in drivers/input/touchscreen/edt-ft5x06.c

```
static const struct i2c_device_id edt_ft5x06_ts_id[] = {
	{ "edt-ft5x06", 0, },
	{ /* sentinel */ }
};
MODULE_DEVICE_TABLE(i2c, edt_ft5x06_ts_id);

#ifdef CONFIG_OF
static const struct of_device_id edt_ft5x06_of_match[] = {
	{ .compatible = "edt,edt-ft5206", },
	{ .compatible = "edt,edt-ft5306", },
	{ .compatible = "edt,edt-ft5406", },
	{ /* sentinel */ }
};
MODULE_DEVICE_TABLE(of, edt_ft5x06_of_match);
#endif

static struct i2c_driver edt_ft5x06_ts_driver = {
	.driver = {
		.owner = THIS_MODULE,
		.name = "edt_ft5x06",
		.of_match_table = of_match_ptr(edt_ft5x06_of_match),
		.pm = &edt_ft5x06_ts_pm_ops,
```

```
    },
    .id_table = edt_ft5x06_ts_id,
    .probe    = edt_ft5x06_ts_probe,
    .remove   = edt_ft5x06_ts_remove,
};

module_i2c_driver(edt_ft5x06_ts_driver);

/**
 * module_i2c_driver() - Helper macro for registering a I2C driver
 * @__i2c_driver: i2c_driver struct
 *
 * Helper macro for I2C drivers which do not do anything special in module
 * init/exit. This eliminates a lot of boilerplate. Each module may only
 * use this macro once, and calling it replaces module_init() and module_exit()
 */
#define module_i2c_driver(__i2c_driver) \
    module_driver(__i2c_driver, i2c_add_driver, \
            i2c_del_driver)
```

==>

module_driver(edt_ft5x06_ts_driver) module_driver(edt_ft5x06_ts_driver, i2c_add_driver, i2c_del_driver)

```
#define module_driver(__driver, __register, __unregister, ...) \
static int __init __driver##_init(void) \
{ \
	return __register(&(__driver) , ##__VA_ARGS__); \
} \
module_init(__driver##_init); \
static void __exit __driver##_exit(void) \
{ \
	__unregister(&(__driver) , ##__VA_ARGS__); \
} \
module_exit(__driver##_exit);
```

==>

```
static int __init edt_ft5x06_ts_driver_init(void)
{
	return	i2c_add_driver(&(edt_ft5x06_ts_driver))
}
module_init(edt_ft5x06_ts_driver_init);
static void __exit edt_ft5x06_ts_driver_exit(void)
{
	i2c_del_driver(&(edt_ft5x06_ts_driver);
}
module_exit(edt_ft5x06_ts_driver_exit);
```

in include/linux/i2c.h

```
#define i2c_add_driver(driver) \
	i2c_register_driver(THIS_MODULE, driver)
```

```
/* use a define to avoid include chaining to get THIS_MODULE */
#define i2c_add_driver(driver) \
	i2c_register_driver(THIS_MODULE, driver)
```

in drivers/i2c/i2c-core.c

```
/*
 * An i2c_driver is used with one or more i2c_client (device) nodes to access
 * i2c slave chips, on a bus instance associated with some i2c_adapter.
 */

int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
{
	int res;

	/* Can't register until after driver model init */
```

```
if (unlikely(WARN_ON(!i2c_bus_type.p)))
        return -EAGAIN;

/* add the driver to the list of i2c drivers in the driver core */
driver->driver.owner = owner;
driver->driver.bus = &i2c_bus_type;

/* When registration returns, the driver core
 * will have called probe() for all matching-but-unbound devices.
 */
res = driver_register(&driver->driver);
if (res)
        return res;

/* Drivers should switch to dev_pm_ops instead. */
if (driver->suspend)
        pr_warn("i2c-core: driver [%s] using legacy suspend method\n",
                driver->driver.name);
if (driver->resume)
        pr_warn("i2c-core: driver [%s] using legacy resume method\n",
                driver->driver.name);

pr_debug("i2c-core: driver [%s] registered\n", driver->driver.name);

INIT_LIST_HEAD(&driver->clients);
```

```
        /* Walk the adapters that are already present */
        i2c_for_each_dev(driver, __process_new_driver);

        return 0;
}

driver_register()
        |
        |
       \|/
bus_add_driver()
        |
        |
       \|/
driver_attach()



/**
 * driver_attach - try to bind driver to devices.
 * @drv: driver.
 *
 * Walk the list of devices that the bus has on it and try to
 * match the driver with each one.  If driver_probe_device()
 * returns 0 and the @dev->driver is set, we've found a
 * compatible pair.
```

```
 */
int driver_attach(struct device_driver *drv)
{
	return bus_for_each_dev(drv->bus, NULL, drv, __driver_attach);
}
```

