communicate with the hardware.
3.

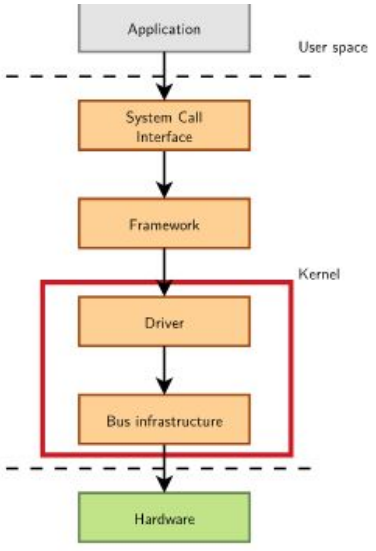

Figure: Linux Device Driver

II. a Device Model data structures

The device model is organized around three main data structures:

1. The struct bus_type structure, which represent one type of bus (USB, PCI, I2C, etc.)
2. The struct device_driver structure, which represents one driver capable of handling certain devices on a certain bus.
3. The struct device structure, which represents one device connected to a bus.

The kernel uses inheritance to create more specialized versions of struct device_driver and struct device for each bus subsystem.

III. BUS DRIVERS

The first component of the device model is the bus driver. One bus driver for each type of bus: USB, PCI, SPI, MMC, I2C, etc.

- It is responsible for registering the bus type (struct bus_type)
- Allowing the registration of adapter drivers (USB controllers, I2C adapters, etc.), able to detect the connected devices, and providing a communication mechanism with the devices
- Allowing the registration of device drivers (USB devices, I2C devices, PCI devices, etc.), managing the devices
- Matching the device drivers against the devices detected by the adapter drivers.
- Provides an API to both adapter drivers and device drivers
- Defining driver and device specific structures, mainly struct usb_driver

IV. PLATFORM DRIVERS

On embedded systems, devices are often not connected through a bus allowing enumeration, hotplugging, and providing unique identifiers for devices.

- For example, the devices on I2C buses or SPI buses, or the devices directly part of the system-on-chip.
- However, we still want all of these devices to be part of the device model.
- Such devices, instead of being dynamically detected, must be statically described in either the kernel source code or the device tree.

A. PLATFORM DEVICES

A platform device is represented by struct platform_device, which, like the rest of the relevant declarations, can be found in <linux/platform_device.h>. These devices are deemed to be connected to a virtual "platform bus"; drivers of platform devices must thus register themselves as such with the platform bus code. This registration is done by way of a platform_driver structure:

```
struct platform_driver {
        int (*probe)(struct platform_device *);
        int (*remove)(struct platform_device *);
        void (*shutdown)(struct platform_device *);
        int (*suspend)(struct platform_device *,
pm_message_t state);
        int (*resume)(struct platform_device *);
        struct device_driver driver;
        const struct platform_device_id *id_table;
};
```

At a minimum, the probe() and remove() callbacks must be supplied; the other callbacks have to do with power management and should be provided if they are relevant. The other thing the driver must provide is a way for the bus code to bind actual devices to the driver; there are two mechanisms which can be used for that purpose. The first is the id_table argument; the relevant structure is:

```
struct platform_device_id {
        char name[PLATFORM_NAME_SIZE];
        kernel_ulonsg_t driver_data;
};
```

If an ID table is present, the platform bus code will scan through it every time it has to find a driver for a new platform device. If the device's name matches the name in an ID table entry, the device will be given to the driver for management; a pointer to the matching ID

table entry will be made available to the driver as well. This happens, though, most platform drivers do not provide an ID table at all; they simply provide a name for the driver itself in the driver field.

```
static struct platform_driver hwAcc_drv = {
  .probe = hwAcc_probe,
  .remove = hwAcc_remove,
  .driver = {
          .name = DEVICE_NAME,
          .owner = THIS_MODULE,
          .of_match_table = of_match_ptr(my_of_ids)
          }
};
```

B. DRIVER REGISTRATION

Platform drivers make themselves known to the kernel with:

```
int platform_driver_register(struct platform_driver
*driver);
```

As soon as this call succeeds, the driver's probe() function can be called with new devices. That function gets as an argument a platform_device pointer describing the device to be instantiated:

```
  struct platform_device {
          const char        *name;
          int               id;
          struct device     dev;
          u32               num_resources;
          struct resource   *resource;
          const struct platform_device_id  *id_entry;
          /* Others omitted */
   };
```

