Linux Driver for the Synopsys(R) Ethernet Controllers "stmmac"

Authors: Giuseppe Cavallaro <peppe.cavallaro@st.com>, Alexandre Torgue <alexandre.torgue@st.com>, Jose Abreu <joabreu@synopsys.com>

Contents

In This Release

This file describes the stmmac Linux Driver for all the Synopsys(R) Ethernet Controllers.

Currently, this network device driver is for all STi embedded MAC/GMAC (i.e. 7xxx/5xxx SoCs), SPEAr (arm), Loongson1B (mips) and XILINX XC2V3000 FF1152AMT0221 D1215994A VIRTEX FPGA board. The Synopsys Ethernet QoS 5.0 IPK is also supported.

DesignWare(R) Cores Ethernet MAC 10/100/1000 Universal version 3.70a (and older) and DesignWare(R) Cores Ethernet Quality-of-Service version 4.0 (and upper) have been used for developing this driver as well as DesignWare(R) Cores XGMAC - 10G Ethernet MAC and DesignWare(R) Cores Enterprise MAC - 100G Ethernet MAC.

This driver supports both the platform bus and PCI.

This driver includes support for the following Synopsys(R) DesignWare(R) Cores Ethernet Controllers and corresponding minimum and maximum versions:

Controller Name	Min. Version	Max. Version	Abbrev. Name
Ethernet MAC Universal	N/A	3.73a	GMAC
Ethernet Quality-of-Service	4.00a	N/A	GMAC4+
XGMAC - 10G Ethernet MAC	2.10a	N/A	XGMAC2+
XLGMAC - 100G Ethernet MAC	2.00a	N/A	XLGMAC2+

For questions related to hardware requirements, refer to the documentation supplied with your Ethernet adapter. All hardware requirements listed apply to use with Linux.

Feature List

The following features are available in this driver:

- GMII/MII/RGMII/SGMII/RMII/XGMII/XLGMII Interface
- Half-Duplex / Full-Duplex Operation
- Energy Efficient Ethernet (EEE)
- IEEE 802.3x PAUSE Packets (Flow Control)
- RMON/MIB Counters
- IEEE 1588 Timestamping (PTP)
- Pulse-Per-Second Output (PPS)
- MDIO Clause 22 / Clause 45 Interface
- MAC Loopback
- ARP Offloading
- Automatic CRC / PAD Insertion and Checking
- Checksum Offload for Received and Transmitted Packets
- Standard or Jumbo Ethernet Packets
- Source Address Insertion / Replacement
- VLAN TAG Insertion / Replacement / Deletion / Filtering (HASH and PERFECT)
- Programmable TX and RX Watchdog and Coalesce Settings
- Destination Address Filtering (PERFECT)
- HASH Filtering (Multicast)
- Layer 3 / Layer 4 Filtering
- Remote Wake-Up Detection

- Receive Side Scaling (RSS)
- Frame Preemption for TX and RX
- Programmable Burst Length, Threshold, Queue Size
- Multiple Queues (up to 8)
- Multiple Scheduling Algorithms (TX: WRR, DWRR, WFQ, SP, CBS, EST, TBS; RX: WRR, SP)
- Flexible RX Parser
- TCP / UDP Segmentation Offload (TSO, USO)
- Split Header (SPH)
- Safety Features (ECC Protection, Data Parity Protection)
- Selftests using Ethtool

Kernel Configuration

The kernel configuration option is `CONFIG_STMMAC_ETH`:

- `CONFIG_STMMAC_PLATFORM`: is to enable the platform driver.
- `CONFIG_STMMAC_PCI`: is to enable the pci driver.

Command Line Parameters

If the driver is built as a module the following optional parameters are used by entering them on the command line with the modprobe command using this syntax (e.g. for PCI module):

```
modprobe stmmac_pci [<option>=<VAL1>,<VAL2>,...]
```

Driver parameters can be also passed in command line by using:

```
stmmaceth=watchdog:100,chain_mode=1
```

The default value for each parameter is generally the recommended setting, unless otherwise noted.

watchdog

Valid Range:	5000-None
Default Value:	5000

This parameter overrides the transmit timeout in milliseconds.

debug

Valid Range:	0-16 (0=none,...,16=all)
Default Value:	0

This parameter adjusts the level of debug messages displayed in the system logs.

phyaddr

Valid Range:	0-31
Default Value:	-1

This parameter overrides the physical address of the PHY device.

flow_ctrl

Valid Range:	0-3 (0=off,1=rx,2=tx,3=rx/tx)
Default Value:	3

This parameter changes the default Flow Control ability.

pause

Valid Range:	0-65535
Default Value:	65535

This parameter changes the default Flow Control Pause time.

tc

Valid Range:	64-256
Default Value:	64

This parameter changes the default HW FIFO Threshold control value.

buf_sz

Valid Range:	1536-16384
Default Value:	1536

This parameter changes the default RX DMA packet buffer size.

eee_timer

Valid Range:	0-None
Default Value:	1000

This parameter changes the default LPI TX Expiration time in milliseconds.

chain_mode

Valid Range:	0-1 (0=off,1=on)
Default Value:	0

This parameter changes the default mode of operation from Ring Mode to Chain Mode.

