The relationship between device tree, device and driver in linux driver

原创 Yuan Hailu Posted at 2020-03-29 15:44:02 2634 Favorite 32 copyright

Category column: Linux kernel and drivers underlying driver

The relationship between device tree , device and driver in linux driver

The general form of a driver
1. platform_driver
2. platform_device

The general form of a driver

Here is a simple driver skeleton:

```
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/mm.h>
#include <linux/module.h>
#include <linux/delay.h>
#include <linux/bcd.h>
#include <linux/mutex.h>
#include <linux/uaccess.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/device.h>
#include <linux/platform_device.h>
static int  my_test_probe(struct platform_device *pdev)
{
    printk("my_test_probe\r\n");

    printk("%s\r\n",pdev->name);


    return 0;
}

static int my_test_remove(struct platform_device *pdev)
{
   return 0;
}


static const struct of_device_id my_test_id[] = {
        {
                .compatible = "test_device_id",
        },
        {}};
MODULE_DEVICE_TABLE(of, my_test_id);

static struct platform_driver my_test_driver = {
        .driver = {
                .name = "my_test",
                .of_match_table = my_test_id,
        },
        .probe = my_test_probe,
        .remove = my_test_remove,
};


```

```
42
43  static int my_test_init(void)
44  {
45      printk("my_test_init\r\n");
46      platform_driver_register(&my_test_driver);
47      return 0;
48  }
49
50  static void my_test_exit(void)
51  {
52       printk("my_test_exit\r\n");
53       platform_driver_unregister(&my_test_driver);
54  }
55
56  module_init(my_test_init);
57  module_exit(my_test_exit);
58
59  MODULE_DESCRIPTION("MY test");
60  MODULE_AUTHOR("yhl");
61  MODULE_LICENSE("GPL");
```

From the code, we find the following key information:

1. platform_driver

Find the definition of the structure:

```
1
2  struct platform_driver {
3          int (*probe)(struct platform_device *);
4          int (*remove)(struct platform_device *);
5          void (*shutdown)(struct platform_device *);
6          int (*suspend)(struct platform_device *, pm_message_t state);
7          int (*resume)(struct platform_device *);
8          struct device_driver driver;
9          const struct platform_device_id *id_table;
10         bool prevent_deferred_probe;
   };
```

Among them, the probe needs to be implemented by the driver, and when the driver and device match, the function will be executed. Devices and drivers are on both sides of the bus, and are matched by the bus.

```
1
2  struct bus_type platform_bus_type = {
3          .name           = "platform",
4          .dev_groups     = platform_dev_groups,
5          .match          = platform_match,
6          .uevent         = platform_uevent,
7          .dma_configure  = platform_dma_configure,
8          .pm             = &platform_dev_pm_ops,
   };
```

The total.match member realizes the matching between the bus and the device:

```
1
2  static int platform_match(struct device *dev, struct device_driver *drv)
3  {
4          struct platform_device *pdev = to_platform_device(dev);
5          struct platform_driver *pdrv = to_platform_driver(drv);
6
7          /* When driver_override is set, only bind to the matching driver */
8          if (pdev->driver_override)
9                  return !strcmp(pdev->driver_override, drv->name);
10
           /* Attempt an OF style match first */
```

```
        if (of_driver_match_device(dev, drv))
                return 1;

        /* Then try ACPI style match */
        if (acpi_driver_match_device(dev, drv))
                return 1;

        /* Then try to match against the id table */
        if (pdrv->id_table)
                return platform_match_id(pdrv->id_table, pdev) != NULL;
twen
twen    /* fall-back to driver name match */
twen    return (strcmp(pdev->name, drv->name) == 0);
twen }
```

Since we are implementing device with device tree, we need to pay attention to of_driver_match_device();

```
static inline int of_driver_match_device(struct device *dev,
                                         const struct device_driver *drv)
{
        return of_match_device(drv->of_match_table, dev) != NULL;
}
```

```
static
const struct of_device_id *__of_match_node(const struct of_device_id *matches,
                                           const struct device_node *node)
{
        const struct of_device_id *best_match = NULL;
        int score, best_score = 0;

        if (!matches)
                return NULL;

        for (; matches->name[0] || matches->type[0] || matches->compatible[0]; matches++) {
                score = __of_device_is_compatible(node, matches->compatible,
                                                  matches->type, matches->name);
                if (score > best_score) {
                        best_match = matches;
                        best_score = score;
                }
        }

        return best_match;
}
```

```
```c
static int __of_device_is_compatible(const struct device_node *device,
                                     const char *compat, const char *type, const char *name)
{
        struct property *prop;
        const char *cp;
        int index = 0, score = 0;

