The Common Mailbox Framework

Author: Jassi Brar <jaswinder.singh@linaro.org>

This document aims to help developers write client and controller drivers for the API. But before we start, let us note that the client (especially) and controller drivers are likely going to be very platform specific because the remote firmware is likely to be proprietary and implement non-standard protocol. So even if two platforms employ, say, PL320 controller, the client drivers can't be shared across them. Even the PL320 driver might need to accommodate some platform specific quirks. So the API is meant mainly to avoid similar copies of code written for each platform. Having said that, nothing prevents the remote f/w to also be Linux based and use the same api there. However none of that helps us locally because we only ever deal at client's protocol level.

Some of the choices made during implementation are the result of this peculiarity of this "common" framework.

Controller Driver (See include/linux/mailbox_controller.h)

Allocate mbox_controller and the array of mbox_chan. Populate mbox_chan_ops, except peek_data() all are mandatory. The controller driver might know a message has been consumed by the remote by getting an IRQ or polling some hardware flag or it can never know (the client knows by way of the protocol). The method in order of preference is IRQ -> Poll -> None, which the controller driver should set via 'txdone_irq' or 'txdone_poll' or neither.

Client Driver (See include/linux/mailbox_client.h)

The client might want to operate in blocking mode (synchronously send a message through before returning) or non-blocking/async mode (submit a message and a callback function to the API and return immediately).

```
struct demo_client {
        struct mbox_client cl;
        struct mbox_chan *mbox;
        struct completion c;
        bool async;
        /* ... */
};

/*
* This is the handler for data received from remote. The behaviour is purely
* dependent upon the protocol. This is just an example.
*/
static void message_from_remote(struct mbox_client *cl, void *mssg)
{
        struct demo_client *dc = container_of(cl, struct demo_client, cl);
        if (dc->async) {
                if (is_an_ack(mssg)) {
                        /* An ACK to our last sample sent */
                        return; /* Or do something else here */
                } else { /* A new message from remote */
                        queue_req(mssg);
                }
        } else {
                /* Remote f/w sends only ACK packets on this channel */
                return;
        }
}

static void sample_sent(struct mbox_client *cl, void *mssg, int r)
{
        struct demo_client *dc = container_of(cl, struct demo_client, cl);
        complete(&dc->c);
}

static void client_demo(struct platform_device *pdev)
{
        struct demo_client *dc_sync, *dc_async;
        /* The controller already knows async_pkt and sync_pkt */
        struct async_pkt ap;
        struct sync_pkt sp;

        dc_sync = kzalloc(sizeof(*dc_sync), GFP_KERNEL);
        dc_async = kzalloc(sizeof(*dc_async), GFP_KERNEL);

        /* Populate non-blocking mode client */
        dc_async->cl.dev = &pdev->dev;
        dc_async->cl.rx_callback = message_from_remote;
        dc_async->cl.tx_done = sample_sent;
        dc_async->cl.tx_block = false;
```

```
        dc_async->cl.tx_tout = 0; /* doesn't matter here */
        dc_async->cl.knows_txdone = false; /* depending upon protocol */
        dc_async->async = true;
        init_completion(&dc_async->c);

        /* Populate blocking mode client */
        dc_sync->cl.dev = &pdev->dev;
        dc_sync->cl.rx_callback = message_from_remote;
        dc_sync->cl.tx_done = NULL; /* operate in blocking mode */
        dc_sync->cl.tx_block = true;
        dc_sync->cl.tx_tout = 500; /* by half a second */
        dc_sync->cl.knows_txdone = false; /* depending upon protocol */
        dc_sync->async = false;

        /* ASync mailbox is listed second in 'mboxes' property */
        dc_async->mbox = mbox_request_channel(&dc_async->cl, 1);
        /* Populate data packet */
        /* ap.xxx = 123; etc */
        /* Send async message to remote */
        mbox_send_message(dc_async->mbox, &ap);

        /* Sync mailbox is listed first in 'mboxes' property */
        dc_sync->mbox = mbox_request_channel(&dc_sync->cl, 0);
        /* Populate data packet */
        /* sp.abc = 123; etc */
        /* Send message to remote in blocking mode */
        mbox_send_message(dc_sync->mbox, &sp);
        /* At this point 'sp' has been sent */

        /* Now wait for async chan to be done */
        wait_for_completion(&dc_async->c);
}
```