DM9000 Network driver

Ben Dooks <ben@simtec.co.uk> <ben-linux@fluff.org>

Introduction

This file describes how to use the DM9000 platform-device based network driver that is contained in the files drivers/net/dm9000.c and drivers/net/dm9000.h.

The driver supports three DM9000 variants, the DM9000E which is the first chip supported as well as the newer DM9000A and DM9000B devices. It is currently maintained and tested by Ben Dooks, who should be CC: to any patches for this driver.

Defining the platform device

The minimum set of resources attached to the platform device are as follows:

1. The physical address of the address register
2. The physical address of the data register
3. The IRQ line the device's interrupt pin is connected to.

These resources should be specified in that order, as the ordering of the two address regions is important (the driver expects these to be address and then data).

An example from arch/arm/mach-s3c/mach-bast.c is:

```
static struct resource bast_dm9k_resource[] = {
      [0] = {
              .start = S3C2410_CS5 + BAST_PA_DM9000,
              .end   = S3C2410_CS5 + BAST_PA_DM9000 + 3,
              .flags = IORESOURCE_MEM,
      },
      [1] = {
              .start = S3C2410_CS5 + BAST_PA_DM9000 + 0x40,
              .end   = S3C2410_CS5 + BAST_PA_DM9000 + 0x40 + 0x3f,
              .flags = IORESOURCE_MEM,
      },
      [2] = {
              .start = IRQ_DM9000,
              .end   = IRQ_DM9000,
              .flags = IORESOURCE_IRQ | IORESOURCE_IRQ_HIGHLEVEL,
      }
};

static struct platform_device bast_device_dm9k = {
      .name           = "dm9000",
      .id             = 0,
      .num_resources  = ARRAY_SIZE(bast_dm9k_resource),
      .resource       = bast_dm9k_resource,
};
```

Note the setting of the IRQ trigger flag in bast_dm9k_resource[2].flags, as this will generate a warning if it is not present. The trigger from the flags field will be passed to request_irq() when registering the IRQ handler to ensure that the IRQ is setup correctly.

This shows a typical platform device, without the optional configuration platform data supplied. The next example uses the same resources, but adds the optional platform data to pass extra configuration data:

```
static struct dm9000_plat_data bast_dm9k_platdata = {
      .flags          = DM9000_PLATF_16BITONLY,
};

static struct platform_device bast_device_dm9k = {
      .name           = "dm9000",
      .id             = 0,
      .num_resources  = ARRAY_SIZE(bast_dm9k_resource),
      .resource       = bast_dm9k_resource,
      .dev            = {
              .platform_data = &bast_dm9k_platdata,
      }
};
```

The platform data is defined in include/linux/dm9000.h and described below.

Platform data

Platform data

Extra platform data for the DM9000 can describe the IO bus width to the device, whether or not an external PHY is attached to the device and the availability of an external configuration EEPROM.

The flags for the platform data .flags field are as follows:

DM9000_PLATF_8BITONLY

The IO should be done with 8bit operations.

DM9000_PLATF_16BITONLY

The IO should be done with 16bit operations.

DM9000_PLATF_32BITONLY

The IO should be done with 32bit operations.

DM9000_PLATF_EXT_PHY

The chip is connected to an external PHY.

DM9000_PLATF_NO_EEPROM

This can be used to signify that the board does not have an EEPROM, or that the EEPROM should be hidden from the user.

DM9000_PLATF_SIMPLE_PHY

Switch to using the simpler PHY polling method which does not try and read the MII PHY state regularly. This is only available when using the internal PHY. See the section on link state polling for more information.

The config symbol DM9000_FORCE_SIMPLE_PHY_POLL, Kconfig entry "Force simple NSR based PHY polling" allows this flag to be forced on at build time.

PHY Link state polling

The driver keeps track of the link state and informs the network core about link (carrier) availability. This is managed by several methods depending on the version of the chip and on which PHY is being used.

For the internal PHY, the original (and currently default) method is to read the MII state, either when the status changes if we have the necessary interrupt support in the chip or every two seconds via a periodic timer.

To reduce the overhead for the internal PHY, there is now the option of using the DM9000_FORCE_SIMPLE_PHY_POLL config, or DM9000_PLATF_SIMPLE_PHY platform data option to read the summary information without the expensive MII accesses. This method is faster, but does not print as much information.

When using an external PHY, the driver currently has to poll the MII link status as there is no method for getting an interrupt on link change.

DM9000A / DM9000B

These chips are functionally similar to the DM9000E and are supported easily by the same driver. The features are:

1. Interrupt on internal PHY state change. This means that the periodic polling of the PHY status may be disabled on these devices when using the internal PHY.
2. TCP/UDP checksum offloading, which the driver does not currently support.

ethtool

The driver supports the ethtool interface for access to the driver state information, the PHY state and the EEPROM.