*User's Guide*

# *TI CPSW3G Debug Guide*

*Shaunak Deshpande, Aswin Puthenpurackal Dileep*

## Abstract

This application note is a comprehensive debugging guide for the CPSW (common platform switch) peripheral integrated within Texas Instruments' AM2x Sitara™ microcontroller family. Many TI controllers and processors use different variants of the CPSW, such as the CPSW2G, CPSW3G, CPSW5G, CPSW9G, and so forth. The CPSW3G variant is a hardware subsystem on AM26x MCUs from TI that enables high-performance Ethernet connectivity through two external gigabit Ethernet ports and one host port, supporting advanced switching and bridging capabilities essential for industrial communication, real-time networking, time-sensitive networking, and embedded Ethernet applications.

This application note is targeted at embedded software engineers and system developers working with designs based on the AM2x MCU and using the MCU+ SDK framework. This document serves as a practical troubleshooting resource for diagnosing and resolving common issues encountered during Ethernet application development. This application note enables developers to quickly identify root causes of connectivity failures, configuration errors, and packet loss.

## Table of Contents



## Trademarks

# 1 Acronyms

**Table 1-1. Acronyms Used in this Guide**

| Acronym | Definition |
|---|---|
| ALE | Address Look-up Engine |
| API | Application Programming Interface |
| ARP | Address Resolution Protocol |
| CBA | Common Bus Architecture |
| CPDMA | Core Packet Direct Memory Access |
| CPPI | Communications Port Programming Interface |
| CPPI | Communications Port Programming Interface |
| CPSW | Common Platform Switch |
| CPTS | Common Platform Timesync |
| CPU | Central Processing Unit |
| CRC | Cyclic Redundancy Check |
| DA | Digital-to-Analog |
| DHCP | Dynamic Host Configuration Protocol |
| DMA | Direct Memory Access |
| DTLS | Datagram Transport Layer Security |
| DUT | Device Under Test |
| eAVB | Ethernet Audio Video Bridging |
| ECC | Error-Correcting Code |
| EEPROM | Electrically Erasable Programmable Read-Only Memory |
| EMI | Electromagnetic Interference |
| ENET | Ethernet |
| ENET-LLD | Unified Ethernet Low-level Driver |
| EST | Electronic Spark Timing |
| ETHPHY | Ethernet Physical Layer |
| EVM | Evaluation Module |
| FIFO | First In, First Out |
| FreeRTOS | Free Real-Time Operating System |
| GEL | General Extension Language |
| GMAC | Gigabit Media Access Control |
| GMII | Gigabit Media Independent Interface |
| gPTP | Generalized Precision Time Protocol |
| GUI | Graphical User Interface |
| HTTP | Hypertext Transfer Protocol |
| ID | Identifier |
| IEEE | Institute of Electrical and Electronics Engineers |
| IET | Institution of Engineering and Technology |

**Table 1-1. Acronyms Used in this Guide (continued)**

| Acronym | Definition |
|---|---|
| IO | Input/Output |
| IOCTL | Input/Output Control |
| IP | Internet Protocol |
| iPerf | Internet Performance |
| KB | Kilobyte |
| LED | Light Emitting Diode |
| LLD | Low-level Driver |
| LLDP | Link Layer Discovery Protocol |
| LwIP | Lightweight Internet Protocol |
| MAC | Media Access Control |
| MB | MegaByte |
| MbedTLS | Arm® Mbed™ Platform Transport Layer Security |
| Mbps | Megabits Per Second |
| MCU | Microcontroller Unit |
| MDC | Management Data Clock |
| MDIO | Management Data Input/Output |
| MII | Media-Independent Interface |
| MTU | Maximum Transmission Unit |
| MUX | Multiplexer |
| NETIF | Network Interface |
| NoRTOS | No Real-Time Operating System |
| OS | Operating System |
| PC | Personal Computer |
| PHY | Physical Layer |
| QoS | Quality of Service |
| QSGMII | Quad Serial Gigabit Media Independent Interface |
| RGMII | Reduced Gigabit Media Independent Interface |
| RMII | Reduced Media-Independent Interface |
| RTOS | Real-Time Operating System |
| RX | Receive |
| SA | Source Address |
| SDK | Software Development Kit |
| SGMII | Serial Gigabit Media Independent Interface |
| SoC | System on a Chip |
| SRAM | Static Random-access Memory |
| STATS | Statistical Static Timing Analysis |
| subnet | sub-network |
| SYSCFG | System Configuration |
| TCM | Tightly Coupled Memory |
| TCP | Transmission Control Protocol |
| TCP/IP | Transmission Control Protocol and Internet Protocol |
| TLS | Transport Layer Security |

**Table 1-1. Acronyms Used in this Guide (continued)**

| Acronym | Definition |
|---|---|
| TSN | Time Since New |
| TSN | Time-Sensitive Networking |
| TX | Transmit |
| TX/RX | Transmit and Receive |
| UDP | User Datagram Protocol |
| VID | Voltage Identification Digital |
| VLAN | Virtual Local Area Network |

## 2 Introduction

The CPSW is an integrated Ethernet switching peripheral that provides hardware-accelerated packet switching, routing, and filtering capabilities across multiple Ethernet ports. On the AM26x devices, the CPSW3G variant integrates a three-port switch fabric with dual external gigabit ports and a host port interface to the CPU subsystem. The peripheral handles Layer 2 switching operations through hardware sub-blocks, including the address look-up engine (ALE) for MAC address management, port controllers for physical interface management, and DMA engines for efficient packet transfer.

The ENET-LLD (unified Ethernet low-level driver) within the MCU+ SDK (microcontroller unit software development kit) abstracts the CPSW3G hardware complexity through a modular software architecture. The driver manages peripheral initialization, PHY (physical layer) configuration, packet buffer allocation, interrupt handling, and provides APIs (application programming interfaces) for frame transmission and reception. The driver architecture separates hardware-specific implementations from application-level interfaces, enabling portable code across TI's devices (AM2x and AM6x families) enabled by the CPSW.

This document focuses on systematic debugging approaches for CPSW3G, demonstrated on the AM261x devices. The debugging practices discussed in this document can also be applied to other CPSW variants and TI devices.

# 3 Introduction to CPSWSS and ENET-LLD

The AM261x MCU targets industrial automation and real-time networking applications. The following lists the key specifications relevant to Ethernet debugging:

Arm® Cortex®-R5F CPU:

- 500MHz with 256KB TCM per core, 1.5MB SRAM
- Sufficient compute for networking stacks and real-time processing

CPSW3G Ethernet Switch:

- Three-port gigabit switch (one host port and two external MAC ports)
- Hardware Layer 2 switching and routing
- TSN support
- Cut-through switching at the line rate
- Address look-up engine (ALE) for MAC filtering and VLAN management
- Hardware inter-VLAN routing
- MII, RMII, RGMII with support for 10Mbps, 100Mbps, and 1000Mbps operations

## 3.1 Hardware

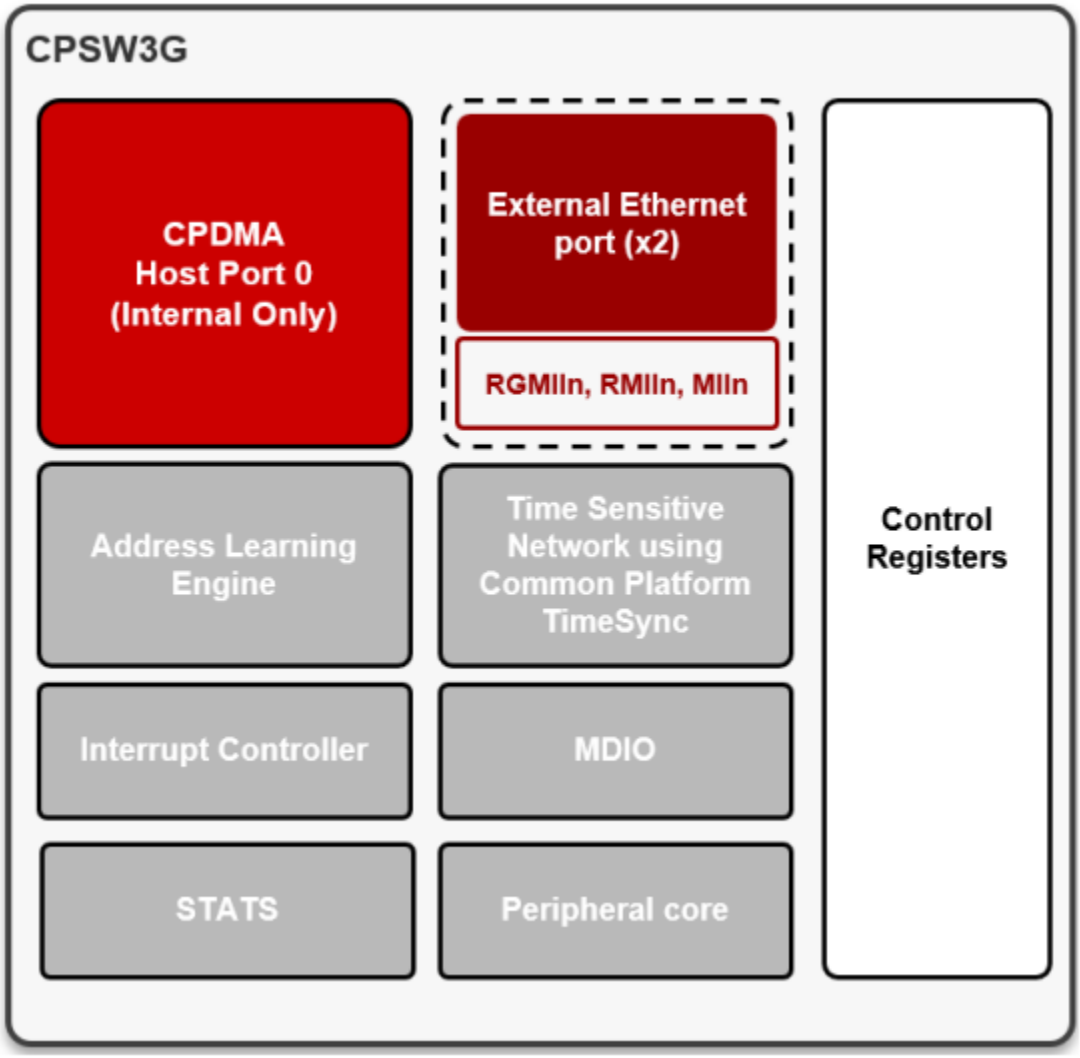


**Figure 3-1. CPSWSS Block Diagram**

The CPSW subsystem provides Ethernet communication with hardware switching capabilities compliant to the IEEE 802.3 (*Ethernet Working Group*) standard. CPSW3G on AM26x devices supports 10Mbps, 100Mbps, or 1000Mbps operations with a selectable MII, RMII, or RGMII.

**Port Architecture**

- Port-0 (host port): Port-0 is a CPDMA and CPPI between the CPU and MAC ports. All packets to and from the Arm Cortex-R5F core traverse this port.
- Port-1 and Port-2 (external MAC ports): Port-1 and Port-2 are physical Ethernet interfaces that support MII, GMII, RMII, RGMII, SGMII, and QSGMII.

SYSCFG automatically generates code and ENET-LLD handles the initial configuration. The application layer performs additional runtime configuration as necessary. Any packet entering the device externally enters through either the MAC port-1 or MAC port-2. If the packet is meant to be consumed by the device, the packet is forwarded to the host port. If the packet is meant to be sent to another port, the packet is switched to the other MAC port.

**CPSW Sub-blocks**

The CPSW subsystem is comprised of these smaller blocks:

- CPDMA: The packet DMA controller handles transfers between the host port and the Arm Cortex-R5F core. The CPDMA implements Ethernet ports that support a line-rate bandwidth interface compliant to TI's CPPI 3.0 and CBA 3.1 transfer standards.
- ALE (address look-up engine): The ALE processes incoming packets using a port number, destination and source MAC, EtherType, and VLAN information. The ALE has an output port-mask that indicates the forwarding destinations. The ALE is critical for packet routing and filtering.
- Interrupt controller: The interrupt controller handles six interrupt sources, including TX/RX packet events, per-port statistical updates, link status changes, ECC errors, and ALE events.
- Statistic module: This module includes hardware counters that track per-port statistics. The module captures good and bad frame counts, drop counts (ALE overrun, FIFO overrun, fragmentation, portmask drops, oversized or undersized frames), and CRC and alignment errors.
- CPTS (common platform timesync): The CPTS is the timestamping engine for TX/RX packets. Timing events are pushed to the CPSW FIFO circuit for gPTP and TSN operations.
- MDIO (management data input/output): The serial management interface that controls the external PHY devices according to IEEE 802.3 (*Standard for Ethernet*). The MDIO handles the PHY configuration (link speed, duplex, power), status monitoring (link up or down), and automatic negotiation.

**Packet Classification**

CPSW hardware can classify packets based on the following:

- Destination MAC address
- Source MAC address
- VLAN ID and priority
- EtherType
- Destination and source IP header fields
- A combination of the parameters in this list

Based on the above list, a variety of rules can be set for consuming, forwarding, or dropping a packet on MAC ports. The CPSW also supports hardware-based modification of some of the above fields of the frame—offloading this operation from the CPU or software.

**VLAN Support**

Hardware VLAN tagging and filtering enables logical network segmentation. CPSW supports the following:

- Insertion and removal of the VLAN tags
- Packet classification based on the VLAN
- Inter-VLAN and Intra-VLAN routing in hardware
- Broadcast domain isolation

Use-cases include isolating the control plane from the traffic of the data plane, prioritizing safety-critical messages, and multi-tenant communication over shared physical infrastructure.

### Rate Limiting

Policers enforce bandwidth limits at the ingress and egress of each MAC port. Packets that exceed the configured thresholds are dropped or remarked. Policers are essential for preventing congestion and maintaining fairness and deterministic latency in time-sensitive applications. Policers combined with QoS and VLAN tagging, provides complete traffic management for industrial Ethernet systems.

