Device Driver Design Patterns

This document describes a few common design patterns found in device drivers. It is likely that subsystem maintainers will ask driver developers to conform to these design patterns.

1. State Container

While the kernel contains a few device drivers that assume that they will only be probed() once on a certain system (singletons), it is custom to assume that the device the driver binds to will appear in several instances. This means that the probe() function and all callbacks need to be reentrant.

The most common way to achieve this is to use the state container design pattern. It usually has this form:

```
struct foo {
    spinlock_t lock; /* Example member */
    (...)
};

static int foo_probe(...)
{
    struct foo *foo;

    foo = devm_kzalloc(dev, sizeof(*foo), GFP_KERNEL);
    if (!foo)
        return -ENOMEM;
    spin_lock_init(&foo->lock);
    (...)
}
```

This will create an instance of struct foo in memory every time probe() is called. This is our state container for this instance of the device driver. Of course it is then necessary to always pass this instance of the state around to all functions that need access to the state and its members.

For example, if the driver is registering an interrupt handler, you would pass around a pointer to struct foo like this:

```
static irqreturn_t foo_handler(int irq, void *arg)
{
    struct foo *foo = arg;
    (...)
}

static int foo_probe(...)
{
    struct foo *foo;

    (...)
    ret = request_irq(irq, foo_handler, 0, "foo", foo);
}
```

This way you always get a pointer back to the correct instance of foo in your interrupt handler.

2. container_of()

Continuing on the above example we add an offloaded work:

```
struct foo {
    spinlock_t lock;
    struct workqueue_struct *wq;
    struct work_struct offload;
    (...)
};

static void foo_work(struct work_struct *work)
{
    struct foo *foo = container_of(work, struct foo, offload);

    (...)
}

static irqreturn_t foo_handler(int irq, void *arg)
{
    struct foo *foo = arg;
```

```
    queue_work(foo->wq, &foo->offload);
    (...)
}

static int foo_probe(...)
{
    struct foo *foo;

    foo->wq = create_singlethread_workqueue("foo-wq");
    INIT_WORK(&foo->offload, foo_work);
    (...)
}
```

The design pattern is the same for an hrtimer or something similar that will return a single argument which is a pointer to a struct member in the callback.

container_of() is a macro defined in <linux/kernel.h>

What container_of() does is to obtain a pointer to the containing struct from a pointer to a member by a simple subtraction using the offsetof() macro from standard C, which allows something similar to object oriented behaviours. Notice that the contained member must not be a pointer, but an actual member for this to work.

We can see here that we avoid having global pointers to our struct foo * instance this way, while still keeping the number of parameters passed to the work function to a single pointer.