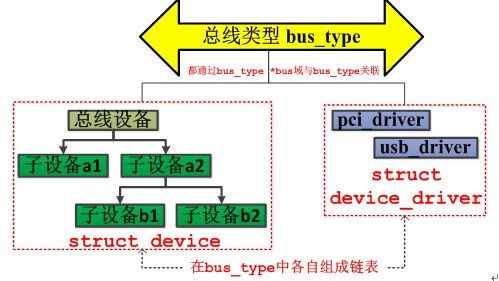
android驱动调试

andorid的驱动都是基于Linux的设备模型

linux设备模型的基本概念

linux设备模型由总线(bus),类(class),设备(device)和驱动(driver)组成.

其架构如下:



Bus(总线):

Linux认为（可以参考include/linux/device.h中struct bus\_type的注释），总线是CPU和一个或多个设备之间信息交互的通道。而为了方便设备模型的抽象，所有的设备都应连接到总线上（无论是CPU内部总线、还是虚拟的总线“platform Bus”）.

/\*\*

\* struct bus\_type - The bus type of the device

\*

\* @name: The name of the bus.

\* @dev\_name: Used for subsystems to enumerate devices like ("foo%u", dev->id).

\* @dev\_root: Default device to use as the parent.

\* @dev\_attrs: Default attributes of the devices on the bus.

\* @bus\_groups: Default attributes of the bus.

\* @dev\_groups: Default attributes of the devices on the bus.

\* @drv\_groups: Default attributes of the device drivers on the bus.

\* @match: Called, perhaps multiple times, whenever a new device or driver

\* is added for this bus. It should return a positive value if the

\* given device can be handled by the given driver and zero

\* otherwise. It may also return error code if determining that

\* the driver supports the device is not possible. In case of

\* -EPROBE\_DEFER it will queue the device for deferred probing.

\* @uevent: Called when a device is added, removed, or a few other things

\* that generate uevents to add the environment variables.

\* @probe: Called when a new device or driver add to this bus, and callback

\* the specific driver's probe to initial the matched device.

\* @remove: Called when a device removed from this bus.

\* @shutdown: Called at shut-down time to quiesce the device.

\*

\* @online: Called to put the device back online (after offlining it).

\* @offline: Called to put the device offline for hot-removal. May fail.

\*

\* @suspend: Called when a device on this bus wants to go to sleep mode.

\* @resume: Called to bring a device on this bus out of sleep mode.

\* @pm: Power management operations of this bus, callback the specific

\* device driver's pm-ops.

\* @iommu\_ops: IOMMU specific operations for this bus, used to attach IOMMU

\* driver implementations to a bus and allow the driver to do

\* bus-specific setup

\* @p: The private data of the driver core, only the driver core can

\* touch this.

\* @lock\_key: Lock class key for use by the lock validator

\*

\* A bus is a channel between the processor and one or more devices. For the

\* purposes of the device model, all devices are connected via a bus, even if

\* it is an internal, virtual, "platform" bus. Buses can plug into each other.

\* A USB controller is usually a PCI device, for example. The device model

\* represents the actual connections between buses and the devices they control.

\* A bus is represented by the bus\_type structure. It contains the name, the

\* default attributes, the bus' methods, PM operations, and the driver core's

\* private data.

\*/

struct bus\_type {

const char \*name;

const char \*dev\_name;

struct device \*dev\_root;

struct device\_attribute \*dev\_attrs; /\* use dev\_groups instead \*/

const struct attribute\_group \*\*bus\_groups;

const struct attribute\_group \*\*dev\_groups;

const struct attribute\_group \*\*drv\_groups;

int (\*match)(struct device \*dev, struct device\_driver \*drv);

int (\*uevent)(struct device \*dev, struct kobj\_uevent\_env \*env);

int (\*probe)(struct device \*dev);

int (\*remove)(struct device \*dev);

void (\*shutdown)(struct device \*dev);

int (\*online)(struct device \*dev);

int (\*offline)(struct device \*dev);

int (\*suspend)(struct device \*dev, pm\_message\_t state);

int (\*resume)(struct device \*dev);

const struct dev\_pm\_ops \*pm;

const struct iommu\_ops \*iommu\_ops;

struct subsys\_private \*p;

struct lock\_class\_key lock\_key;

};

Class（分类）：在Linux设备模型中，Class的概念非常类似面向对象程序设计中的Class（类），它主要是集合具有相似功能或属性的设备，这样就可以抽象出一套可以在多个设备之间共用的数据结构和接口函数。因而从属于相同Class的设备的驱动程序，就不再需要重复定义这些公共资源，直接从Class中继承即可。

/\*\*

\* struct class - device classes

\* @name: Name of the class.

\* @owner: The module owner.

\* @class\_attrs: Default attributes of this class.

\* @dev\_groups: Default attributes of the devices that belong to the class.

\* @dev\_kobj: The kobject that represents this class and links it into the hierarchy.

\* @dev\_uevent: Called when a device is added, removed from this class, or a

\* few other things that generate uevents to add the environment

\* variables.