## 3.2 Software

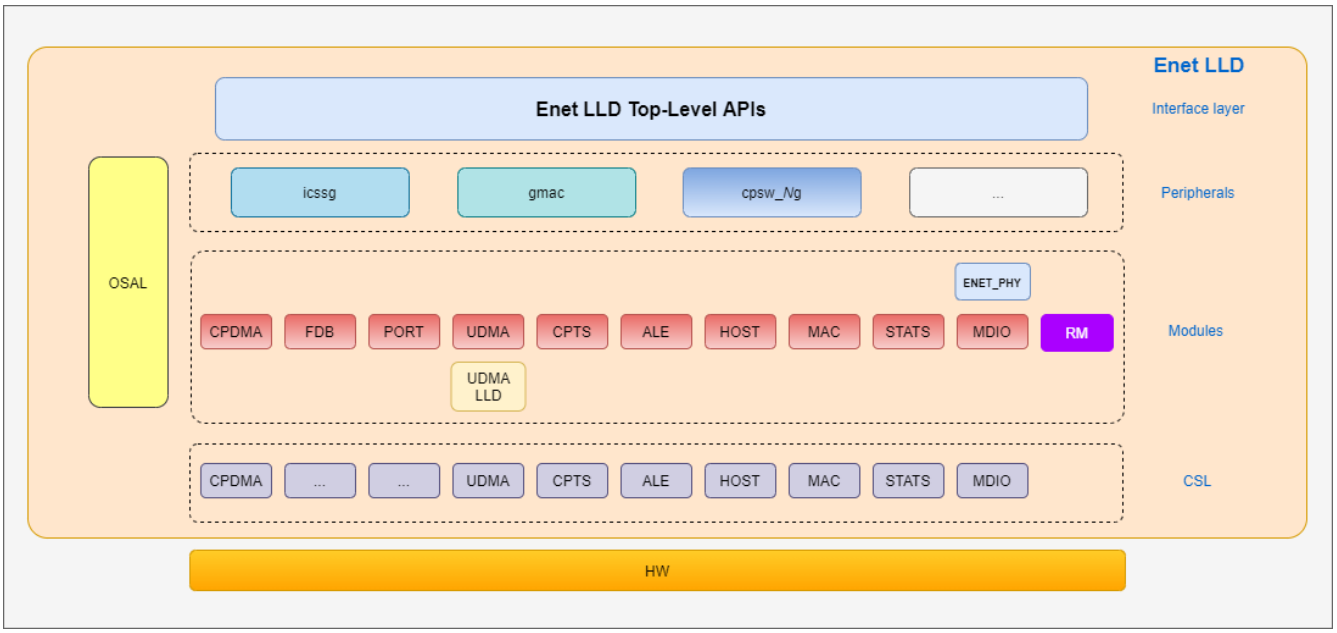


**Figure 3-2. ENET-LLD Block Diagram**

The ENET-LLD abstracts the CPSW hardware through a modular driver architecture. The driver provides hardware initialization, runtime configuration, and data path APIs for packet transmission and reception. The core responsibilities of the ENET-LLD include:

- **Peripheral initialization**: Initializing the CPSW module clocking, reset sequencing, port configuration, and register programming
- **PHY management**: Managing PHY detection through MDIO, link state monitoring, automatic negotiation handling, and speed and duplex configuration
- **DMA setup**: Setting up the CPDMA descriptor ring allocation, TX/RX channel configuration, interrupt routing, and buffer management
- **Packet APIs**: Includes zero-copy TX/RX interfaces, scatter-gather support, and timestamp extraction for CPTS events

The driver separates the hardware abstraction (CPSW register access, DMA operations) from the application interfaces, enabling portability across TI devices with CPSW variants (AM2x and AM6x families). ENET-LLD integrates with initialization code generated through SYSCFG. The SYSCFG GUI exposes configuration parameters (port modes, PHY addresses, MDIO settings, and pin MUX) that generate compile-time structures consumed by the ENET-LLD during the Enet_open() function. Runtime configurations (ALE entries, packet classifiers, and rate limiters) are handled by the ENET-LLD IOCTL interface after initialization.

The driver operates in polling or interrupt mode. The interrupt-driven operation uses the interrupt controller events of the CPSW (RX packet ready, TX complete, and link change) to trigger callbacks registered by the application. For debugging, the ENET-LLD provides IOCTL commands to dump the ALE tables, read hardware statistics, query PHY registers, and access the internal state of the CPSW.

## 3.3 Application Software

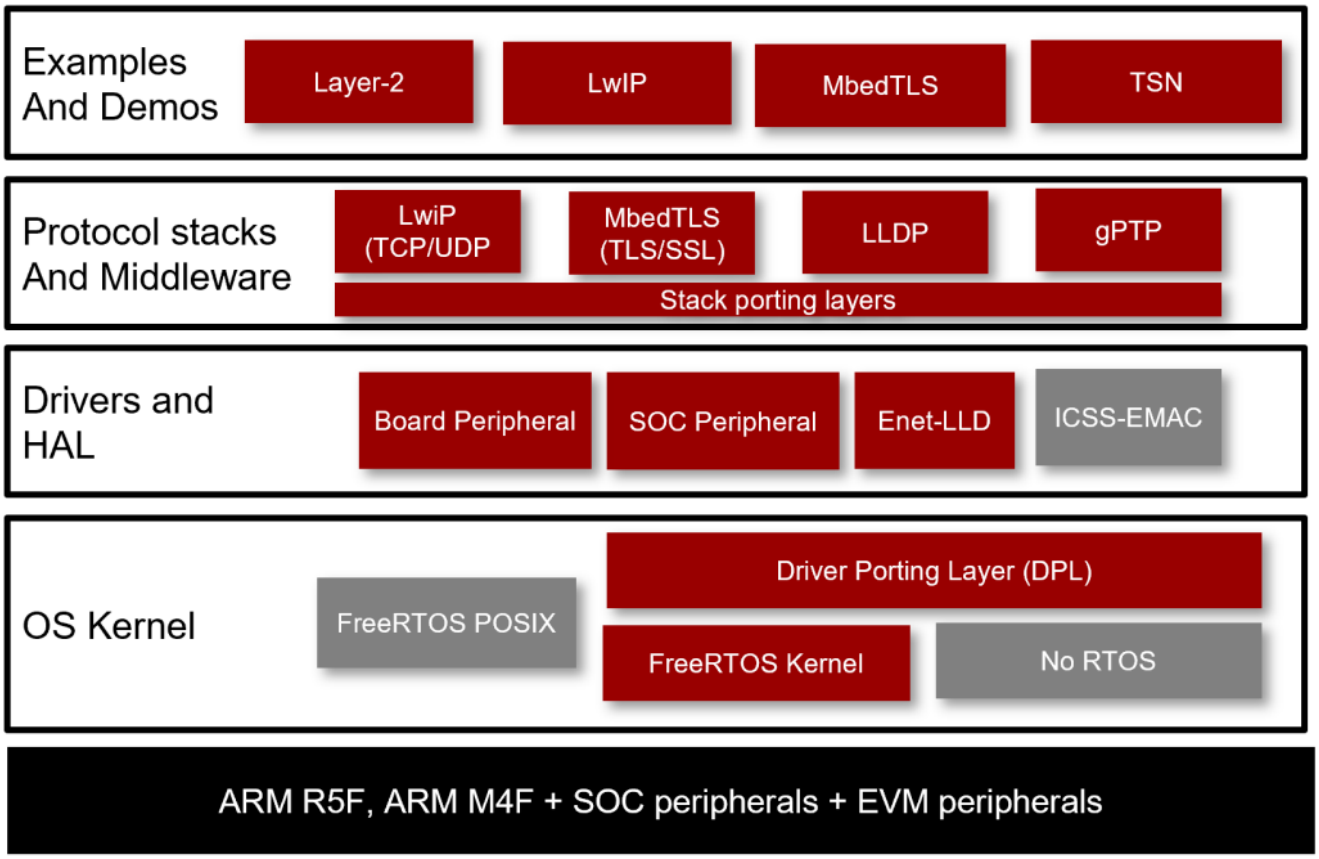


**Figure 3-3. Software Block Diagram**

The CPSW applications use a five-layer software architecture. Understanding this layering is critical for isolating debug issues to the correct subsystem.

### *3.3.1 Board and Peripherals Initialization (SYSCFG)*

- SYSCFG automatically generates the pin MUX, clocking, and peripheral initialization code
- The configuration is done using the GUI (.syscfg file)

### *3.3.2 CPSW Configuration (ENET-LLD)*

- Initializes CPSW3G hardware and PHY
- Configures port control registers, ALE entries, port states (forwarding or blocked)
- Brings up external MAC ports with PHY drivers and link monitoring
- Sets up TX/RX DMA descriptors for host port
- Registers interrupts for link change events

### *3.3.3 Operating System (FreeRTOS or NoRTOS)*

- Provides CPU start-up, interrupt handling, MPU, cache, semaphore, timer, and queue APIs
- FreeRTOS adds task scheduling
- ENET-LLD is OS agnostic; middleware stack depends on OS

### *3.3.4 Middleware Stack (LwIP, Arm® Mbed™ Platform TLS, TSN)*

- Protocol stacks:
  - LwIP (TCP/IP), Arm® Mbed™ Platform TLS (TLS/DTLS)
  - gPTP
  - eAVB
  - LLDP, and so forth

- Sits between application and the ENET-LLD

### 3.3.5 Application Layer

- Creates TX and RX tasks for packet handling
- Manages packet allocation, freeing, and buffer recycling (zero-copy where possible)
- Programs ALE in no-learning mode for deterministic switching
- Configures the CPSW classifiers for packet matching
- Initializes middleware stacks
- Defines IET, EST, inter-VLAN configurations (hardware configuration executed by ENET-LLD), and so forth

# 4 Debugging Hardware and Software

This section outlines the different steps for debugging hardware and software.

## 4.1 Hardware Debugging

### *4.1.1 Schematic Review Checklist*

Before finalizing the schematics, verify the following hardware implementation details in the following subsections.

#### 4.1.1.1 Management Data Input/Output (MDIO and MDC)

- **Pullup Resistors:** Verify that the appropriate pullup resistors are present on both the MDIO and MDC lines.

#### 4.1.1.2 RGMII Interface

- **Transmit (TX) Signals:** Place a series of termination resistors close to the MAC or MCU.
- **Receive (RX) Signals:** Place a series of termination resistors close to the PHY chip.
- **Resistor Presence:** Confirm that a resistor placeholder (discrete or array) is populated for every signal line, even if the line is currently designated as a 0Ω jumper.

Contact your FAE and submit the schematics for review by TI experts.

### *4.1.2 PHY Debug*

#### 4.1.2.1 PHY Bootstrap Settings

Bootstrap (strapping) pins configure the PHY behavior during power-up, before any MDIO communication occurs. These pins are sampled once at power-on or reset, with the logic levels determined by external resistors. Getting bootstrap configuration incorrect is a common cause of Ethernet failures that are difficult to diagnose.

**Common Bootstrap Settings:**

- **PHY Address**: Sets the MDIO address (typically 0-31). The address must match the address that the software expects. Verify the software configures the PHY address that is bootstrapped in the hardware.

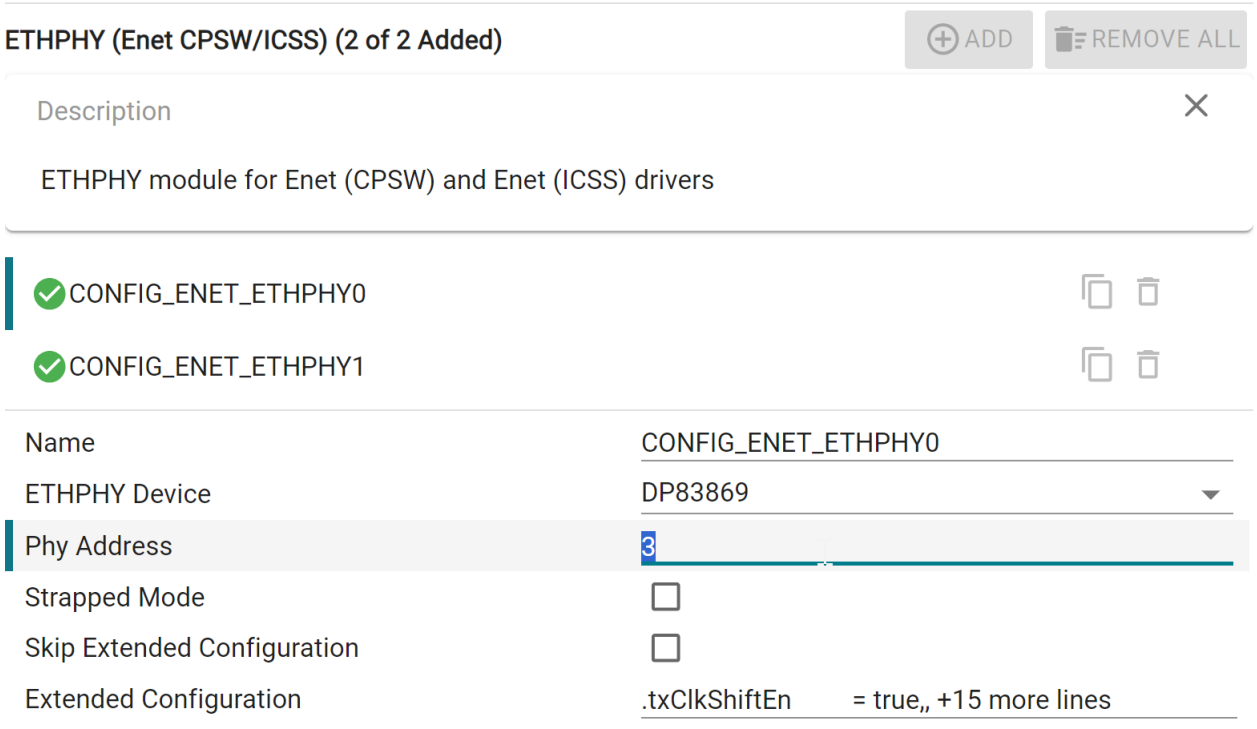


**Figure 4-1. PHY Address Configuration**

- **Interface Mode**: Selects RGMII, RMII, MII, and so forth. The interface mode must match the MAC configuration.