        /* Compatible match has highest priority */
        if (compat && compat[0]) {
                prop = __of_find_property(device, "compatible", NULL);//找到compatible节点
                for (cp = of_prop_next_string(prop, NULL); cp;
                     cp = of_prop_next_string(prop, cp), index++) {
                        if (of_compat_cmp(cp, compat, strlen(compat)) == 0) {
```
```

```
                                score = INT_MAX/2 - (index << 2);
                                break;
                        }
                }
                if (!score)
                        return 0;
        }

        /* Matching type is better than matching name */
        if (type && type[0]) {
                if (!__of_node_is_type(device, type))
                        return 0;
                score += 2;
        }

        /* Matching name is a bit better than not */
        if (name && name[0]) {
                if (!of_node_name_eq(device, name))
                        return 0;
                score++;
        }

        return score;
}
```

Eventually, it will be resolved to the compatible attribute node of the device tree.

From the above analysis, we can know that when .of_match_table = my_test_id matches the compatible = ""; node in the device tree, it means that the device and driver match, and the probe function will be called back.

You can see the registered devices in the system in /sys/bus/platform:

```
[0@root platform]# cd devices/
[0@root devices]# ls
200.rstc                    f8005000.watchdog
700.pinctrl                 f8006000.memory-controller
Fixed MDIO bus.0            f8007000.devcfg
amba                        f8007100.adc
ci_hdrc.0                   f8891000.pmu
cpuidle-zynq.0              f8f00200.timer
e0001000.serial             f8f00600.timer
e0002000.usb                f8f02000.cache-controller
e0004000.i2c                fixedregulator
e000a000.gpio               fpga-full
e000b000.ethernet           mytest@0
e0100000.mmc                phy0
f8000000.slcr               reg-dummy
f8001000.timer              snd-soc-dummy
f8002000.timer
```


You can also check the registered drivers in /sys/bus/platform:

```
[0@root drivers]# ls
Xilinx Watchdog       of-fpga-region        vexpress-syscfg
alarmtimer            of_fixed_clk          vexpress-sysreg
armv7-pmu             of_fixed_factor_clk   xadc
axi-i2s               physmap-flash         xilinx-pcie
axi-spdif             pwrseq_emmc           xilinx-tpg
basic-mmio-gpio       pwrseq_simple         xilinx-udc
cdns-i2c              reg-dummy             xilinx-vdma
cdns-spi              reg-fixed-voltage     xilinx-video
cdns-wdt              reset_zynq            xilinx-vtc
chipidea-usb2         sdhci-arasan          xilinx_axienet
ci_hdrc               snd-soc-dummy         xilinx_can
cpuidle-zynq          soc-audio             xilinx_emaclite
gpio-clk              synopsys-edac         xilinx_spi
gpio-keys             syscon                xlnx_pr_decoupler
gpio-keys-polled      tegra-udc             xuartps
gpio-xilinx           uio_pdrv_genirq       zevio_usb
imx_usb               usb_phy_generic       zynq-gpio
leds-gpio             usbmisc_imx           zynq-pinctrl
macb                  vexpress-osc          zynq-qspi
msm_hsusb             vexpress-reset        zynq_fpga_manager
```


Of course, there are many buses under /sys/bus, some of which are actual The hardware bus, platfrom is the software virtual bus in the linux operating system.