\* @devnode: Callback to provide the devtmpfs.

\* @class\_release: Called to release this class.

\* @dev\_release: Called to release the device.

\* @suspend: Used to put the device to sleep mode, usually to a low power

\* state.

\* @resume: Used to bring the device from the sleep mode.

\* @shutdown: Called at shut-down time to quiesce the device.

\* @ns\_type: Callbacks so sysfs can detemine namespaces.

\* @namespace: Namespace of the device belongs to this class.

\* @pm: The default device power management operations of this class.

\* @p: The private data of the driver core, no one other than the

\* driver core can touch this.

\*

\* A class is a higher-level view of a device that abstracts out low-level

\* implementation details. Drivers may see a SCSI disk or an ATA disk, but,

\* at the class level, they are all simply disks. Classes allow user space

\* to work with devices based on what they do, rather than how they are

\* connected or how they work.

\*/

struct class {

const char \*name;

struct module \*owner;

struct class\_attribute \*class\_attrs;

const struct attribute\_group \*\*dev\_groups;

struct kobject \*dev\_kobj;

int (\*dev\_uevent)(struct device \*dev, struct kobj\_uevent\_env \*env);

char \*(\*devnode)(struct device \*dev, umode\_t \*mode);

void (\*class\_release)(struct class \*class);

void (\*dev\_release)(struct device \*dev);

int (\*suspend)(struct device \*dev, pm\_message\_t state);

int (\*resume)(struct device \*dev);

int (\*shutdown)(struct device \*dev);

const struct kobj\_ns\_type\_operations \*ns\_type;

const void \*(\*namespace)(struct device \*dev);

const struct dev\_pm\_ops \*pm;

struct subsys\_private \*p;

};

Device（设备）：抽象系统中所有的硬件设备，描述它的名字、属性、从属的Bus、从属的Class等信息。

/\*\*

\* struct device - The basic device structure

\* @parent: The device's "parent" device, the device to which it is attached.

\* In most cases, a parent device is some sort of bus or host

\* controller. If parent is NULL, the device, is a top-level device,

\* which is not usually what you want.

\* @p: Holds the private data of the driver core portions of the device.

\* See the comment of the struct device\_private for detail.

\* @kobj: A top-level, abstract class from which other classes are derived.

\* @init\_name: Initial name of the device.

\* @type: The type of device.

\* This identifies the device type and carries type-specific

\* information.

\* @mutex: Mutex to synchronize calls to its driver.

\* @bus: Type of bus device is on.

\* @driver: Which driver has allocated this

\* @platform\_data: Platform data specific to the device.

\* Example: For devices on custom boards, as typical of embedded

\* and SOC based hardware, Linux often uses platform\_data to point

\* to board-specific structures describing devices and how they

\* are wired. That can include what ports are available, chip

\* variants, which GPIO pins act in what additional roles, and so

\* on. This shrinks the "Board Support Packages" (BSPs) and

\* minimizes board-specific #ifdefs in drivers.

\* @driver\_data: Private pointer for driver specific info.

\* @power: For device power management.

\* See Documentation/power/devices.txt for details.

\* @pm\_domain: Provide callbacks that are executed during system suspend,

\* hibernation, system resume and during runtime PM transitions

\* along with subsystem-level and driver-level callbacks.

\* @pins: For device pin management.

\* See Documentation/pinctrl.txt for details.

\* @msi\_list: Hosts MSI descriptors

\* @msi\_domain: The generic MSI domain this device is using.

\* @numa\_node: NUMA node this device is close to.

\* @dma\_mask: Dma mask (if dma'ble device).

\* @coherent\_dma\_mask: Like dma\_mask, but for alloc\_coherent mapping as not all

\* hardware supports 64-bit addresses for consistent allocations

\* such descriptors.

\* @dma\_pfn\_offset: offset of DMA memory range relatively of RAM

\* @dma\_parms: A low level driver may set these to teach IOMMU code about

\* segment limitations.

\* @dma\_pools: Dma pools (if dma'ble device).

\* @dma\_mem: Internal for coherent mem override.

\* @cma\_area: Contiguous memory area for dma allocations

\* @archdata: For arch-specific additions.

\* @of\_node: Associated device tree node.

\* @fwnode: Associated device node supplied by platform firmware.

\* @devt: For creating the sysfs "dev".

\* @id: device instance

\* @devres\_lock: Spinlock to protect the resource of the device.

\* @devres\_head: The resources list of the device.

\* @knode\_class: The node used to add the device to the class list.

\* @class: The class of the device.

\* @groups: Optional attribute groups.

\* @release: Callback to free the device after all references have

\* gone away. This should be set by the allocator of the

\* device (i.e. the bus driver that discovered the device).

\* @iommu\_group: IOMMU group the device belongs to.

\* @iommu\_fwspec: IOMMU-specific properties supplied by firmware.

\*

\* @offline\_disabled: If set, the device is permanently online.