The dev structure can be used in contexts where it is needed - the DMA mapping API, for example. If the device was matched using an ID table entry, id_entry will point to the specific entry matched. The resource array can be used to learn where various resources, including memory-mapped I/O registers and interrupt lines, can be found. There are a number of helper functions for getting data out of the resource array; these include:

- struct resource *platform_get_resource(struct platform_device *pdev, unsigned int type, unsigned int n);

- struct resource *platform_get_resource_byname(struct platform_device *pdev, unsigned int type, const char *name);
  int platform_get_irq(struct platform_device *pdev, unsigned int n);

The "n" parameter says which resource of that type is desired, with zero indicating the first one. Thus, for example, a driver could find its second MMIO region with:

```
r = platform_get_resource(pdev, IORESOURCE_MEM,
1);
```

Assuming the probe() function finds the information it needs, it should verify the device's existence to the extent possible, register the "real" devices associated with the platform device, and return zero.

V. DEVICE TREES

On many embedded architectures, manual instantiation of platform devices was considered to be too verbose and not easily maintainable. Such architectures are moving, or have moved, to use the device tree. It is a tree of nodes that models the hierarchy of devices in the system, from the devices inside the processor to the devices on the board. Each node can have a number of properties describing various properties of the devices: addresses, interrupts, clocks, etc. At boot time, the kernel is given a compiled version, the Device Tree Blob, which is parsed to instantiate all the devices described in the DT. On ARM, they are located in arch/arm/boot/dts/.

DEVICE TREE EXAMPLE:

```
acc: hwAcc@101f9000 {
        compatible = "gem5,hwAcc";
        reg = <0x0 0x101f9000 0x0 0x8>;
        interrupt-parent = <&vic>;
        interrupts = <16>;
        };
```

- hwAcc@101f9000 is the node name
- acc is a label, that can be referred to in other parts of the DT as &acc..
- Other lines are properties. Their values are usually strings, list of integers, or references to other nodes.

V.A. DEVICE TREE OVERLAYS

Each particular hardware platform has its own device tree. However, several hardware platforms use the same processor, and often various processors in the same family share a number of similarities. To allow this, a device tree file can include another one. The trees described by the including file overlays the tree described by the included file. This can be done:

Either by using the /include/ statement provided by the Device Tree language. Either by using the #include statement, which requires calling the C preprocessor before parsing the Device Tree.
Linux currently uses either one technique or the other.

V.B. Device Tree: compatible string

- With the device tree, a device is bound to the corresponding driver using the compatible string.
- The of_match_table field of struct device_driver lists the compatible strings supported by the driver.

.

```
#if defined(CONFIG_OF)
/* string to match :- of_device_match */
static const struct of_device_id my_of_ids[] = {
 { .compatible = "gem5,hwAcc" },
  { }
};

#endif
```

V.c Device Tree Resources

The drivers will use the same mechanism that we saw previously to retrieve basic information: interrupts numbers, physical addresses, etc.

The available resources list will be built up by the kernel at boot time from the device tree, so that you don't need to make any unnecessary lookups to the DT when loading your driver. Any additional information will be specific to a driver or the class it belongs to, defining the bindings

V. d Device Tree bindings

The compatible string and the associated properties define what is called a device tree binding. Device tree bindings are all documented in Documentation /devicetree/bindings. Since the Device Tree is normally part of the kernel ABI, the
bindings must remain compatible over time.

- A new kernel must be capable of using an old Device Tree.
- This requires a very careful design of the bindings.
- A Device Tree binding should contain only a description of the hardware and not configuration.
- An interrupt number can be part of the device tree as it describes the hardware. But not whether DMA should be used for a device or not, as it is a configuration choice.

VI. PLATFORM DRIVERS WITH DEVICE TREES

Older mechanism was to use "board files," each of which describes a single type of board. Kernels are typically built around a single board file and cannot boot on any other type of system. Board files sort of worked when there were relatively small numbers of embedded system types to deal with. Now Linux-based embedded systems are everywhere, architectures which have typically depended on board files (ARM, in particular) are finding their way into more types of systems, and the whole scheme looks poised to collapse under its own weight.

The solution implemented is device trees; in essence, a device tree is a textual description of a specific system's hardware configuration. The device tree is passed to the kernel at boot time; the kernel then reads through it to learn about what kind of system it is actually running on. With luck, device trees will abstract the differences between systems into boot-time data and allow generic kernels to run on a much wider variety of hardware.