bus_for_each_dev() enumerate all devices on the bus (I2C bus) and make the drv identify the specific device.

```
static int __driver_attach(struct device *dev, void *data)
{
	struct device_driver *drv = data;

	/*
	 * Lock device and try to bind to it. We drop the error
	 * here and always return 0, because we need to keep trying
	 * to bind to devices and some drivers will return an error
	 * simply if it didn't support the device.
	 *
	 * driver_probe_device() will spit a warning if there
	 * is an error.
	 */

	if (!driver_match_device(drv, dev))
		return 0;
```

```
        if (dev->parent)        /* Needed for USB */
                device_lock(dev->parent);
        device_lock(dev);
        if (!dev->driver)
                driver_probe_device(drv, dev);
        device_unlock(dev);
        if (dev->parent)
                device_unlock(dev->parent);

        return 0;
}

static inline int driver_match_device(struct device_driver *drv,
                                      struct device *dev)
{
        return drv->bus->match ? drv->bus->match(dev, drv) : 1;
}
```

这里的drv->bus->match即是

in drivers/i2c/i2c-core.c

```
struct bus_type i2c_bus_type = {
        .name           = "i2c",
```

```
	.match		= i2c_device_match,
	.probe		= i2c_device_probe,
	.remove		= i2c_device_remove,
	.shutdown	= i2c_device_shutdown,
	.pm		= &i2c_device_pm_ops,
};

static int i2c_device_match(struct device *dev, struct device_driver *drv)
{
	struct i2c_client	*client = i2c_verify_client(dev);
	struct i2c_driver	*driver;

	if (!client)
		return 0;

	/* Attempt an OF style match */
	if (of_driver_match_device(dev, drv))
		return 1;

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

	driver = to_i2c_driver(drv);
	/* match on an id table if there is one */
```

```
	if (driver->id_table)
		return i2c_match_id(driver->id_table, client) != NULL;

	return 0;
}

/**
 * of_driver_match_device - Tell if a driver's of_match_table matches a device.
 * @drv: the device_driver structure to test
 * @dev: the device structure to match against
 */
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;
}

drv->of_match_table ==>

#ifdef CONFIG_OF
static const struct of_device_id edt_ft5x06_of_match[] = {
	{ .compatible = "edt,edt-ft5206", },
	{ .compatible = "edt,edt-ft5306", },
	{ .compatible = "edt,edt-ft5406", },
	{ /* sentinel */ }
```

```
};

MODULE_DEVICE_TABLE(of, edt_ft5x06_of_match);

#endif


static struct i2c_driver edt_ft5x06_ts_driver = {

	.driver = {

		.owner = THIS_MODULE,

		.name = "edt_ft5x06",

		.of_match_table = of_match_ptr(edt_ft5x06_of_match),

		.pm = &edt_ft5x06_ts_pm_ops,

	},

	.id_table = edt_ft5x06_ts_id,

	.probe    = edt_ft5x06_ts_probe,

	.remove   = edt_ft5x06_ts_remove,

};
```

把edt_ft5x06_of_match[]中的".compatible"与bus上的device node中的".compatible" property相比较。

如果match，则driver_match_device(drv, dev) return 1.

```
static int __driver_attach(struct device *dev, void *data)

{

	struct device_driver *drv = data;
```

```
        /*
         * Lock device and try to bind to it. We drop the error
         * here and always return 0, because we need to keep trying
         * to bind to devices and some drivers will return an error
         * simply if it didn't support the device.
         *
         * driver_probe_device() will spit a warning if there
         * is an error.
         */

        if (!driver_match_device(drv, dev))
                return 0;

        if (dev->parent)        /* Needed for USB */
                device_lock(dev->parent);
        device_lock(dev);
        if (!dev->driver)
                driver_probe_device(drv, dev);
        device_unlock(dev);
        if (dev->parent)
                device_unlock(dev->parent);

        return 0;
}
```

```
/**
 * driver_probe_device - attempt to bind device & driver together
 * @drv: driver to bind a device to
 * @dev: device to try to bind to the driver
 *
 * This function returns -ENODEV if the device is not registered,
 * 1 if the device is bound successfully and 0 otherwise.
 *
 * This function must be called with @dev lock held.  When called for a
 * USB interface, @dev->parent lock must be held as well.
 */
int driver_probe_device(struct device_driver *drv, struct device *dev)
{
	int ret = 0;

	if (!device_is_registered(dev))
		return -ENODEV;

	pr_debug("bus: '%s': %s: matched device %s with driver %s\n",
		 drv->bus->name, __func__, dev_name(dev), drv->name);

	pm_runtime_barrier(dev);
	ret = really_probe(dev, drv);
	pm_request_idle(dev);
```

```
	return ret;
}
```