Driver Information and Notes

Transmit Process

The xmit method is invoked when the kernel needs to transmit a packet; it sets the descriptors in the ring and informs the DMA engine that there is a packet ready to be transmitted.

By default, the driver sets the `NETIF_F_SG` bit in the features field of the `net_device` structure, enabling the scatter-gather feature. This is true on chips and configurations where the checksum can be done in hardware.

Once the controller has finished transmitting the packet, timer will be scheduled to release the transmit resources.

Receive Process

When one or more packets are received, an interrupt happens. The interrupts are not queued, so the driver has to scan all the descriptors in the ring during the receive process.

This is based on NAPI, so the interrupt handler signals only if there is work to be done, and it exits. Then the poll method will be scheduled at some future point.

The incoming packets are stored, by the DMA, in a list of pre-allocated socket buffers in order to avoid the memcpy (zero-copy).

Interrupt Mitigation

The driver is able to mitigate the number of its DMA interrupts using NAPI for the reception on chips older than the 3.50. New chips have an HW RX Watchdog used for this mitigation.

Mitigation parameters can be tuned by ethtool.

WoL

Wake up on Lan feature through Magic and Unicast frames are supported for the GMAC, GMAC4/5 and XGMAC core.

DMA Descriptors

Driver handles both normal and alternate descriptors. The latter has been only tested on DesignWare(R) Cores Ethernet MAC Universal version 3.41a and later.

stmmac supports DMA descriptor to operate both in dual buffer (RING) and linked-list(CHAINED) mode. In RING each descriptor points to two data buffer pointers whereas in CHAINED mode they point to only one data buffer pointer. RING mode is the default.

In CHAINED mode each descriptor will have pointer to next descriptor in the list, hence creating the explicit chaining in the descriptor itself, whereas such explicit chaining is not possible in RING mode.

Extended Descriptors

The extended descriptors give us information about the Ethernet payload when it is carrying PTP packets or TCP/UDP/ICMP over IP. These are not available on GMAC Synopsys(R) chips older than the 3.50. At probe time the driver will decide if these can be actually used. This support also is mandatory for PTPv2 because the extra descriptors are used for saving the hardware timestamps and Extended Status.

Ethtool Support

Ethtool is supported. For example, driver statistics (including RMON), internal errors can be taken using:

```
ethtool -S ethX
```

Ethtool selftests are also supported. This allows to do some early sanity checks to the HW using MAC and PHY loopback mechanisms:

```
ethtool -t ethX
```

Jumbo and Segmentation Offloading

Jumbo frames are supported and tested for the GMAC. The GSO has been also added but it's performed in software. LRO is not supported.

TSO Support

TSO (TCP Segmentation Offload) feature is supported by GMAC > 4.x and XGMAC chip family. When a packet is sent through TCP protocol, the TCP stack ensures that the SKB provided to the low level driver (stmmac in our case) matches with the maximum frame len (IP header + TCP header + payload <= 1500 bytes (for MTU set to 1500)). It means that if an application using TCP want to send a packet which will have a length (after adding headers) > 1514 the packet will be split in several TCP packets: The data payload is split and headers (TCP/IP ..) are added. It is done by software.

When TSO is enabled, the TCP stack doesn't care about the maximum frame length and provide SKB packet to stmmac as it is. The GMAC IP will have to perform the segmentation by it self to match with maximum frame length.

This feature can be enabled in device tree through `snps,tso` entry.

Energy Efficient Ethernet

Energy Efficient Ethernet (EEE) enables IEEE 802.3 MAC sublayer along with a family of Physical layer to operate in the Low Power Idle (LPI) mode. The EEE mode supports the IEEE 802.3 MAC operation at 100Mbps, 1000Mbps and 1Gbps.

The LPI mode allows power saving by switching off parts of the communication device functionality when there is no data to be transmitted & received. The system on both the side of the link can disable some functionalities and save power during the period of low-link utilization. The MAC controls whether the system should enter or exit the LPI mode and communicate this to PHY.

As soon as the interface is opened, the driver verifies if the EEE can be supported. This is done by looking at both the DMA HW capability register and the PHY devices MCD registers.

To enter in TX LPI mode the driver needs to have a software timer that enable and disable the LPI mode when there is nothing to be transmitted.