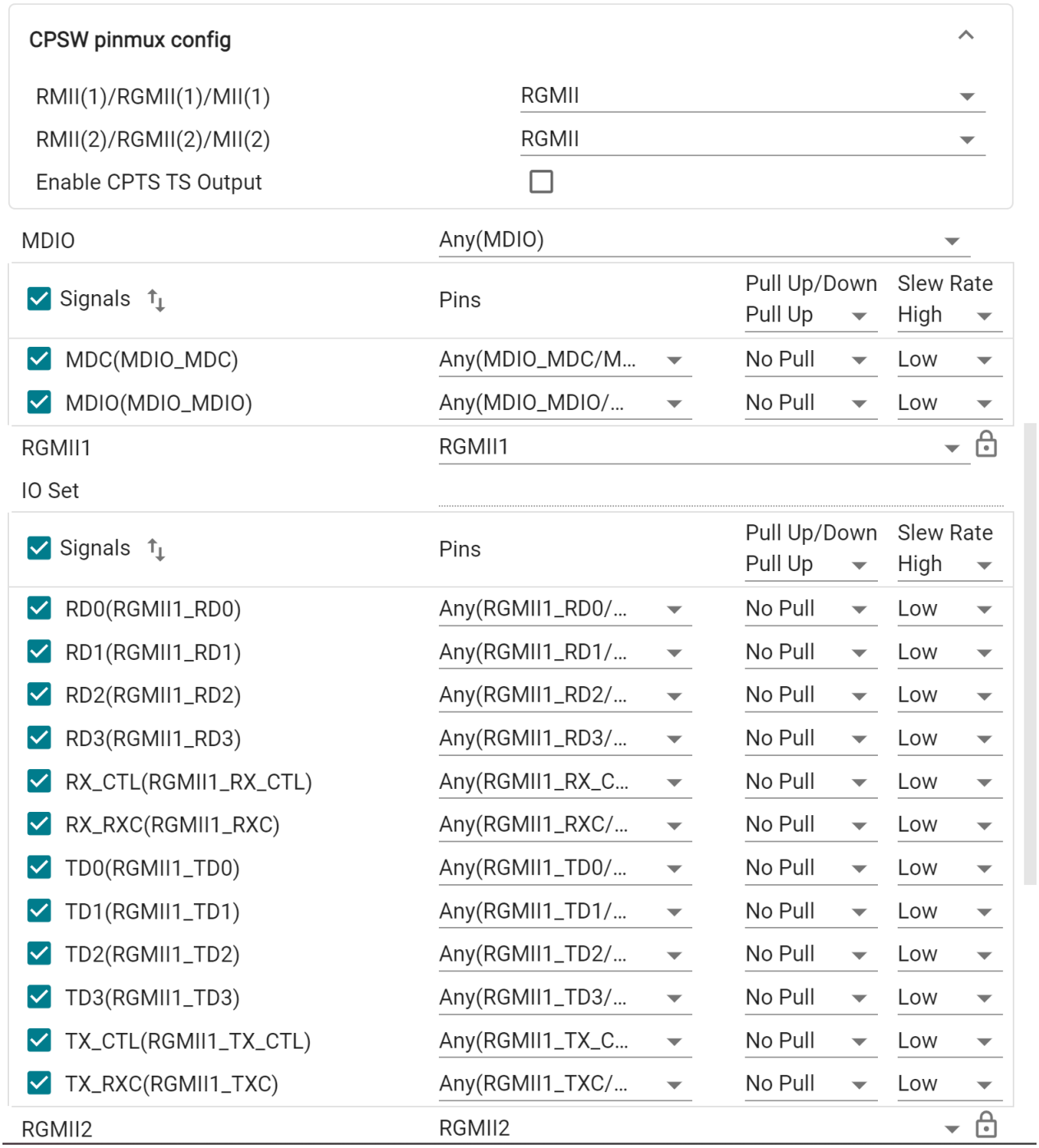


**Figure 4-2. Pin MUX Configuration**

**Common Issues**:

- **Wrong resistor values**: Using 4.7k instead of 10k can result in marginal logic levels that fail intermittently or at temperature extremes.
- **Floating pins**: Missing pull-up resistors cause undefined behavior. Some Ethernet PHYs have weak internal pullups, and these pull-up resistors are unreliable.
- **Incorrect pullup direction**: A pullup occurs where a pulldown is required and puts the PHY in the wrong mode entirely.

**Verification Steps**:

1. Check the bootstrap table of the PHY in the appropriate datasheet. Compare the required resistor values and configurations against the appropriate schematic.
2. Measure bootstrap pin voltages during power-up with an oscilloscope or multimeter. Verify the voltages reach valid logic high (>2.0V) or low (<0.8V) levels.
3. Read the PHY configuration registers through the MDIO after initialization. Compare the actual mode (interface type, address) against the expected values.
4. If the bootstrap settings are incorrect, check for schematic errors, incorrect resistor population, or PCB shorts or opens.

**Bootstrap Versus MDIO Conflicts**:

Some PHY configuration parameters can only be set using bootstrap and are locked after power-up. Interface mode selection (RGMII, RMII, MII) is typically bootstrap-only. If software attempts to configure these parameters differently through MDIO, the PHY silently ignores the writes, creating a mismatch between software expectations and actual hardware state.

**Debug Approach**:

- Read back the PHY registers after MDIO configuration to verify writes took effect.
- Compare bootstrap-configured interface mode against software configuration.
- Check if MDIO writes to read-only registers are returning unexpected values.
- Verify PHY datasheet specifies which settings are bootstrap-only versus MDIO-configurable.

**Auto-negotiation Bootstrap**:

Many PHYs provide bootstrap pins to enable or disable auto-negotiation or force specific speeds (10Mbps, 100Mbps, or 1000Mbps). Incorrect bootstrap configuration can disable auto-negotiation when software expects that auto-negotiation is enabled, or force a speed incompatible with the link partner. This manifests as a link never establishing or establishing at the wrong speed.

**Debug Approach**:

- Read the PHY basic control register (register 0x00) to verify the current auto-negotiation state.
- Check if the auto-negotiation complete bit is set in the basic status register (0x01).
- Verify bootstrap matches the auto-negotiation configuration of the software.
- Test with forced speed or duplex on both link partners to isolate auto-negotiation issues.
- The CPSW3G variant supports two external MAC ports, each requiring a PHY with a unique MDIO address. Each PHY must be bootstrapped to a different address (typically 0 and 1, or 0 and 3). Common failures occur when both PHYs share the same bootstrap address, causing MDIO bus collisions where both PHYs respond simultaneously, corrupting read data.
- Read the PHY ID registers (0x02, 0x03) at each expected address and verify the correct PHY responds (refer to section xx.xx to learn how to read PHY registers).
- Check if an unexpected PHY responds at the wrong address (indicates incorrect bootstrap).
- Use the MDIO bus analyzer or scope to detect multiple PHYs driving the bus simultaneously.
- Verify bootstrap resistors are independent for each PHY, not shared on same net.
- Confirm the software MDIO address configuration matches the PHY bootstrap addresses.

#### 4.1.2.2 Trace Length

Trace length in Ethernet PHY design refers to the physical length of the copper tracks on a PCB connecting the Ethernet PHY (physical layer) chip to the magnetic modules (transformers) and the RJ45 jack, or to the MAC (media access control) chip. The RGMII specification operates at a clock frequency of 125MHz. Since RGMII utilizes double data rate (DDR) signaling, data is sampled on both the rising and falling edges of the clock. This DDR signaling creates a narrow 4.0ns data window.

To attain clean data latching and maximum setup and hold margins, the clock signal must arrive exactly in the center of this data window (a center-aligned clock, requiring a 1.5ns to 2.0ns delay relative to the data).

Depending on the hardware architecture, this timing alignment can be achieved using one of two methods:

1. RGMII v1.3 (physical trace delay): The transmitting device outputs clock and data signals edge-aligned (arriving at the same time). To achieve the required 1.5–2.0ns center alignment, the hardware engineer must

manually add the physical trace length to the clock line on the PCB. This method is deprecated and not recommended for modern layouts.

2. RGMII v2.0 / RGMII-ID (internal delay): The internal delay feature compensates for RGMII trace length mismatches on the PCB. RGMII uses source-synchronous clocking where the clock and data must arrive at the receiver within a tight timing window (typically ±0.5ns). When the PCB traces clock and data have different lengths, this difference introduces skew that can violate the setup and hold margins of the receiver. **CPSW_CONTROL.RGMII*_ID_MODE**, when set to 1, enables the internal delay mode for the transmit path of the corresponding RGMII port. This mode provides a phase shift of a quarter cycle between clock and data.

For AM26x devices, internal delay can be enabled using SYSCFG.

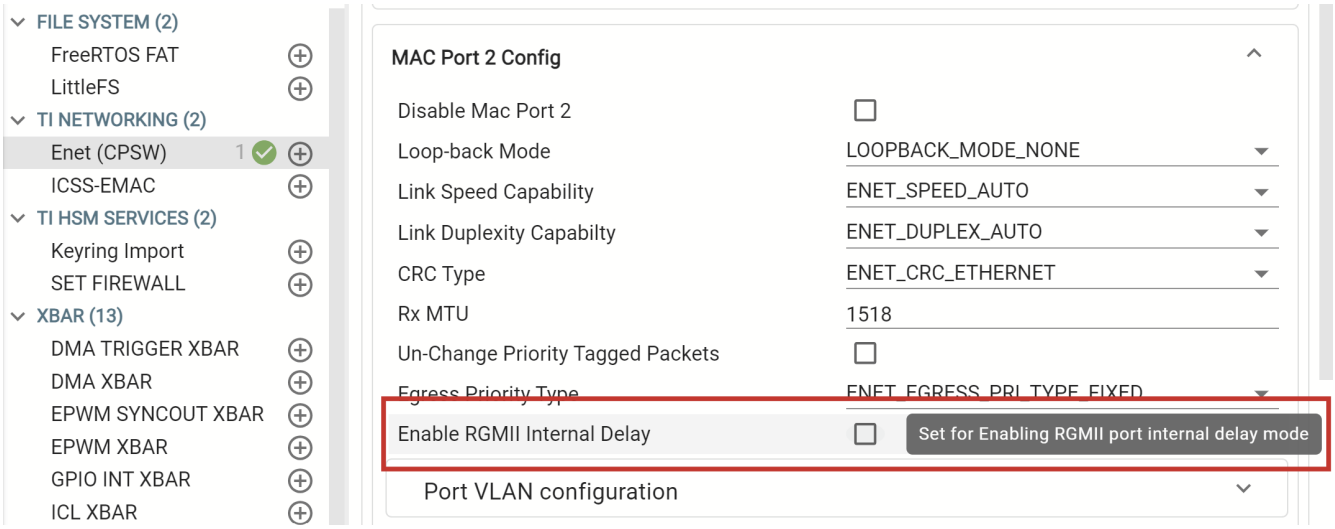


**Figure 4-3. SYSCFG RGMII Delays**

Writing 1'b1 disables the internal clock delays, and those delays must be handled onboard.

A similar internal delay feature is also available from a PHY perspective. There are timing path operating modes. Timing paths can operate in either aligned mode or shift mode. Aligned mode introduces zero clock skew. Shift mode allows clock skew adjustments in ≅0.25ns increments through the register settings. Configure these modes using the RGMII control register (RGMIICTL) at 0x0032. If using shift mode, adjust the specific clock skew values using the RGMII delay control register (RGMIIDCTL) at address 0x0086. This entirely depends on the PHY being used. Please refer to the PHY datasheet for more details.

The ENET-LLD PHY drivers provide support for these extended configurations. Extended configurations are PHY specific configurations. For example, the dp83867 PHY supports such features. This can be utilized by configuring the extended configuration data structure.

```
typedef struct Dp83867_Cfg_s
 {
 /*! Enable TX clock shift */
 bool txClkShiftEn;
 /*! Enable RX clock shift */
 bool rxClkShiftEn;
 /*! TX delay value */
 uint32_t txDelayInPs;
 /*! RX delay value */
 uint32_t rxDelayInPs;
 /*! ...other fields */
 } Dp83867_Cfg;
```

There is provision for configuring extended configurations in SYSCFG for AM26x devices.

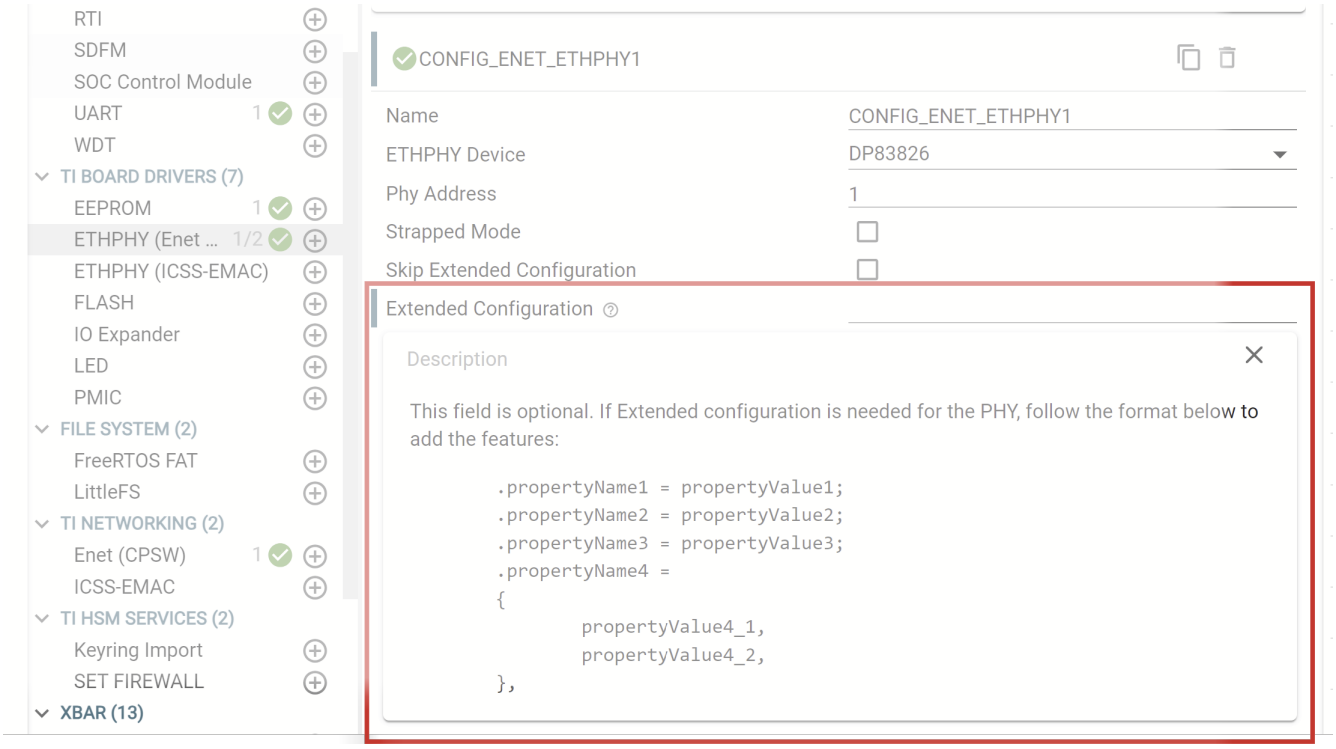


**Figure 4-4. PHY Extended Configuration**

Users can also check the *Timing Requirements* section in the PHY datasheet and verify that the board design adheres to the same timing requirements.