这里drv就是edt-ft5x06 driver，而dev则是touch screen I2C slave device node。

```
static int really_probe(struct device *dev, struct device_driver *drv)
{
	int ret = 0;
	int local_trigger_count = atomic_read(&deferred_trigger_count);

	atomic_inc(&probe_count);
	pr_debug("bus: '%s': %s: probing driver %s with device %s\n",
		 drv->bus->name, __func__, drv->name, dev_name(dev));
	WARN_ON(!list_empty(&dev->devres_head));

	dev->driver = drv;

	/* If using pinctrl, bind pins now before probing */
	ret = pinctrl_bind_pins(dev);
	if (ret)
		goto probe_failed;

	if (driver_sysfs_add(dev)) {
		printk(KERN_ERR "%s: driver_sysfs_add(%s) failed\n",
			__func__, dev_name(dev));
```

```
        goto probe_failed;
    }

    if (dev->bus->probe) {
        ret = dev->bus->probe(dev);                    (1)
        if (ret)
            goto probe_failed;
    } else if (drv->probe) {
        ret = drv->probe(dev);
        if (ret)
            goto probe_failed;
    }

    driver_bound(dev);
    ret = 1;
    pr_debug("bus: '%s': %s: bound device %s to driver %s\n",
         drv->bus->name, __func__, dev_name(dev), drv->name);
    goto done;

probe_failed:
    devres_release_all(dev);
    driver_sysfs_remove(dev);
    dev->driver = NULL;
    dev_set_drvdata(dev, NULL);
```

```
        if (ret == -EPROBE_DEFER) {
                /* Driver requested deferred probing */
                dev_info(dev, "Driver %s requests probe deferral\n", drv->name);
                driver_deferred_probe_add(dev);
                /* Did a trigger occur while probing? Need to re-trigger if yes */
                if (local_trigger_count != atomic_read(&deferred_trigger_count))
                        driver_deferred_probe_trigger();
        } else if (ret != -ENODEV && ret != -ENXIO) {
                /* driver matched but the probe failed */
                printk(KERN_WARNING
                       "%s: probe of %s failed with error %d\n",
                       drv->name, dev_name(dev), ret);
        } else {
                pr_debug("%s: probe of %s rejects match %d\n",
                         drv->name, dev_name(dev), ret);
        }
        /*
         * Ignore errors returned by ->probe so that the next driver can try
         * its luck.
         */
        ret = 0;
done:
        atomic_dec(&probe_count);
        wake_up(&probe_waitqueue);
        return ret;
```

```
}
```

(1)

由于dev->bus->probe != NULL

```
struct bus_type i2c_bus_type = {
        .name           = "i2c",
        .match          = i2c_device_match,
        .probe          = i2c_device_probe,
        .remove         = i2c_device_remove,
        .shutdown       = i2c_device_shutdown,
        .pm         = &i2c_device_pm_ops,
};

static int i2c_device_probe(struct device *dev)
{
        struct i2c_client       *client = i2c_verify_client(dev);
        struct i2c_driver       *driver;
        int status;

        if (!client)
                return 0;

        driver = to_i2c_driver(dev->driver);
        if (!driver->probe || !driver->id_table)
```

```
        return -ENODEV;


    if (!device_can_wakeup(&client->dev))
        device_init_wakeup(&client->dev,
                           client->flags & I2C_CLIENT_WAKE);
    dev_dbg(dev, "probe\n");


    status = of_clk_set_defaults(dev->of_node, false);
    if (status < 0)
        return status;


    status = dev_pm_domain_attach(&client->dev, true);
    if (status != -EPROBE_DEFER) {
        status = driver->probe(client, i2c_match_id(driver->id_table,    (2)
                               client));
        if (status)
            dev_pm_domain_detach(&client->dev, true);
    }


    return status;
}
```

(2)

最终调用的driver自己定义的probe()，也就是edt_ft5x06_ts_probe()，这样driver的初始化就这样开始了！