Precision Time Protocol (PTP)

The driver supports the IEEE 1588-2002, Precision Time Protocol (PTP), which enables precise synchronization of clocks in measurement and control systems implemented with technologies such as network communication.

In addition to the basic timestamp features mentioned in IEEE 1588-2002 Timestamps, new GMAC cores support the advanced timestamp features. IEEE 1588-2008 can be enabled when configuring the Kernel.

SGMII/RGMII Support

New GMAC devices provide own way to manage RGMII/SGMII. This information is available at run-time by looking at the HW capability register. This means that the stmmac can manage auto-negotiation and link status w/o using the PHYLIB stuff. In fact, the HW provides a subset of extended registers to restart the ANE, verify Full/Half duplex mode and Speed. Thanks to these registers, it is possible to look at the Auto-negotiated Link Parter Ability.

Physical

The driver is compatible with Physical Abstraction Layer to be connected with PHY and GPHY devices.

Platform Information

Several information can be passed through the platform and device-tree.

```
struct plat_stmmacenet_data {
```

1. Bus identifier:

   ```
   int bus_id;
   ```

2) PHY Physical Address. If set to -1 the driver will pick the first PHY it finds:

```
int phy_addr;
```

3. PHY Device Interface:

   ```
   int interface;
   ```

4. Specific platform fields for the MDIO bus:

   ```
   struct stmmac_mdio_bus_data *mdio_bus_data;
   ```

5. Internal DMA parameters:

   ```
   struct stmmac_dma_cfg *dma_cfg;
   ```

6. Fixed CSR Clock Range selection:

   ```
   int clk_csr;
   ```

7. HW uses the GMAC core:

   ```
   int has_gmac;
   ```

8. If set the MAC will use Enhanced Descriptors:

   ```
   int enh_desc;
   ```

9. Core is able to perform TX Checksum and/or RX Checksum in HW:

   ```
   int tx_coe;
   int rx_coe;
   ```

11) Some HWs are not able to perform the csum in HW for over-sized frames due to limited buffer sizes. Setting this flag the csum will be done in SW on JUMBO frames:

```
int bugged_jumbo;
```

12. Core has the embedded power module:

   ```
   int pmt;
   ```

13. Force DMA to use the Store and Forward mode or Threshold mode:

   ```
   int force_sf_dma_mode;
   int force_thresh_dma_mode;
   ```

15. Force to disable the RX Watchdog feature and switch to NAPI mode:

   ```
   int riwt_off;
   ```

16. Limit the maximum operating speed and MTU:

   ```
   int max_speed;
   int maxmtu;
   ```

18. Number of Multicast/Unicast filters:

   ```
   int multicast_filter_bins;
   int unicast_filter_entries;
   ```

20. Limit the maximum TX and RX FIFO size:

   ```
   int tx_fifo_size;
   int rx_fifo_size;
   ```

21. Use the specified number of TX and RX Queues:

   ```
   u32 rx_queues_to_use;
   u32 tx_queues_to_use;
   ```

22. Use the specified TX and RX scheduling algorithm:

   ```
   u8 rx_sched_algorithm;
   u8 tx_sched_algorithm;
   ```

23. Internal TX and RX Queue parameters:

   ```
   struct stmmac_rxq_cfg rx_queues_cfg[MTL_MAX_RX_QUEUES];
   struct stmmac_txq_cfg tx_queues_cfg[MTL_MAX_TX_QUEUES];
   ```

24) This callback is used for modifying some syscfg registers (on ST SoCs) according to the link speed negotiated by the physical layer:

```
void (*fix_mac_speed)(void *priv, unsigned int speed);
```

25) Callbacks used for calling a custom initialization; This is sometimes necessary on some platforms (e.g. ST boxes) where the HW needs to have set some PIO lines or system cfg registers. init/exit callbacks should not use or modify platform data:

```
int (*init)(struct platform_device *pdev, void *priv);
void (*exit)(struct platform_device *pdev, void *priv);
```

26) Perform HW setup of the bus. For example, on some ST platforms this field is used to configure the AMBA bridge to generate more efficient STBus traffic:

```
struct mac_device_info *(*setup)(void *priv);
void *bsp_priv;
```

27. Internal clocks and rates:

    ```
    struct clk *stmmac_clk;
    struct clk *pclk;
    struct clk *clk_ptp_ref;
    unsigned int clk_ptp_rate;
    unsigned int clk_ref_rate;
    s32 ptp_max_adj;
    ```