#### 4.1.2.3 Clock Configuration

Clock stability is the single most important starting point for debugging hardware Ethernet issues. Verify that the clock configurations are correct. To satisfy some other peripheral, the clock configuration of the CPSW clock and the source PLL can sometimes change during development. Always compare the clock configuration that is already made in the out-of-box SDK examples and the CPSW example. This can be done by taking the TOP RCM register dump of PLLs and the MSS RCM register dump of CPSW from your application and by comparing the dump with the same in SDK examples.

#### 4.1.2.4 Mode Settings

Master and slave mode determines which device provides the master clock for the physical layer connection.

- **Master PHY** uses the internal local oscillator to generate the transmit clock.
- **Slave PHY** locks onto the incoming signal from the master using a phase-locked loop (PLL) and recovers the clock to synchronize the individual transmissions.

In gigabit Ethernet, one end of the link must act as master (transmits the clock) and the other as slave (receives and locks to the clock). This configuration can be established through different means.

- **Hardware Strapping or Register Bit Settings**: The user can manually force a device to act exclusively as a master or a slave. This is common in automotive Ethernet (like 100BASE-T1) or fixed infrastructure. Basically, if the speed and duplexity is forced then this configuration must be enabled in hardware.
- **Auto-Negotiation Resolution**: By default, devices use a priority hierarchy to make decisions. For example, a multiport switch typically takes priority as the master over a single-port network interface card (NIC). If two identical devices connect, these devices use a random seed generator to determine which is the master.

From a SW perspective, this configuration can be done through a SYSCFG extended PHY configuration.
For example, the DP83TG721 PHY has this feature of configuring master or slave mode using an extended configuration. The same PHY also has the provision to configure the master or slave mode using HW strapping.

More details on these configurations are available in the specific datasheet for the PHYs. The configuration made using the extended configuration is used to set the PMA_CTRL register.

In case you are using a media convertor, the media convertor EVM also has a master or slave jumper which must be set correctly. An example is shown in Figure 4-5 for the DP83TG720/TG721 evaluation module for a 1Gbps media convertor.

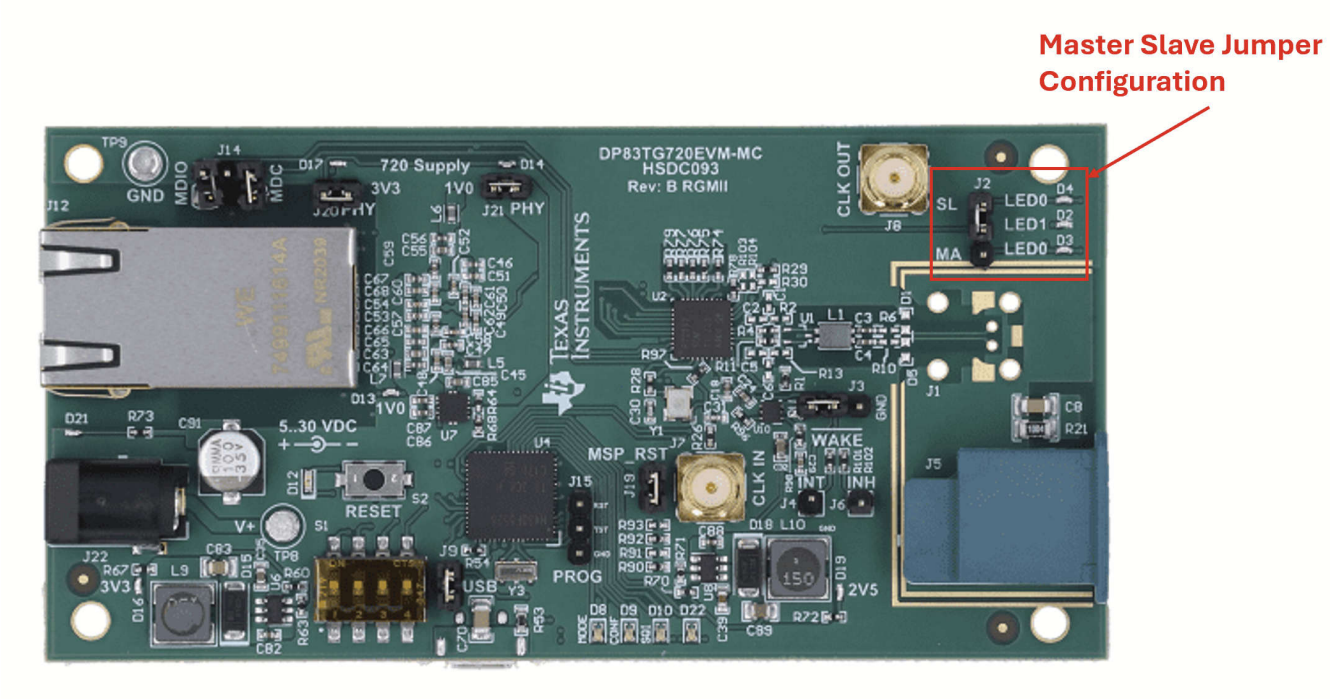


**Figure 4-5. PHY Jumper Configuration**

Figure 4-6 shows an example for configuring an MDI master or slave configuration using SYSCFG.

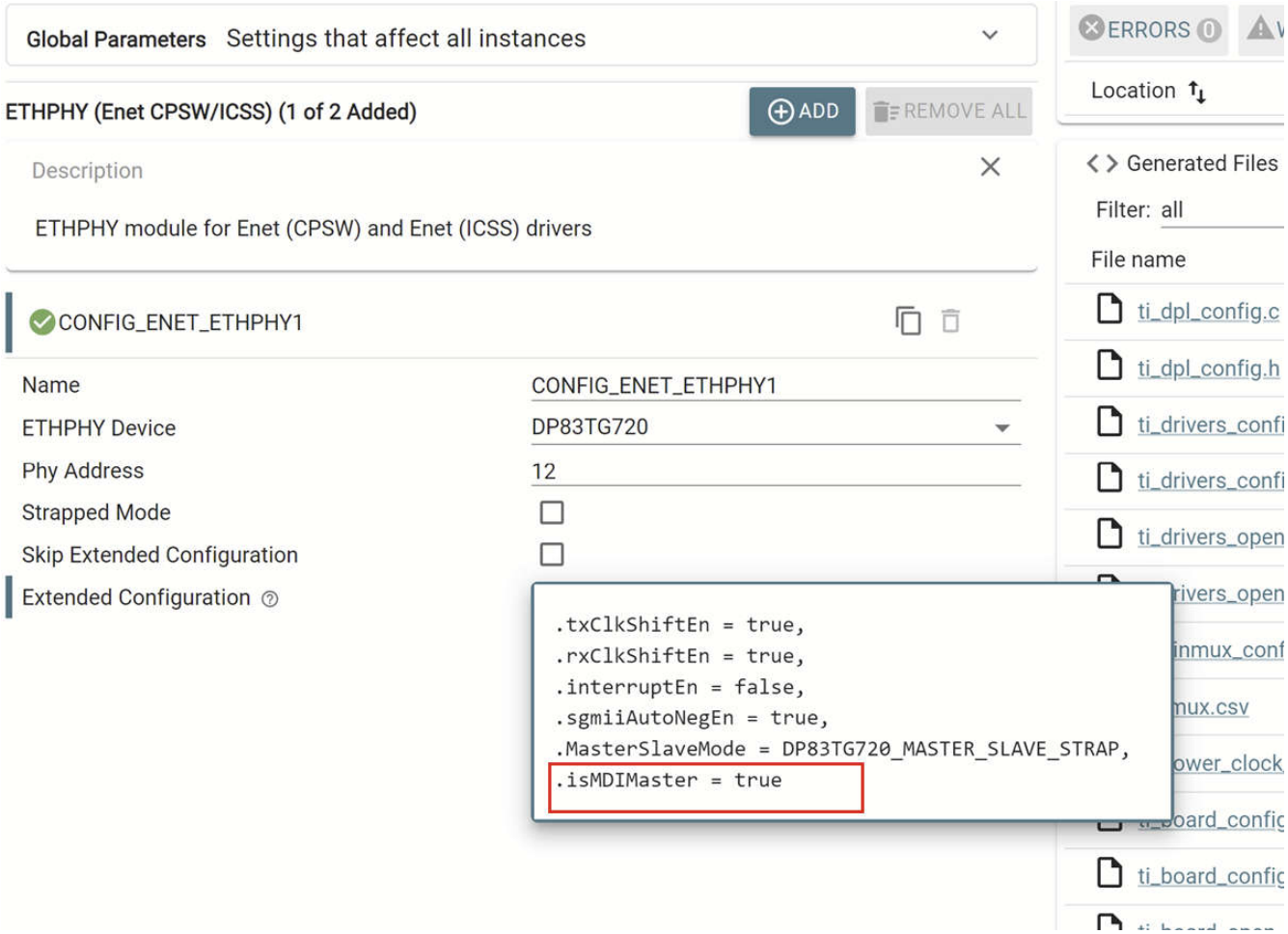


**Figure 4-6. SYSCFG Setting for MDI Master**

#### 4.1.2.5 IO MUX and SW Switch Settings

Certain TI EVMs, like the AM263P control card (CC), route signals to the PHY through an IO expander and switches. This action is done to reduce space and reuse many hardware entities. When working on a TI EVM, verify that these settings are correct. The TI AM263P CC can be taken as an example in this case. The correct MUX selection, switch positions, and resistor configurations are mentioned in the EVM user guide. In the case of the AM263P device, **see the *Ethernet Routing*** section, in particular the *SoC Source* and *Destination* columns. Depending on your requirement, refer to the table and make the necessary configurations.

#### 4.1.2.6 PHY Troubleshooting Guides

The following lists some debug guides created by TI for helping debug PHY issues:

- DP83867 Troubleshooting Guide: Diagnostic steps for the DP83867; covering voltage checks, reset signals, and bootstrap configurations.
- DP83825 Troubleshooting Guide: Schematic checklist and device health checks for the DP83825.
- DP83TC812 Troubleshooting Guide: Diagnostic guide for automotive DP83TC812 devices.
- DP83TG720S-Q1 SGMII Troubleshooting: SGMII-specific debug steps and bootstrap verification.
- DP83826A Troubleshooting Guide: Probing instructions for strap pins and mode configurations.
- DP83869 Troubleshooting Guide: Serial management interface (SMI) read/write sequencing.
- DP83TD510E Troubleshooting Guide: Single and dual-supply voltage checks for the DP83TD510E.
- FAQ on Solving Link Issues with PHYs
- FAQ on Ethernet PHY Strapping

#### 4.1.2.7 Custom Pin MUX Settings

Pin muxing of MAC to the PHY interface is different for custom boards. Issues typically occur due to incorrect pin MUX settings. So, pin muxing must be done correctly for MDIO and any RGMII, MII, and RMII. The pin muxing can be done in the CPSW section of SYSCFG. The MDIO pins are in the *MDIO* section. The MAC to PHY interface pins are in the *IO Set* section. In addition to the pin MUX settings, the MAC to the PHY interface type must also be chosen, since pin muxing ultimately depends on this factor.

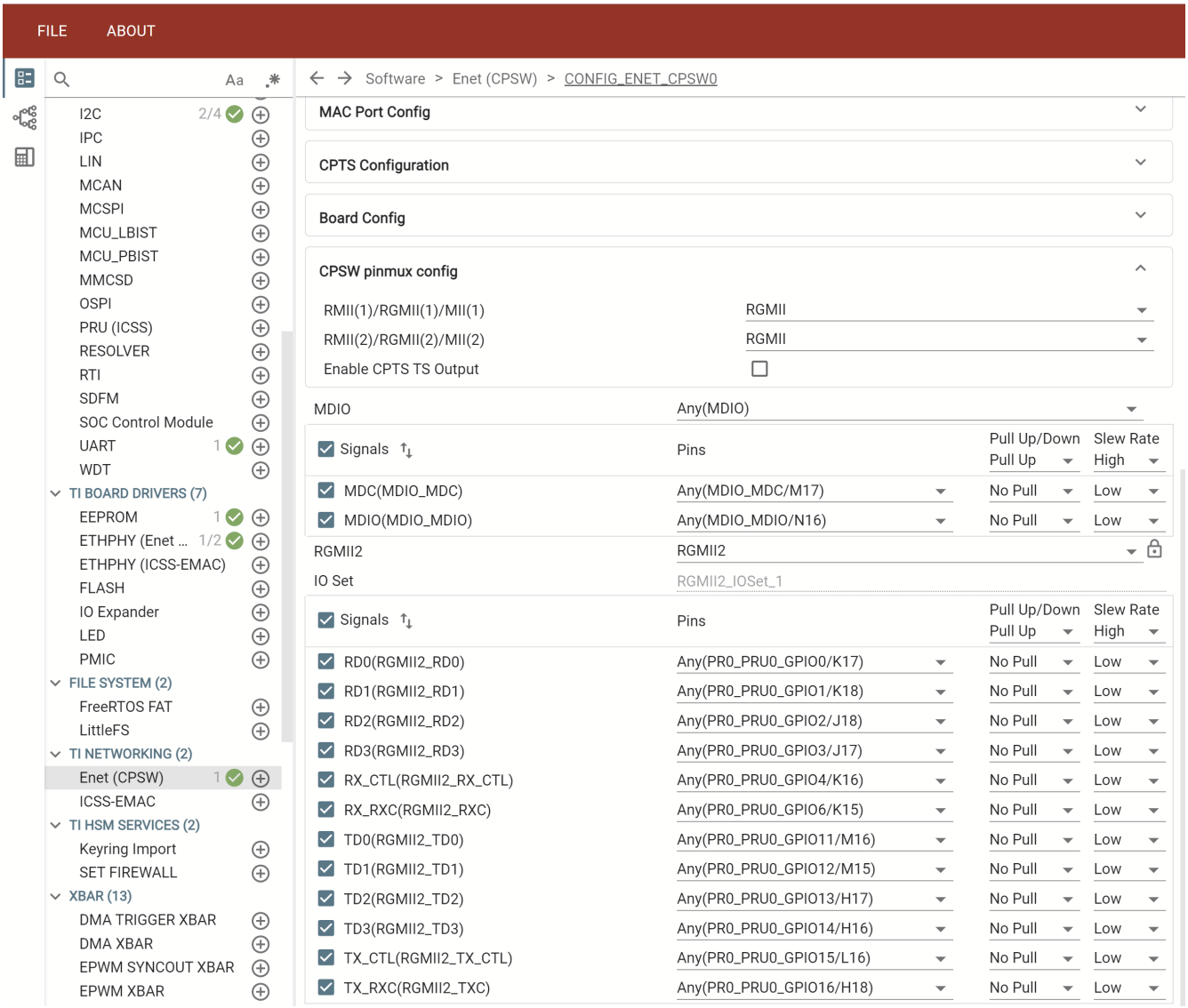


**Figure 4-7. SYSCFG Custom Pin MUX**

The board_config.c file must be developed correctly. This file sets up the MAC port. Some board designs use an IO expander to route MDIO and data signals from MAC to the PHY. The routing logic must be handled correctly in this file.