2. platform_device

This structure appears here in my_test_probe(struct platform_device *pdev). When the probe function is executed, which parameters can be obtained/modified from this parameter? Track it in and see:

```
struct platform_device {
        const char      *name;
        int             id;
        bool            id_auto;
        struct device   dev;
        u64             dma_mask;
        u32             num_resources;
        struct resource *resource;

        const struct platform_device_id *id_entry;
        char *driver_override; /* Driver name to force a match */

        /* MFD cell pointer */
        struct mfd_cell *mfd_cell;

        /* arch specific additions */
        struct pdev_archdata    archdata;
};
```

Another key member struct device dev;

```

struct device {
        struct kobject kobj;      //相当于内核的基类
        struct device           *parent;  //用于实现链表

        struct device_private   *p;  //私有数据

        const char              *init_name; /* initial name of the device */
        const struct device_type *type;                             //设备类型

        struct bus_type *bus;           /* type of bus device is on */   //总线类型
        struct device_driver *driver;   /* which driver has allocated this
                                           device */
        void            *platform_data; /* Platform specific data, device
                                           core doesn't touch it */
        void            *driver_data;   /* Driver data, set and get with
                                           dev_set_drvdata/dev_get_drvdata */
#ifdef CONFIG_PROVE_LOCKING
        struct mutex            lockdep_mutex;
#endif
        struct mutex            mutex;  /* mutex to synchronize calls to
                                         * its driver.
                                         */

        struct dev_links_info   links;
        struct dev_pm_info      power;
        struct dev_pm_domain    *pm_domain;

#ifdef CONFIG_GENERIC_MSI_IRQ_DOMAIN
        struct irq_domain       *msi_domain;
#endif
#ifdef CONFIG_PINCTRL
        struct dev_pin_info     *pins;
#endif
#ifdef CONFIG_GENERIC_MSI_IRQ
        struct list_head        msi_list;
#endif

        const struct dma_map_ops *dma_ops;
        u64             *dma_mask;      /* dma mask (if dma'able device) */

```

```
        u64             coherent_dma_mask;/* Like dma_mask, but for
                                             alloc_coherent mappings as
                                             not all hardware supports
                                             64 bit addresses for consistent
                                             allocations such descriptors. */
        u64             bus_dma_limit;  /* upstream dma constraint */
        unsigned long   dma_pfn_offset;

        struct device_dma_parameters *dma_parms;

        struct list_head        dma_pools;      /* dma pools (if dma'ble) */

#ifdef CONFIG_DMA_DECLARE_COHERENT
        struct dma_coherent_mem *dma_mem; /* internal for coherent mem
                                             override */
#endif
#ifdef CONFIG_DMA_CMA
        struct cma *cma_area;           /* contiguous memory area for dma
                                           allocations */
#endif
        /* arch specific additions */
        struct dev_archdata     archdata;

        struct device_node      *of_node; /* associated device tree node */
        struct fwnode_handle    *fwnode; /* firmware device node */

#ifdef CONFIG_NUMA
        int             numa_node;      /* NUMA node this device is close to */
#endif
        dev_t                   devt;   /* dev_t, creates the sysfs "dev" */
        u32                     id;     /* device instance */

        spinlock_t              devres_lock;
        struct list_head        devres_head;

        struct class            *class;
        const struct attribute_group **groups;  /* optional groups */

        void    (*release)(struct device *dev);
        struct iommu_group      *iommu_group;
        struct iommu_fwspec     *iommu_fwspec;
        struct iommu_param      *iommu_param;

        bool                    offline_disabled:1;
        bool                    offline:1;
        bool                    of_node_reused:1;
        bool                    state_synced:1;
#if defined(CONFIG_ARCH_HAS_SYNC_DMA_FOR_DEVICE) || \
    defined(CONFIG_ARCH_HAS_SYNC_DMA_FOR_CPU) || \
    defined(CONFIG_ARCH_HAS_SYNC_DMA_FOR_CPU_ALL)
        bool                    dma_coherent:1;
#endif
};

```

Here again kobject is involved.

The most direct interaction between kobject and us is the /sys directory.

```
struct kobject {
        const char              *name;
        struct list_head        entry;
        struct kobject          *parent;
        struct kset             *kset;
        struct kobj_type        *ktype;
        struct kernfs_node      *sd; /* sysfs directory entry */
        struct kref             kref;
#ifdef CONFIG_DEBUG_KOBJECT_RELEASE
        struct delayed_work     release;
```

```
#endif
        unsigned int state_initialized:1;
        unsigned int state_in_sysfs:1;
        unsigned int state_add_uevent_sent:1;
        unsigned int state_remove_uevent_sent:1;
        unsigned int uevent_suppress:1;
};
```

The knowledge points of the article are matched with the official knowledge files, and relevant knowledge can be further learned