\* @offline: Set after successful invocation of bus type's .offline().

\*

\* At the lowest level, every device in a Linux system is represented by an

\* instance of struct device. The device structure contains the information

\* that the device model core needs to model the system. Most subsystems,

\* however, track additional information about the devices they host. As a

\* result, it is rare for devices to be represented by bare device structures;

\* instead, that structure, like kobject structures, is usually embedded within

\* a higher-level representation of the device.

\*/

struct device {

struct device \*parent;

struct device\_private \*p;

struct kobject kobj;

const char \*init\_name; /\* initial name of the device \*/

const struct device\_type \*type;

struct mutex mutex; /\* mutex to synchronize calls to

\* its driver.

\*/

struct bus\_type \*bus; /\* type of bus device is on \*/

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/get\_drvdata \*/

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

#ifdef CONFIG\_NUMA

int numa\_node; /\* NUMA node this device is close to \*/

#endif

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. \*/

unsigned long dma\_pfn\_offset;

struct device\_dma\_parameters \*dma\_parms;

struct list\_head dma\_pools; /\* dma pools (if dma'ble) \*/

struct dma\_coherent\_mem \*dma\_mem; /\* internal for coherent mem

override \*/

#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 \*/

dev\_t devt; /\* dev\_t, creates the sysfs "dev" \*/

u32 id; /\* device instance \*/

spinlock\_t devres\_lock;

struct list\_head devres\_head;

struct klist\_node knode\_class;

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;

bool offline\_disabled:1;

bool offline:1;

};

案例分析

以触摸屏gt1x驱动为例:

驱动设备的设备树:

&i2c\_2 { /\*i2c2\*/ //设备挂载在总线i2c 2上面

gt1x@14 { //设备的硬件设置

compatible = "goodix,gt1x";

reg = <0x14>;

goodix,rst-gpio = <&gpio 165 0>;

goodix,irq-gpio = <&gpio 166 0>;

vdd\_ana-supply = <&dldo8>;

vcc\_i2c-supply = <&aldo10>;

pinctrl-names = "i2c\_mode", "gpio\_mode";

pinctrl-0 = <&gt1x\_i2c\_mode>;

pinctrl-1 = <&gt1x\_gpio\_mode>;

};

};

驱动文件:

static int \_\_init gt1x\_ts\_init(void)

{

GTP\_DEBUG\_FUNC();

GTP\_INFO("GTP driver installing...");

gt1x\_wq = create\_singlethread\_workqueue("gt1x\_wq");

if (!gt1x\_wq) {

GTP\_ERROR("Creat workqueue failed.");

return -ENOMEM;

}

//i2c设备驱动

return i2c\_add\_driver(&gt1x\_ts\_driver);

}

//设备驱动

static struct i2c\_driver gt1x\_ts\_driver = {

.probe = gt1x\_ts\_probe,

.remove = gt1x\_ts\_remove,

.id\_table = gt1x\_ts\_id,

.driver = {

.name = GTP\_I2C\_NAME,

.owner = THIS\_MODULE,

#ifdef GTP\_CONFIG\_OF

.of\_match\_table = gt1x\_match\_table,

#endif

#if !defined(CONFIG\_FB) && defined(CONFIG\_PM)

.pm = &gt1x\_ts\_pm\_ops,

#endif

},

};

//设备去遇到匹配,执行驱动probe,完成设备的初始化,注册等工作。

static int gt1x\_ts\_probe(struct i2c\_client \*client, const struct i2c\_device\_id \*id)

{

s32 ret = -1;

#if GTP\_AUTO\_UPDATE

struct task\_struct \*thread = NULL;

#endif

//do NOT remove these logs

pr\_err("stephen in gt1x\_ts\_probe\n");

GTP\_INFO("GTP Driver Version: %s", GTP\_DRIVER\_VERSION);

GTP\_INFO("GTP I2C Address: 0x%02x", client->addr);

gt1x\_i2c\_client = client;

spin\_lock\_init(&irq\_lock);

if (!i2c\_check\_functionality(client->adapter, I2C\_FUNC\_I2C)) {

GTP\_ERROR("I2C check functionality failed.");

return -ENODEV;

}

#ifdef GTP\_CONFIG\_OF /\* device tree support \*/

if (client->dev.of\_node) {

gt1x\_parse\_dt(&client->dev);

}

#endif

ret = gt1x\_request\_io\_port();

if (ret < 0) {

GTP\_ERROR("GTP request IO port failed.");

return ret;

}

gt1x\_init();

INIT\_WORK(&gt1x\_work, gt1x\_ts\_work\_func);

ret = gt1x\_request\_input\_dev();

if (ret < 0) {

GTP\_ERROR("GTP request input dev failed");

}

ret = gt1x\_request\_irq();

if (ret < 0) {

GTP\_INFO("GTP works in polling mode.");

} else {

GTP\_INFO("GTP works in interrupt mode.");

}

#if GTP\_GESTURE\_WAKEUP

enable\_irq\_wake(client->irq);

#endif