The steps for correctly developing the board_config.c file is explained in the **XREF – Custom Board Enablement in SYSCFG** section of the document.

### *4.1.3 Test Setup*

When debugging Ethernet applications, one crucial aspect is to set up the test flawlessly. Always verify the link partner for the device under debug is configured properly. The following lists important checks to complete when setting up the test:

- Verify the Ethernet cable is working.
- Verify the DHCP server is running on the other end when running lwIP applications and requiring automatic IP assignment.
- Verify a firewall is not blocking packets sent by the DUT if unable to see packets on the packet sniffing tools.
- Verify the link status of the partner is UP.
- Verify the capabilities of the link partner are known and configured properly. For example, if automatic negotiation is not supported, the PHY on the AM261x must not be set to automatic negotiate. In such cases, the settings for speed, duplexity, and interface can be forced using the SYSCFG of the application.

### *4.1.4 Software Debugging*

#### 4.1.4.1 Using GEL Scripts in CCS

CCS has GEL scripts for retrieving specific CPSW related information such as PHY register dumps, CPSW statistics, ALE table dumps, and so forth. To use a GEL script in CCS, make sure your CCS debug session is running and the Ethernet application is loaded and running on the CPU. To get a GEL script dump, follow the steps listed below:

1. Verify the CPU is in the HALTED state.
2. Navigate to *Scripts* in CCS.

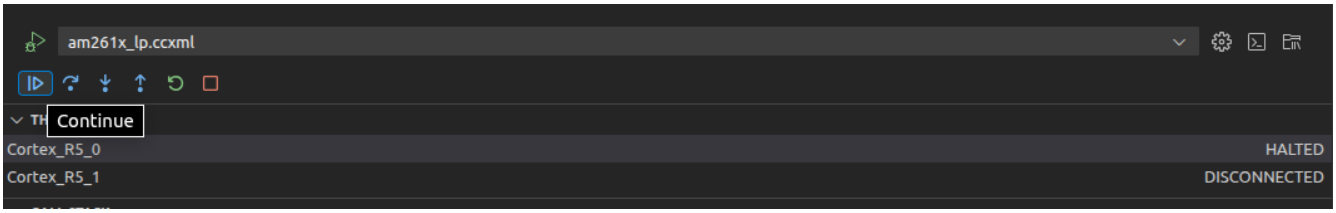


**Figure 4-8. CCS Debug Session**

3. Select the GEL script to run.

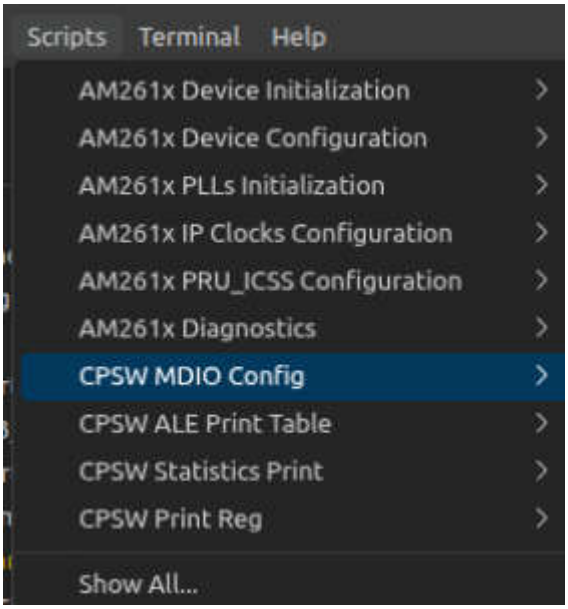


**Figure 4-9. CCS GEL Script Selection**

**Note**

The GEL output section shows the result of the GEL script. As seen in the Figure 4-10 example, the screen shot shows the ALE table dump.

```
Cortex_R5_0: ------------------------------------------------
Cortex_R5_0: -------CPSW3G ALE TABLE----------------------
Cortex_R5_0: ------------------------------------------------
Cortex_R5_0: ------------------------------------------------
Cortex_R5_0:  Entry 0 - Unicast
Cortex_R5_0: ------------------------------------------------
Cortex_R5_0: PORT_NUMBER      = 0
Cortex_R5_0: BLOCK            = 0
Cortex_R5_0: SECURE           = 1
Cortex_R5_0: UNICAST_TYPE     = 0
Cortex_R5_0: ENTRY_TYPE       = 1
Cortex_R5_0: UNICAST_ADDR     = 0x000070FF 0x761FCF9E
Cortex_R5_0: ------------------------------------------------
Cortex_R5_0:  Entry 1 - Unicast
Cortex_R5_0: ------------------------------------------------
Cortex_R5_0: PORT_NUMBER      = 0
Cortex_R5_0: BLOCK            = 0
Cortex_R5_0: SECURE           = 1
Cortex_R5_0: UNICAST_TYPE     = 0
Cortex_R5_0: ENTRY_TYPE       = 1
Cortex_R5_0: UNICAST_ADDR     = 0x000070FF 0x761FCF9F
Cortex_R5_0: ------------------------------------------------
Cortex_R5_0:  Entry 2 - Multicast
Cortex_R5_0: ------------------------------------------------
Cortex_R5_0: PORT_MASK        = 0x00000007
Cortex_R5_0: SUPER            = 0
Cortex_R5_0: MCAST_FWD_STATE  = 0
Cortex_R5_0: ENTRY_TYPE       = 1
Cortex_R5_0: MULTICAST_ADDR   = 0x0000FFFF 0xFFFFFFFF
Cortex_R5_0: Completed analysis of 512 ALE entries
```

**Figure 4-10. CCS ALE Table GEL Script Output**

Similarly, GEL scripts can be used to retrieve a dump of the PHY registers, and the CPSW statistics. This eases the debug by providing crucial register level information which can be very useful.

Figure 4-11 shows the PHY register dump for the TI DP83869 PHY connected to the AM261x-LP (verify that the PHY address is entered correctly and that the CPSW variant is chosen as CPSW3G).

```
Cortex_R5_0: PHY_REG ADDRESS   = 0
Cortex_R5_0: PHYREG READ VALUE =  0x00001140
Cortex_R5_0: ******************************
Cortex_R5_0: PHY_REG ADDRESS   = 1
Cortex_R5_0: PHYREG READ VALUE =  0x00007949
Cortex_R5_0: ******************************
Cortex_R5_0: PHY_REG ADDRESS   = 2
Cortex_R5_0: PHYREG READ VALUE =  0x00002000
Cortex_R5_0: ******************************
Cortex_R5_0: PHY_REG ADDRESS   = 3
Cortex_R5_0: PHYREG READ VALUE =  0x0000A0F3
Cortex_R5_0: ******************************
Cortex_R5_0: PHY_REG ADDRESS   = 4
Cortex_R5_0: PHYREG READ VALUE =  0x000001E1
Cortex_R5_0: ******************************
Cortex_R5_0: PHY_REG ADDRESS   = 5
Cortex_R5_0: PHYREG READ VALUE =  0x00000000
Cortex_R5_0: ******************************
Cortex_R5_0: PHY_REG ADDRESS   = 6
Cortex_R5_0: PHYREG READ VALUE =  0x00000064
Cortex_R5_0: ******************************
Cortex_R5_0: PHY_REG ADDRESS   = 7
Cortex_R5_0: PHYREG READ VALUE =  0x00002001
Cortex_R5_0: ******************************
Cortex_R5_0: PHY_REG ADDRESS   = 8
Cortex_R5_0: PHYREG READ VALUE =  0x00000000
Cortex_R5_0: ******************************
Cortex_R5_0: PHY_REG ADDRESS   = 9
Cortex_R5_0: PHYREG READ VALUE =  0x00000200
Cortex_R5_0: ******************************
Cortex_R5_0: PHY_REG ADDRESS   = 10
Cortex_R5_0: PHYREG READ VALUE =  0x00000000
Cortex_R5_0: ******************************
Cortex_R5_0: PHY_REG ADDRESS   = 11
Cortex_R5_0: PHYREG READ VALUE =  0x00000000
Cortex_R5_0: ******************************
```

**Figure 4-11. CCS PHY Register GEL Script Dump**

#### *4.1.4.1.1 Statistics Using GEL Scripts*

Use the steps in Section 4.1.4.1 to use GEL scripts to retrieve or reset statistics on CPSW.

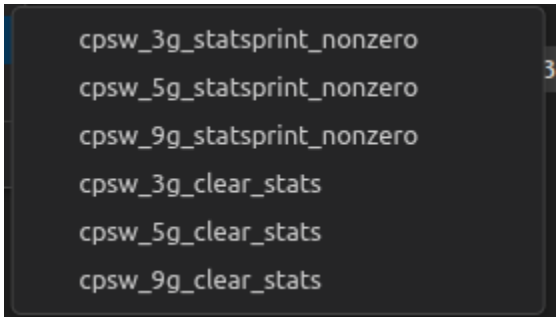


**Figure 4-12. CCS CPSW Statistics GEL Script**

The GEL script prints all the non-zero statistics for the host port and the MAC port (see the sample in Figure 4-13):

```
Cortex_R5_0:            STATS
Cortex_R5_0: --------------------------------
Cortex_R5_0:            PORT0 STATS
Cortex_R5_0: --------------------------------
Cortex_R5_0: STAT_0_RXGOODFRAMES              = 0x00000011
Cortex_R5_0: STAT_0_RXBROADCASTFRAMES         = 0x0000000C
Cortex_R5_0: STAT_0_ALE_DROP                  = 0x00000005
Cortex_R5_0: STAT_0_RXOCTETS                  = 0x00000CC6
Cortex_R5_0: STAT_0_TXGOODFRAMES              = 0x00000037
Cortex_R5_0: STAT_0_TXBROADCASTFRAMES         = 0x00000007
Cortex_R5_0: STAT_0_TXMULTICASTFRAMES         = 0x0000002A
Cortex_R5_0: STAT_0_TXOCTETS                  = 0x0000236A
Cortex_R5_0: STAT_0_OCTETFRAMES64             = 0x00000020
Cortex_R5_0: STAT_0_OCTETFRAMES65T127         = 0x00000009
Cortex_R5_0: STAT_0_OCTETFRAMES128T255        = 0x00000007
Cortex_R5_0: STAT_0_OCTETFRAMES256T511        = 0x00000018
Cortex_R5_0: STAT_0_NETOCTETS                 = 0x00003030
Cortex_R5_0: STAT_0_PORTMASK_DROP             = 0x00000005
Cortex_R5_0: --------------------------------
Cortex_R5_0:            PORT1 STATS
Cortex_R5_0: --------------------------------
Cortex_R5_0: STAT_1_RXGOODFRAMES              = 0x00000037
Cortex_R5_0: STAT_1_RXBROADCASTFRAMES         = 0x00000007
Cortex_R5_0: STAT_1_RXMULTICASTFRAMES         = 0x0000002A
Cortex_R5_0: STAT_1_RXOCTETS                  = 0x0000236A
Cortex_R5_0: STAT_1_TXGOODFRAMES              = 0x0000000C
Cortex_R5_0: STAT_1_TXBROADCASTFRAMES         = 0x00000007
Cortex_R5_0: STAT_1_TXOCTETS                  = 0x000005DC
Cortex_R5_0: STAT_1_OCTETFRAMES64             = 0x00000020
Cortex_R5_0: STAT_1_OCTETFRAMES65T127         = 0x00000009
Cortex_R5_0: STAT_1_OCTETFRAMES128T255        = 0x00000007
Cortex_R5_0: STAT_1_OCTETFRAMES256T511        = 0x00000013
Cortex_R5_0: STAT_1_NETOCTETS                 = 0x00002946
Cortex_R5_0: STAT_1_ALE_UNKN_UNI              = 0x00000001
Cortex_R5_0: STAT_1_ALE_UNKN_UNI_BCNT         = 0x0000015A
Cortex_R5_0: STAT_1_ALE_UNKN_MLT              = 0x00000011
Cortex_R5_0: STAT_1_ALE_UNKN_MLT_BCNT         = 0x0000055A
Cortex_R5_0: STAT_1_ALE_UNKN_BRD              = 0x00000001
Cortex_R5_0: STAT_1_ALE_UNKN_BRD_BCNT         = 0x00000154
```

**Figure 4-13. CCS CPSW Statistics GEL Script Dump**

The exact details of each statistic can be inferred from the TRM for the device.

#### *4.1.4.1.2 Statistics Using Expressions*

The CPSW statistics can also be viewed directly from the CCS expressions window. Add the following to access the window: (CSL_Xge_cpswRegs *)0x52820000.