If the device tree includes a platform device (where such devices, in the device tree context, are those which are direct children of the root or are attached to a "simple bus"), that device will be instantiated and matched against a driver. The memory-mapped I/O and interrupt resources will be marshalled from the device tree description and made available to the device's probe() function in the usual way. The driver need not know that the device was instantiated out of a device tree rather than from a hard-coded platform device definition.

The kernel provides an of_device_id structure which can be used for this purpose:

```
static const struct of_device_id my_of_ids[] = {
        { .compatible = "long,funky-device-tree-name"
},
        { }
   };
```

When the platform driver is declared, it stores a pointer to this table in the driver substructure:

```
  static struct platform_driver my_driver = {
        /* ... */
        .driver  = {
                .name = "hwAcc_drv",
                .of_match_table = my_of_ids
        }
  };
```

The driver can also declare the ID table as a device table to enable autoloading of the module as the device tree is instantiated:

```
MODULE_DEVICE_TABLE(of, my_of_ids);
```

The one other thing capable of complicating the situation is platform data. Needless to say, the device tree code is unaware of the specific structure used by a given driver for its platform data, so it will be unable to provide that information in that form. On the other hand, the device tree mechanism is equipped to allow the passing of just about any information that the driver may need to know. Making use of that information will require the driver to become a bit more aware of the device tree subsystem, though.

Drivers expecting platform data should check the dev.platform_data pointer in the usual way. If there is a non-null value there, the driver has been instantiated in the traditional way and device tree does not enter into the picture; the platform data should be used in the usual way. If, however, the driver has been instantiated from the device tree code, the platform_data pointer will be null, indicating that the information must be acquired from the device tree directly.
In this case, the driver will find a device_node pointer in the platform devices dev.of_node field. The various device tree access functions (of_get_property(), primarily) can then be used to extract the needed information from the device tree.

It is mostly a matter of getting the right names in place so that the binding between a device tree node and the driver can be made, with a bit of additional work required in cases where platform data is in use. The nice result is that the static platform_device declarations can go away, along with the board files that contain them. That should, eventually, allow the removal of a bunch of boilerplate code from the kernel while simultaneously making the kernel more flexible.

VII. INTERRUPT MANAGEMENT

A. REGISTERING AN INTERRUPT HANDLER

The managed API is recommended:

*int devm_request_irq(struct device *dev,*
unsigned int irq,
irq_handler_t handler,
unsigned long irq_flags,
*const char *devname,*
*void *dev_id);*

- device *dev: device for automatic freeing at device or module release time.
- irq: requested IRQ channel. For platform devices, use platform_get_irq() to retrieve the interrupt number.
- handler is a pointer to the IRQ handler
- irq_flags are option masks (see next slide)
- devname is the registered name (for /proc/interrupts)
- dev_id is an opaque pointer. It can typically be used to pass a pointer to a per-device data structure. It cannot be NULL as it is used as an identifier for freeing interrupts on a shared line.

B. RELEASING AN INTERRUPT HANDLER

*void devm_free_irq(struct device *dev,*
unsigned int irq,
*void *dev_id);*

Explicitly release an interrupt handler. Done automatically in normal situations. Defined in include/linux/interrupt.h

C. INTERRUPT HANDLER PROTOTYPE

irqreturn_t foo_interrupt(int irq,
*void *dev_id);*

- irq, the IRQ number
- dev_id, the per-device pointer that was passed to devm_request_irq()

Return value

IRQ_HANDLED: recognized and handled interrupt
IRQ_NONE: used by the kernel to detect spurious interrupts, and disable the interrupt line if none of the interrupt handlers has handled the interrupt.

- Acknowledge the interrupt to the device (otherwise no more interrupts will be generated, or the interrupt will keep firing over and over again).
- Read/write data from/to the device.
- Wake up any process waiting for such data, typically on a per-device wait queue:

wake_up_interruptible(&device_queue);

VIII. CONCLUSION

This project is a part of a bigger goal to integrate custom hardware accelerators to gem5 simulator. Adding device driver support to receive interrupts serves the larger purpose of including custom hardware built to include as a device into gem5. The driver can always extended to include device specific data and device specific controls.

X. REFERENCES

1. https://www.kernel.org/doc/
2. https://www.codeproject.com/Tips/1080177/Linux-Platform-Device-Driver
3. https://lwn.net/Articles/448502/
4. www.freeelectrons.com
5. https://kerneltweaks.wordpress.com/2014/03/30/platform-device-and-platform-driver-linux/
6.