28. Main reset:

    ```
    struct reset_control *stmmac_rst;
    ```

29. AXI Internal Parameters:

    ```
    struct stmmac_axi *axi;
    ```

30. HW uses GMAC>4 cores:

    ```
    int has_gmac4;
    ```

31. HW is sun8i based:

    ```
    bool has_sun8i;
    ```

32. Enables TSO feature:

    ```
    bool tso_en;
    ```

33. Enables Receive Side Scaling (RSS) feature:

    ```
    int rss_en;
    ```

34. MAC Port selection:

    ```
    int mac_port_sel_speed;
    ```

35. Enables TX LPI Clock Gating:

    ```
    bool en_tx_lpi_clockgating;
    ```

36. HW uses XGMAC>2.10 cores:

    ```
    int has_xgmac;
    ```

```
}
```

For MDIO bus data, we have:

```
struct stmmac_mdio_bus_data {
```

1. PHY mask passed when MDIO bus is registered:

    ```
    unsigned int phy_mask;
    ```

2. List of IRQs, one per PHY:

    ```
    int *irqs;
    ```

3. If IRQs is NULL, use this for probed PHY:

    ```
    int probed_phy_irq;
    ```

4. Set to true if PHY needs reset:

    ```
    bool needs_reset;
    ```

```
}
```

For DMA engine configuration, we have:

```
struct stmmac_dma_cfg {
```

1. Programmable Burst Length (TX and RX):

    ```
    int pbl;
    ```

2. If set, DMA TX / RX will use this value rather than pbl:

   ```
   int txpbl;
   int rxpbl;
   ```

3. Enable 8xPBL:

   ```
   bool pblx8;
   ```

4. Enable Fixed or Mixed burst:

   ```
   int fixed_burst;
   int mixed_burst;
   ```

5. Enable Address Aligned Beats:

   ```
   bool aal;
   ```

6. Enable Enhanced Addressing (> 32 bits):

   ```
   bool eame;
   ```

```
}
```

For DMA AXI parameters, we have:

```
struct stmmac_axi {
```

1. Enable AXI LPI:

   ```
   bool axi_lpi_en;
   bool axi_xit_frm;
   ```

2. Set AXI Write / Read maximum outstanding requests:

   ```
   u32 axi_wr_osr_lmt;
   u32 axi_rd_osr_lmt;
   ```

3. Set AXI 4KB bursts:

   ```
   bool axi_kbbe;
   ```

4. Set AXI maximum burst length map:

   ```
   u32 axi_blen[AXI_BLEN];
   ```

5. Set AXI Fixed burst / mixed burst:

   ```
   bool axi_fb;
   bool axi_mb;
   ```

6. Set AXI rebuild incrx mode:

   ```
   bool axi_rb;
   ```

```
}
```

For the RX Queues configuration, we have:

```
struct stmmac_rxq_cfg {
```

1. Mode to use (DCB or AVB):

   ```
   u8 mode_to_use;
   ```

2. DMA channel to use:

   ```
   u32 chan;
   ```

3. Packet routing, if applicable:

   ```
   u8 pkt_route;
   ```

4. Use priority routing, and priority to route:

   ```
   bool use_prio;
   u32 prio;
   ```

```
}
```

For the TX Queues configuration, we have:

```
struct stmmac_txq_cfg {
```

1. Queue weight in scheduler:

   ```
   u32 weight;
   ```

2. Mode to use (DCB or AVB):

   ```
   u8 mode_to_use;
   ```

3. Credit Base Shaper Parameters:

   ```
   u32 send_slope;
   u32 idle_slope;
   u32 high_credit;
   u32 low_credit;
   ```

4. Use priority scheduling, and priority:

   ```
   bool use_prio;
   u32 prio;
   ```

```
}
```

Device Tree Information

Please refer to the following document: Documentation/devicetree/bindings/net/snps,dwmac.yaml

HW Capabilities

Note that, starting from new chips, where it is available the HW capability register, many configurations are discovered at run-time for example to understand if EEE, HW csum, PTP, enhanced descriptor etc are actually available. As strategy adopted in this driver, the information from the HW capability register can replace what has been passed from the platform.

Debug Information

The driver exports many information i.e. internal statistics, debug information, MAC and DMA registers etc.

These can be read in several ways depending on the type of the information actually needed.

For example a user can be use the ethtool support to get statistics: e.g. using: `ethtool -S ethX` (that shows the Management counters (MMC) if supported) or sees the MAC/DMA registers: e.g. using: `ethtool -d ethX`

Compiling the Kernel with `CONFIG_DEBUG_FS` the driver will export the following debugfs entries:

- `descriptors_status`: To show the DMA TX/RX descriptor rings
- `dma_cap`: To show the HW Capabilities

Developer can also use the `debug` module parameter to get further debug information (please see: NETIF Msg Level).

Support

If an issue is identified with the released source code on a supported kernel with a supported adapter, email the specific information related to the issue to netdev@vger.kernel.org