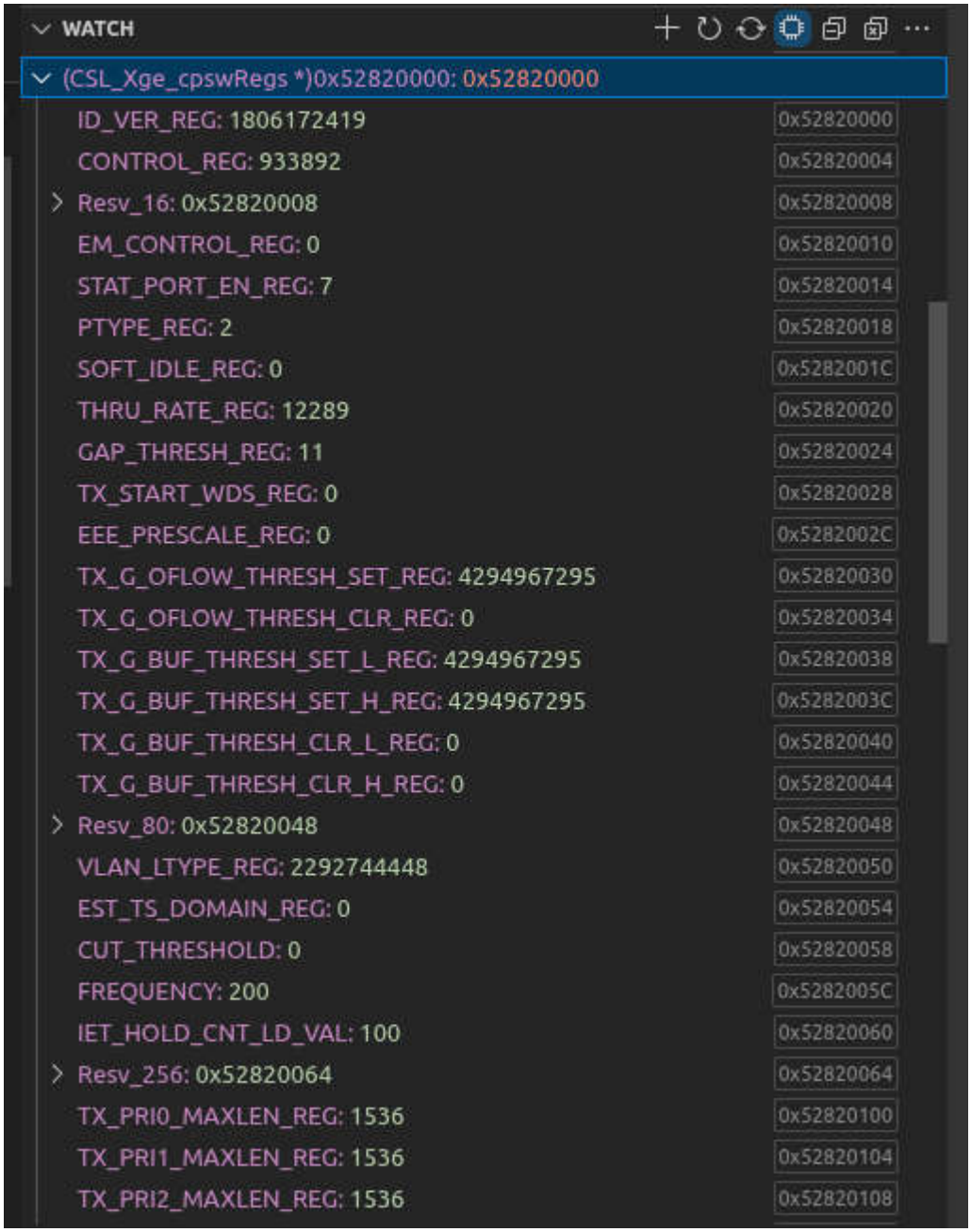


**Figure 4-14. CPSW Statistical Expression**

Follow the instructions at the FAQs link to initiate the CPSW statistics dump using the GEL scripts:

**FAQ Link**: How to View CPSW Registers and CPSW Stats for AM26x Devices Using GEL Script

**Note**

In the case of the AM261 device, the GEL script must be uploaded into the CCS.

For the AM263x and AM263Px devices, the GEL scripts are already available in the CCS.

### *4.1.5 Debugging Custom Ethernet Software*

Debugging CPSW Ethernet connectivity problems requires a systematic approach that combines hardware validation with deep software analysis. Even if the hardware looks perfectly fine on an oscilloscope, software errors in the ENET-LLD driver stack cause most connectivity failures. This section provides practical techniques

for debugging the AM261x CPSW software initialization sequence, data flow, and statistics collection using the MCU+ SDK ENET-LLD APIs. The examples reference actual function names and structures from the SDK to help engineers quickly locate and diagnose issues in their applications.

The AM261x MCU+ SDK uses SYSCFG to generate initialization code that properly sequences the CPSW peripheral bring-up. Understanding this flow is critical for debugging initialization failures. The SYSCFG tool generates several key files:

- ti_enet_open_close.c
- ti_enet_init.c
- ti_enet_config.c

These files contain the complete peripheral configuration for CPSW. The driver initialization begins with a call to the EnetApp_driverInit() function, this function invokes the Enet_init() function from the ENET-LLD core library. The Enet_init() function initializes the ENET utilities configuration structure and must complete successfully before any peripheral operations. Following driver initialization, the application calls the EnetApp_driverOpen() function, this is the primary entry point for opening the CPSW peripheral instance. The EnetApp_driverOpen() function first retrieves the CPSW configuration structure through the EnetApp_getCpswCfg() function and EnetApp_getCpswCfg() returns a pointer to the gEnetCpswCfg configuration, a pre-configured structure containing:

- ALE settings
- VLAN configuration
- Host port parameters
- MDIO settings

#### 4.1.5.1 Debugging Initialization Sequence

This section describes the sequence for initializing debugging.

**Note**

The CPSW initialization sequence involves multiple steps, each of which can fail silently if return codes are not checked.

1. **Step 1**: Initialization failures prevent all subsequent operations, so start here first.
   a. Set the first breakpoint at the Enet_open() function in the SYSCFG-generated code. This is typically called from the EnetApp_driverOpen() function.
   b. Step through and capture the return value when this breakpoint hits.

   **Note**

   If the Enet_open() function returns NULL instead of a valid handle, the initialization has failed.

   Common causes of failure include invalid clock configuration in SYSCFG, resource allocation failure, or a mismatch between the peripheral instance ID and what is available on your device.

2. **Step 2**: The next critical checkpoint is the EnetApp_enableHostPort() function.
   a. Set a breakpoint at the EnetApp_enableHostPort() function and verify both IOCTL operations return ENET_SOK.

   **Note**

   The first IOCTL sets the ALE port state to forwarding mode (CPSW_ALE_IOCTL_SET_PORT_STATE). If this fails with ENET_EINVALIDPARAMS, the ALE configuration structure is corrupt or was not properly initialized by SYSCFG.

   The second IOCTL (ENET_HOSTPORT_IOCTL_ENABLE) activates the individual host port. Failure here typically means the Enet_open() function did not complete successfully, even though the function returned a non-NULL handle, which indicates internal resource exhaustion.

3. **Step 3**: After host port enablement, Each MAC port must be opened using the EnetApp_enableMacPort() function.
   a. Set a breakpoint at the ENET_PER_IOCTL_OPEN_PORT_LINK IOCTL call and examine the input structure.
   b. Verify the MAC interface type (RGMII, RMII, MII) matches your hardware schematic.

**Note**

A common mistake is configuring RGMII when the board uses RMII, which causes the MAC to expect signals on pins that are not connected.

   c. Check the PHY address in the configuration structure matches the hardware strapping.

**Note**

If this IOCTL returns ENET_ETIMEOUT, the PHY is not responding on the MDIO bus at the specified address, indicating either a wrong address configuration or a hardware problem.

4. **Step 4**: The final initialization checkpoint is DMA channel opening.
   a. Set breakpoints at the EnetAppUtils_openTxCh() and EnetAppUtils_openRxCh() functions and verify both return non-NULL channel handles.

**Note**

Check the MAC address pool configuration in ti_enet_init.c if the EnetAppUtils_openRxCh() function succeeds but the MAC address allocation step fails.

**Note**

An exhausted address pool prevents the application from obtaining addresses to add to the ALE table, which silently breaks RX filtering.

#### 4.1.5.2 PHY Debugging

This section discusses the process for debugging the PHY.

1. **Step 1**: Even after initialization succeeds, PHY detection can fail if MDIO communication is not working properly. The software must successfully read the PHY registers before link establishment can occur.
   a. Set a breakpoint inside the EnetApp_getPhyAliveStatus() function called during PHY detection. This function issues the ENET_MDIO_IOCTL_IS_ALIVE IOCTL for each potential PHY address.
   b. Step through the loop and examine the alive boolean output parameter after each iteration. If all addresses return false, no PHYs are responding on the MDIO bus. This response is potentially due to a software problem (MDIO clock frequency configured incorrectly in SYSCFG) or hardware (broken traces, wrong pullups on MDIO data lines). To prevent problems with hardware, refer to the previous *Hardware Debugging* section.
2. **Step 2**: If the EnetApp_getPhyAliveStatus() function succeeds and detects the PHY, but ENET_PER_IOCTL_OPEN_PORT_LINK still fails with ENET_ETIMEOUT, the problem is likely in the register read operations of the PHY driver. The ENET-LLD reads the PHY ID registers 0x02 and 0x03 to identify the device. If these reads timeout, the MDIO timing is marginal.
   a. Check the MDIO clock divider configuration in SYSCFG and verify the MCU produces a frequency within the specified range (typically 1–2.5MHz) of the PHY. Setting the frequency too low or high can cause setup or hold violations that produce intermittent read failures.

#### 4.1.5.3 MAC Port Debugging

This section describes the process for debugging the MAC port.

1. **Step 1**: When link never establishes after initialization.
   a. Begin by verifying the MAC port opened successfully.

b. Set a breakpoint at the Enet_open() function and step through to the return value. ENET_SOK indicates success, but you must also verify the MAC port handle is non-NULL.

2. **Step 2**: After the Enet_open() function succeeds.
   a. Set a breakpoint at the EnetMacPort_open() function or the port-specific variant like the Cpsw_openPort() function.
   b. Inspect the macCfg parameter before the call executes.

**Note**

The macCfg.layerType field must match your hardware configuration exactly: ENET_MAC_LAYER_GMII_RGMII for RGMII interfaces or ENET_MAC_LAYER_RMII for RMII.

   c. Check the linkCfg.speed and linkCfg.duplexity fields.

**Note**

If these fields are set to fixed values like ENET_SPEED_100MBIT and ENET_DUPLEX_FULL, but your link partner or PHY is configured for automatic negotiation, link never comes up. The most reliable approach is setting the speed to ENET_SPEED_AUTO and the duplexity to ENET_DUPLEX_AUTO, then letting the PHY handle negotiation. A mismatched speed or duplex between the MAC configuration and the PHY capability causes perpetual link-down states.

3. **Step 3**: When link appears established in the PHY registers but no traffic flows.
   a. Verify the MAC port is actually enabled.
   b. Set a breakpoint at the Cpsw_enablePort() function, or equivalent, after link-up is detected.

**Note**

If this function is never called, your link status callback is not triggering port enable, which is required even after a successful MAC initialization.

   c. Check the return code from the Cpsw_enablePort() function.

**Note**

ENET_EPERM indicates the port was already enabled, which is harmless. ENET_EINVALIDPARAMS means the port number or configuration is wrong. A frequent configuration error is enabling the MAC port before the PHY reports link-up. The MAC port must remain disabled until the PHY link callback fires and confirms link establishment. Enabling the port prematurely causes the MAC to transmit into a disconnected PHY, resulting in dropped packets and no error indication.

   d. Set a watch point on your link status variable and verify the point transitions to true before the cpsw_enablePort() function is called.

**Note**

If the sequence is reversed, restructure your initialization to wait for the PHY link-up before enabling the MAC port.

4. **Step 4**: To verify correct MAC port configuration after initialization.
   a. Read back the port registers using the debug console or a memory browser.

**Note**

The MAC mode register shows the interface type matching your hardware. The MAC control register indicates the port is enabled only after link-up. If these registers show unexpected values, the MAC configuration structure was not applied correctly, possibly due to using a default configuration instead of the hardware-specific settings. Always explicitly set the *layerType*, *speed*, and *duplexity* fields rather than relying on structure initialization defaults. When link is up and initialization succeeds, but the transmitted packets never appear on the wire, the problem lies in the TX submission path or DMA descriptor management.

5. **Step 5**: Checking the packet queue
   a. Set a breakpoint at the EnetDma_submitTxPktQ() function in your transmit function.
   b. Examine the packet queue being submitted before the call executes.
   c. Check that the queue count is non-zero by inspecting the internal queue structure.

**Note**

If the queue is empty, the problem is earlier in the code where the dequeue packets from txFreePktInfoQ. A common bug is calling the EnetQueue_deq() function on an exhausted free queue and not checking for a NULL return value, then attempting to submit an invalid packet pointer.

6. **Step 6**: If the packet queue is valid.
   a. Inspect each sgList structure of the packet.

**Note**

The first bufPtr of the segment must point to valid memory containing the Ethernet frame, and segmentFilledLen must equal the frame length. A common mistake is setting segmentFilledLen to the maximum buffer size instead of the actual frame length. This mistake causes the DMA to transmit garbage padding bytes, which potentially exceed the MTU and get dropped by the PHY or link partner.

   b. Step through and verify segmentFilledLen is between 64 and 1518 bytes for standard Ethernet frames.

7. **Step 7**: After the EnetDma_submitTxPktQ() function returns.
   a. Check the status code.

**Note**

ENET_SOK means the submission succeeded. ENET_EINVALIDPARAMS indicates the packet structure is malformed. ENET_ENOMEM means the DMA descriptor ring is full, which happens when transmitted packets are not being retrieved and freed.

   b. Set a breakpoint in the EnetDma_retrieveTxPktQ() function and verify this function is called regularly, typically from a periodic timer or after each transmit.

**Note**

If this function is never called or called too infrequently, the TX descriptor ring fills up and transmission stops silently.

#### 4.1.5.4 TX Path Debugging

This section describes the steps for debugging the TX port.

TX path failures are easier to diagnose than RX failures because the flow is simpler, but these failures still require systematic analysis to isolate whether the problem is in packet submission, DMA transfer, ALE forwarding, or MAC transmission. Understanding the complete packet flow from application to wire helps quickly identify the failure point.

1. **Step 1**: The expected TX packet flow.

When your application transmits a packet, the application follows the following sequence:

a. The application submits the packet → The EnetDma_submitTxPktQ() function returns success.
b. DMA transfers the packet from the CPU to the CPSW → The hostStats.txGoodFrames increments.
c. The ALE examines the destination MAC and determines the output port → The ALE completes the look up.
d. The MAC port transmits the packet to wire → The macStats.txGoodFrames increments.
e. Packets are dropped or never reach the wire if any of these steps fail.

2. **Step 2**:
   a. Setting a breakpoint at the EnetDma_submitTxPktQ() function in your transmit function.
   b. Examine the packet queue being submitted before the call executes.
   c. Inspect the txSubmitQ structure.
   d. Check the status code after EnetDma_submitTxPktQ() returns.
      i. If the status is ENET_SOK and the submission succeeded, and the packet is handed to DMA.
      ii. If the status is ENET_EINVALIDPARAMS and the packet structure is malformed, check if the bufPtr is valid, the segmentFilledLen is reasonable, and verify the packet queue was initialized correctly.
      iii. If the status is ENET_ENOMEM, the DMA descriptor ring is full, and the transmitted packets are not being retrieved and freed, set the breakpoint in the EnetDma_retrieveTxPktQ() function to verify this is called regularly.

