NuttX Porting guide

30 April, 2020 17:33

6.3.X SocketCAN Device Drivers

- **include/nuttx/net/netdev.h**. All structures and APIs needed to work with drivers are provided in this header file. The structure struct net_driver_s defines the interface and is passed to the network via netdev_register().
- **include/nuttx/can.h.** CAN & CAN FD frame data structures.
- **int netdev_register(FAR struct net_driver_s *dev, enum net_lltype_e lltype);**. Each driver registers itself by calling netdev_register().
- **Include/nuttx/net/can.h** contains lookup tables for CAN dlc to CAN FD len sizes named
  extern const uint8_t can_dlc_to_len[16];
  extern const uint8_t len_to_can_dlc[65];
- **Initialization sequence** is as follows
  1. up_netinitialize(void) is called on startup of NuttX in this function you call your own init function to initialize your CAN driver
  2. In your own init function you create the net_driver_s structure set required init values and register the required callbacks for SocketCAN
  3. Then you ensure that the CAN interface is in down mode (usually done by calling the d_ifdown function)
  4. Register the net_driver_s using netdev_register
- **Receive sequence** is as follows
  1. Device generates interrupt
  2. Process this interrupt in your interrupt handler
  3. When a new CAN frame has been received you process this frame
  4. When the CAN frame is a normal CAN frame you allocate the can_frame struct, when it's a CAN FD frame you allocate a canfd_frame struct (note you can of course preallocate and just use the pointer).
  5. Copy the frame from the driver to the struct you've allocated in the previous step.
  6. Point the net_driver_s d_buf pointer to the allocated can_frame
  7. Call the can_input(FAR struct net_driver_s *dev) function **include/nuttx/net/can.h**
- **Transmit sequence** is as follows
  1. Socket layer executes d_txavail callback
  2. A txavail function looks like this

```
static void driver_txavail(struct net_driver_s *dev)
{
 FAR struct driver_s *priv =
    (FAR struct driver *)dev->d_private;

  /* Ignore the notification if the interface is not yet up */

  net_lock();
  if (priv->bifup)
    {
      /* Check if there is room in the hardware to hold another outgoing
       * packet.
       */

      if (!txfull(priv))
        {
          /* No, there is space for another transfer.  Poll the network
for
           * new XMIT data.
           */

          devif_poll(&priv->dev, s32k1xx_txpoll);
```

```
        }
      }

    net_unlock();
  }
```

3. A txpoll looks like this

```
static int driver_txpoll(struct net_driver_s *dev)
{
  FAR struct driver_s *priv =
    (FAR struct driver_s *)dev->d_private;

  /* If the polling resulted in data that should be sent out on the
network,
   * the field d_len is set to a value > 0.
   */

  if (priv->dev.d_len > 0)
    {
      if (!devif_loopback(&priv->dev))
        {
          /* Send the packet */

          transmit(priv);

          /* Check if there is room in the device to hold another packet.
If
           * not, return a non-zero value to terminate the poll.
           */

          if (txfull(priv))
            {
              return -EBUSY;
            }
        }
    }

  /* If zero is returned, the polling will continue until all connections
   * have been examined.
   */

  return 0;
}
```

4. In your transmit(struct driver_s *priv) function you check the length of net_driver_s d_len whether it matches the size of a can_frame struct or canfd_struct then you cast the content of the net_driver_s d_buf pointer to the correct CAN frame struct

- **Example**: arch/arm/src/s32k1xx/s32k1xx_flexcan.c