3. **Step 3**: TX descriptors must be retrieved after transmission completes to free the descriptors for reuse.
   a. Set a breakpoint at the EnetDma_retrieveTxPktQ() function, typically called from a periodic timer or completion callback.

---

**Note**

If this function is never called or called too infrequently:

- The TX descriptor ring fills up overtime and the EnetDma_submitTxPktQ() function starts returning ENET_ENOMEM.
- The transmission stops silently, even though link is up. The descriptor ring is finite (configured in SYSCFG). If packets are submitted faster than completed descriptors are retrieved, the ring exhausts. Verify retrieval happens after each transmit or on a periodic timer fast enough to keep up with the transmit rate.

---

4. **Step 4**: Statistics counters show exactly where packets are dropped in the TX path.
   a. Read the host port and MAC port statistics before and after transmission. The pattern of counter changes identifies the failure point in the following scenarios:
      i. **Scenario 1**: hostStats.txGoodFrames does NOT increment →
         - The packet never reached DMA (submission failed).
         - Check the EnetDma_submitTxPktQ() function return code.
         - Verify the TX channel opened successfully (hTxCh is valid).
         - Check the DMA descriptor ring is not exhausted (retrieve completed packets).
      ii. **Scenario 2**: hostStats.txGoodFrames increments and macStats.txGoodFrames does NOT.
         - DMA accepted the packet but MAC did not transmit.
         - The ALE is dropping a packet or forwarding to a wrong port.
         - Check the hostStats.portMaskDrop counter.
         - If portMaskDrop increments, destination MAC entry has the wrong port mask.
         - Dump the ALE table and verify the destination MAC address entry exists with the correct output port.
      iii. **Scenario 3**: Both hostStats and macStats txGoodFrames increment, but link partner identifies nothing.
         - The packets transmitted but did not reach link partner.
         - Check the MAC port link status (must be UP).
         - Verify the cable is connected and the link partner is listening.
         - Use the Wireshark® on link partner to confirm frames are not arriving at all.
      iv. **Scenario 4**: Link partner receives packets but identifies CRC errors.

- The packets transmitted but are corrupt.
- Check the packet buffer contents before submission.
- Verify segmentFilledLen matches the actual frame data length.
- Enable the CPSW automatic CRC append if the application does not calculate a CRC.

#### 4.1.5.5 Systematic Debugging Checklist

When transmitted packets do not reach the wire, work through these diagnostics in the following order:

1. Check the return code of the EnetDma_submitTxPktQ() function → Confirms the submission succeeded.
2. Check the hostStats.txGoodFrames function → Confirms DMA transfers to the CPSW.
3. Check macStats.txGoodFrames → Confirms MAC is transmitted to wire.
4. If the host increments but MAC does not: Check hostStats.portMaskDrop (ALE filtering issue).
5. Verify the EnetDma_retrieveTxPktQ() function is called regularly → Prevents descriptor exhaustion.
6. Check the packet structure before submission → Validates bufPtr and segmentFilledLen.
7. Verify the MAC port link is UP → Confirms the physical layer is ready.
8. Use Wireshark® on the link partner → Confirms the frames reach the destination.

This methodical approach isolates whether the failure is in an application submission, DMA transfer, ALE forwarding, or MAC transmission.

##### *4.1.5.5.1 RX Path Debugging*

RX path failures are more subtle than TX failures because packets can potentially arrive at the PHY but never reach the application due to ALE filtering or DMA buffer exhaustion. Understanding the complete packet flow from wire to application helps isolate where the failure occurs. The expected RX packet flow is:

When a packet arrives from the network, the packet follows this sequence:

1. The packet arrives at an external MAC port → macStats.rxGoodFrames increments.
2. The ALE examines the destination MAC and forwards the destination to the host port → hostStats.rxGoodFrames increments.
3. DMA transfers the packet to memory → An RX interrupt fires.
4. The application retrieves the packet → The EnetDma_retrieveRxPktQ() function returns packet.

If any step fails, packets are dropped silently. Start debugging by setting a breakpoint in your RX interrupt handler or callback function, typically named something like EnetApp_rxIsrFxn(). Send a test packet to the device from a link partner. If the breakpoint never hits:

1. Check that the EnetDma_enableRxEvent() function was called on your RX channel handle during initialization.
2. Verify the interrupt routing configuration in SYSCFG matches the CPSW interrupt controller setup.
3. Read the host port statistics directly and check if rxGoodFrames is incrementing.

**Note**

If rxGoodFrames increases, but the interrupt handler never called, the problem is in the interrupt configuration not the packet reception.

If the interrupt handler breakpoint does hit, the RX hardware path is working correctly.

1. Move to the next checkpoint by setting a breakpoint at the EnetDma_retrieveRxPktQ() function inside your RX task.

**Note**

This function retrieves packets from the DMA and places those packets in a queue for application processing.

2. Step through the retrieve call and inspect the returned queue count.

A critical, but often overlooked aspect of RX debugging, is buffer resubmission. After the application processes each received packet, the application must return the buffer to the DMA by calling the EnetDma_submitRxPktQ() function.

1. Set a breakpoint at this function call and verify the breakpoint executes for every packet retrieved. If buffers are not resubmitted:
   a. The DMA descriptor pool gradually exhausts.
   b. Subsequent packets are dropped even though the packets arrive at the host port.
   c. Manifests as the initial packets being received successfully, followed by complete RX silence.
   d. The rxBottomOfFifoDrop counter in host port statistics increments rapidly.

**Note**

Your code flow must be: Retrieve packet → Process data → Resubmit buffer. Missing the resubmit step is the most common RX implementation bug.

Statistics counters provide definitive evidence of where packets are being dropped. Read both the MAC port and the host port statistics before and after sending a test packet in the following scenarios:

**Scenario 1: MAC rxGoodFrames increments but the host rxGoodFrames does NOT →**

- The ALE is filtering packets before the packets reach the host (the most common RX failure).
- Check the hostStats.portMaskDrop counter.
- If the portMaskDrop increments: The ALE entry exists but the port mask excludes the host port.
- Dump the ALE table with CPSW_ALE_IOCTL_DUMP_TABLE and verify the destination MAC has entry with the host port included.

**Scenario 2: Both MAC and the host rxGoodFrames increment, but the application identifies nothing.**

- The packets reach DMA but the application does not retrieve the packets.
- Check the RX task is running and being scheduled by RTOS.
- Verify the EnetDma_retrieveRxPktQ() function is called in response to the RX interrupt.
- Common bug: Forgetting to post semaphore in the interrupt handler to wake the RX task.

**Scenario 3: Counters increment initially, then stop.**

- The RX descriptor pool is exhausted (missing buffer resubmission).
- Check the rxBottomOfFifoDrop counter.
- Verify the EnetDma_submitRxPktQ() function is called after processing each packet.

This methodical approach isolates the exact failure point in the RX path.

#### *4.1.5.5.2 Multicast or Broadcast Does Not Work, But Unicast Works*

Multicast and broadcast reception failures are difficult to diagnose because unicast traffic works efficiently while multicast frames vanish silently. The link is up, TX works, but specific multicast addresses are not received. The CPSW ALE requires explicit multicast entries in static mode. If the application does not add the multicast MAC address to the ALE table with the correct port mask, frames are dropped.

1. First, verify the frames reach the MAC port by reading the MAC port statistics.
2. Verify that the frames reach the host port.

**Note**

If hostStats.rxGoodFrames does NOT increment but the MAC port rxGoodFrames does, the ALE is filtering.

3. Check the hostStats.portMaskDrop counter.
4. Dump the ALE table using the GEL scripts or the IOCTL functions and look for the multicast or broadcast MAC address.
   a. Check the PortMask field:
      i. Bit 0 = Host port (0x01)

ii. Bit 1 = MAC port 1 (0x02)
iii. Bit 2 = MAC port 2 (0x04)

**Note**

If the PortMask does NOT include bit 0 (host port), the frames do not reach the CPU.

5. Add the broadcast or multicast ALE entry, if needed, using the CPSW_ALE_IOCTL_ADD_MCAST IOCTL.

```
CpswAle_SetMcastEntryInArgs mcastInArgs;
uint32_t entryIdx;
uint8_t mcastMacAddr[ENET_MAC_ADDR_LEN] = {0x01, 0x00, 0x5E, 0x00, 0x00, 0x01}; // Your multicast
MAC
mcastInArgs.addr.vlanId = 0U;
mcastInArgs.info.portMask = (1 << CPSW_ALE_HOST_PORT_NUM) | (1 << ENET_MAC_PORT_1);
mcastInArgs.info.super = false;
mcastInArgs.info.fwdState = CPSW_ALE_FWDSTLVL_FWD;
mcastInArgs.info.numIgnBits = 0;
EnetUtils_copyMacAddr(&mcastInArgs.addr.addr[0U], mcastMacAddr);
ENET_IOCTL_SET_INOUT_ARGS(&prms, &mcastInArgs, &entryIdx);
status = Enet_ioctl(hEnet, coreId, CPSW_ALE_IOCTL_ADD_MCAST, &prms);
```

- Join multicast group (if using LwIP), for LwIP applications using IGMP, join the multicast group:

```
ip4_addr_t multicast_addr;
IP4_ADDR(&multicast_addr, 239, 0, 0, 1); // Your multicast IP
err_t result = igmp_joingroup_netif(&netif, &multicast_addr);
 if (result != ERR_OK) {
EnetAppUtils_print("IGMP join failed: %d\n", result);
}
```

**Note**

IGMP join tells the network stack to accept packets for this group. The ALE entry is still required for hardware filtering.

## 4.2 Custom Hardware Bring-Up Process

To bring up CPSW on a custom board, there are a list of example we can run on the device to verify if the CPSW and ethernet configurations are healthy.

### *4.2.1 Example 1: CPSW PHY Loopback*

The PHY loopback example tests the internal loopback mode of the external chip of the PHY. The test enables the loopback register (typically bit 14 of the PHY control register) of the PHY to route transmitted packets directly back to the receiver within the PHY silicon.

**Location:** source/networking/enet/core/examples/enet_loopback/enet_cpsw_loopback/

The example tests the path from the CPU to the PHY and back.

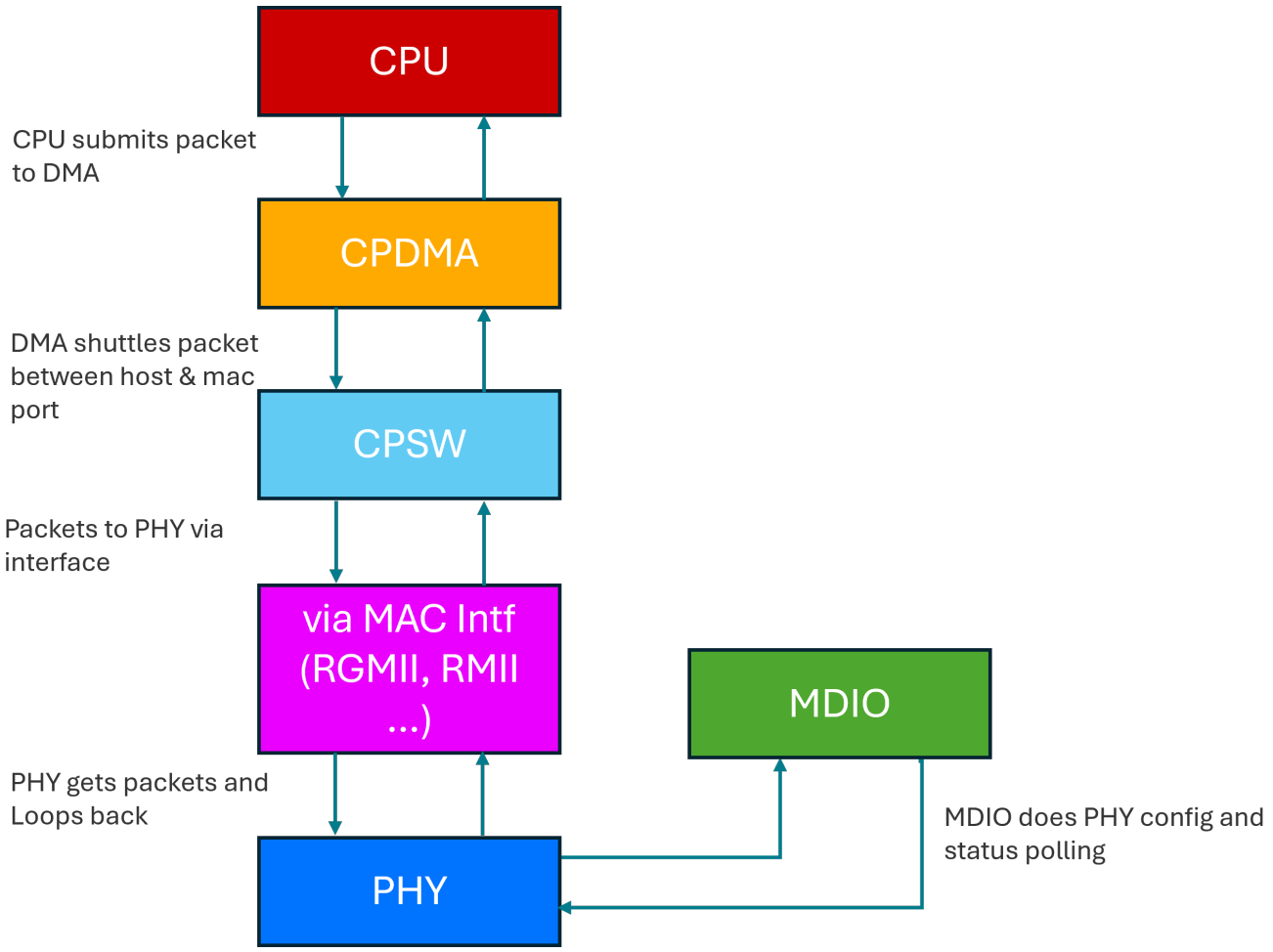


**Figure 4-15. CPSW PHY Loopback Flow**

Successful demonstration of the PHY loopback on the custom board means that the path to and from the CPU to the PHY is correct. If the examples fail, then there is some issue with one or more of the components. The following sections provide some sample failure cases.

#### 4.2.1.1 Failure: PHY Not Detected or MDIO Bus Not Alive

The driver cannot find the PHY chip on the MDIO bus.

**Table 4-1. PHY Failures Summary**

| Cause | How to Check | Fix |
|---|---|---|
| Wrong PHY address | Check the EEPROM or board label. | Update phyCfg.phyAddr in ti_board_config.c.<br>If multiple PHYs are present, each PHY must have a unique PHY address. |
| PHY not powered | Measure 3.3V or 1.8V on PHY power pins. | Verify the power supply, check for shorts. |
| MDIO pins not connected | Trace the MDC or MDIO pins on the PCB. | Verify the pin MUX in example.syscfg matches the schematic. |
| MDIO pin short or open | Continuity test with multimeter. | Fix the routing of the PCB, resolder if necessary. |
| PHY reset not released | Check the reset pin in the PHY datasheet. | Verify the reset circuit, check if the circuit is held low. |
| Link partner issue | Check if the other end is connected to the link partner. | Connect to a different link partner and check if the PHY alive log is coming. |
| RGMII clock delays incorrect | Check txDelayInPs and rxDelayInPs in ti_board_config.c. | Typical: 1500–2000ps; tune based on scope measurements |

**Table 4-1. PHY Failures Summary (continued)**

| Cause | How to Check | Fix |
|---|---|---|
| MDIO bus not functional | The board logs show *PHY not detected*. | Check the connectivity of the MDIO pin; verify the EEPROM settings<br>Run the PHY loopback example and check the *PHY debug* section |

#### 4.2.1.2 Failure: TX Packets Transmitted But RX Count = 0

**Table 4-2. RX Errors Summary**

| Cause | How to Check | Fix |
|---|---|---|
| Loopback not enabled | Read the PHY control register, check bit 14 | Verify the PHY driver supports loopback. Check the PHY datasheet to verify the loopback register setting. |
| PHY reset during test | Check if the link LED on the PHY turns off | Verify the power stability of the PHY. |
| DMA RX not configured | Check if hRxCh is valid | Verify the DMA channel open succeeded. Check if the RX interrupts are enabled. Check the statistics to see if packets are reaching the host port. |
| RX buffer pool empty | Check if rxFreeQ has buffers | Verify LargePoolPktCount and MediumPoolPktCount in SYSCFG is adequate. |
| PHY clock domain isolation | Check the board schematic for clock routing | Potentially requires a PCB rework if not isolated properly. |

### *4.2.2 Example 2: CPSW MAC Loopback Example*

The MAC loopback example enables internal loopback mode in the individual CPSW MAC hardware. Transmitted packets are routed back to the receiver within the SoC, completely bypassing the external PHY.

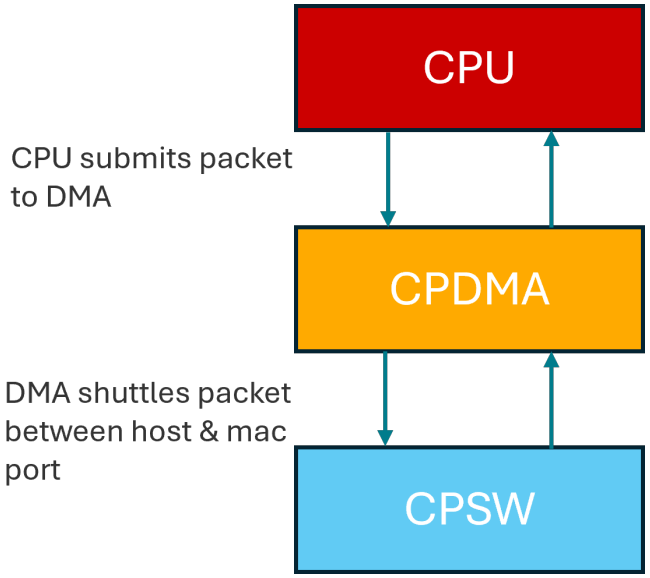


**Figure 4-16. CPSW MAC Loopback Example Flow**

The following sections provide examples of some possible failure cases.

#### 4.2.2.1 Failure: MAC Loopback Initialization Fails

The driver fails to enable MAC loopback mode.

**Table 4-3. MAC Loopback Error Summary**

| Cause | How to Check | Fix |
|---|---|---|
| Wrong MAC interface mode | Check mii.layerType and mii.sublayerType in ti_board_config.c. | MAC loopback only supports RMII or RGMII (not MII). |

**Table 4-3. MAC Loopback Error Summary (continued)**

| Cause | How to Check | Fix |
|---|---|---|
| MAC port not open | Check the return status of the EnetApp_open() function. | Verify the MAC port selection in configuration. |

#### 4.2.2.2 Failure: TX Packets Increase But RX = 0

MAC loopback mode is active, but packets are not being looped back to the receiver.

**Table 4-4. RX Error Summary**

| Cause | How to Check | Fix |
|---|---|---|
| ALE port state not FORWARD (need to check) | Read the ALE port state register. | Manually set the port to the FORWARD state. |
| RX DMA channel not opened | Check if the EnetDma_openRxFlow() function succeeded. | Verify the DMA configuration in ti_enet_dma_init.c. |
| RX buffer pool exhausted | Count the buffers in rxFreeQ. | Increase the packet pool sizes in SYSCFG. |
| MAC port not in loopback | Read the MAC control register directly. | Try resetting and re-enabling loopback. |

#### 4.2.2.3 Failure: Nonzero Error Counters

Packets are being transmitted and received, but many packets are corrupt or lost.

**Table 4-5. Nonzero Counters Error Summary**

| Cause | How to Check | Fix |
|---|---|---|
| DMA ring is too small | Check the descriptor count in configuration. | Increase the TX or RX ring sizes. |
| TX underrun | TX underrun stat > 0. | Increase LargePoolPktCount in SYSCFG. |
| RX overrun | RX overrun stat > 0. | Check the packet size; is the packet potentially exceeding MTU. |
| Memory bandwidth is limited | Check the system load. | Reduce the test packet rate or size. |

### *4.2.3 Example 3: Enet_Layer2_CPSW and Enet_Layer2_cpsw_switch*

The Layer 2 CPSW example is a simple Ethernet forwarding application that receives frames from an external network, processes these frames in the application, and forwards the frames back. The example tests the complete real-world data path with link negotiation, frame reception, and transmission.

**Location**: source/networking/enet/core/examples/enet_layer2_cpsw/, source/networking/enet/core/examples/enet_layer2_cpsw_switch/

This test is the first real network test. If this test fails, the problem is in the PHY configuration, external connectivity, and ALE. The example helps to test the following factors:

- PHY link negotiation
- Full PHY initialization
- MAC TX to the network
- MAC RX from the network
- ALE learning and forwarding
- Frame processing

#### 4.2.3.1 Hardware Setup

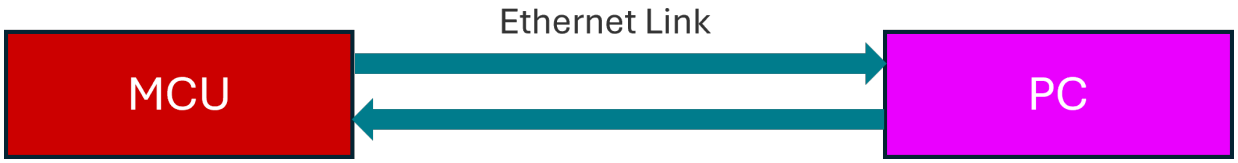


**Figure 4-17. Hardware Setup With MCU and External PC**

#### 4.2.3.2 Failure: Link Never Comes UP

The PHY is detected by the driver, but the external link negotiation fails or times out.

**Table 4-6. Link Up Failure Summary**

| Cause | How to Check | Fix |
|---|---|---|
| Ethernet cable is not connected or bad | Verify the physical connection of the cable and LED lights. | Replace the cable and test with known-good cable. |
| Host PC Ethernet is disabled or asleep | Check the Windows® Device Manager or Linux® Ethernet tool | Wake the PC, enable Ethernet, and disable power savings. |
| Switch or hub is not powered | Check the LED lights on the network equipment. | Power on the switch, hub, or router. |
| PHY is strapped for the wrong mode | Check the PHY datasheet strapping pins. | Verify the PCB strapping resistors match the schematic. If using a media convertor, verify that the PHY is connected to the SoC and the media convertor is not in the same mode (for example, if using a media convertor, verify that the PHY is connected to the SoC and the media convertor is not configured to assume the master or slave resolution. |

#### 4.2.3.3 Failure: Link is Up But No Frames Are Received or Transmitted

The link negotiation succeeded, but the packet is not received or transmitted.

**Table 4-7. Frame Transmission Error Summary**

| Cause | How to Check | Fix |
|---|---|---|
| The ALE port is not in the FORWARD state | Check the ALE port state (is meant to be FORWARD not LEARN/DISABLED). | Verify the ALE is configured to forward frames. |
| MAC RX is not enabled | Check if the EnetDma_openRxFlow() function succeeded. | Verify the DMA RX channel configuration in code. |
| The RX buffer pool is empty | Check rxFreeQ has buffers before test. | Increase LargePoolPktCount in SYSCFG. |
| The TX descriptors are not configured | Check if the EnetDma_openTxCh() function succeeded. | Verify the TX DMA channel is open. |
| The MAC address is not in the ALE | The frame potentially drops if the ALE did not learn the frame. | Send the broadcast frame to trigger ALE learning. |
| The frame size exceeds MTU | The RX controller potentially drops oversized frames. | Check that the frame is ≤ 1500 bytes (or configured as MTU). |

#### 4.2.3.4 Failure: RX and TX Counters Increase But Error Rates Are High

Frames are being transmitted and received, but many frames are corrupt or drop.

**Table 4-8. Error Rates Statistical Summary**

| Cause | How to check | Fix |
|---|---|---|
| RGMII clock and data timing issues | Check that macStats.rxCrcErrors > 0. | Tune txDelayInPs and rxDelayInPs in ti_board_config.c. Check the *Internal Delay* section. |
| The DMA descriptor ring is too small | Check the ring size in configuration. This triggers the top or bottom of FIFO drops. | Increase the TX and RX descriptor count in SYSCFG. |

### *4.2.4 Example 4: Enet_lwip_cpsw_example*

The enet_lwip_cpsw example is the primary Layer 3 bring-up test. The test runs the LwIP TCP/IP stack over the ENET driver, and exercises the full software path from Ethernet wire to the IP socket, including DHCP, ARP, ping, TCP echo, UDP echo, and iPerf throughput tests.

Use this example **after** the Layer 2 CPSW example passes. The test validates that the LwIP NETIF integration, packet buffer memory, and IP networking work correctly on your board

#### 4.2.4.1 Failure: Link Never Comes Up

The example starts but *MAC Port 1: link up* never appears on the console. The portLinkStatusChangeCb registered is never called.

**Table 4-9. MAC Port Link Up Issues Summary**

| Cause | How to Check | Fix |
|---|---|---|
| Cable is not connected or faulty | Inspect the link LED on the PHY—no LED means no physical link. | Replace the cable. Verify the RJ45 connector is seated. |
| Wrong PHY address in the board configuration | The console prints *PHY not detected* or the EnetApp_waitForPhyAlive() function hangs. | Correct the PHY address in ti_board_config.c to match the board schematic. |
| The MDIO bus is not functional | Call ENET_MDIO_IOCTL_IS_ALIVE—returns false. | Check that the MDC and MDIO pin MUX in the example.syscfg. Verify the pullups on the MDIO line. |
| The RGMII clock delay is misconfigured | The link negotiates but drops immediately. There are CRC errors on the MAC statistics. | Tune txDelayInPs and rxDelayInPs in ti_board_config.c (Typical: 1500–2000ps). |
| External PHY mode is enabled but the PHY is driver missing | ENET_SYSCFG_ENABLE_EXTPHY=1 but the extPhyMgmt/ driver is not in build. | Register corrects the PHY driver or disables ENET_SYSCFG_ENABLE_EXTPHY in SYSCFG. |

#### 4.2.4.2 Failure: Links Up But No IP Address Is Assigned

*MAC Port 1: link up* prints but *Board IP* never appears. The application waits indefinitely for DHCP.

**Table 4-10. IP Address Assignment Issues Summary**

| Cause | How to Check | Fix |
|---|---|---|
| No DHCP server on the network | The PC has no DHCP server running. The board sends DISCOVER with no response. | Run dnsmasq on the PC, or switch to static IP: set #define USE_DHCP 0 in lwipcfg.h:73. |
| The broadcast ALE entry is missing | hostStats.rxBcastFrames stays zero when the PC sends the DHCP offer. | Confirm the EnetApp_addMCastEntry() function at test_enet.c:147 is called before the main_loop() function. |
| The state of the ALE port is not FORWARD | macStats.aleDrop or macStats.aleBlockDrop are incrementing. | Set the state of the MAC port and host port ALE to CPSW_ALE_PORTSTATE_FORWARD. |
| The host PC firewall is blocking DHCP | Packet capture on the PC shows DISCOVER is sent but the OFFER never leaves the interface. | Disable the Windows® and Linux® firewalls on the test Ethernet interface. |

#### 4.2.4.3 Failure: Ping Fails Despite Link and IP Address

The board prints an IP address ping <board-ip> from the PC and gets no reply.

**Table 4-11. Ping Failure Summary**

| Cause | How to Check | Fix |
|---|---|---|
| The ALE is not forwarding unicast to the host port | macStats.rxGoodFrames > 0 but hostStats.rxGoodFrames = 0; check macStats.portMaskDrop. | Verify the ALE host port is in the FORWARD state. Check that the DA entry portmask includes bit 0. |
| There is DMA RX buffer starvation | hostStats.rxBottomOfFifoDrop > 0. | Increase numRxPkts in the EnetDma_openRxFlow() function using SYSCFG. |
| LwIP NETIF is not set to UP | netif_is_up(&netif) returns false in debugger. | Verify that the netif_set_up() function is called after the netif_add() function completes in the main_loop() function. |
| The ARP request dropped | macStats.rxGoodFrames = 0 on the ARP of the PC. There is no response from the board. | Check hostStats.aleUnknownUcast; confirm the broadcast ALE entry is present. |
| The PC and board are on different subnets | The IP addresses differ in subnet (for example, 192.168.1.x versus 10.0.0.x). | Assign the PC IP to same /24 subnet as the board; check the gateway settings. |

Furthermore, if other stacks are used, for example the LLDP, gPTP, MbedTLS, and so forth, the specific examples can be tested.

## 4.3 Debugging Packet Forwarding Issues (ALE and Statistics)

Hardware statistics counters provide the fastest path to isolating packet forwarding failures on the CPSW. For pinpointing the root cause, the CPSW STATS can be leveraged to find where the issues lie, whether in the ALE, MAC port, or host port configurations. STATS is a functional block in CPSW.

There are two ways to get the statistical values:

- Use the GEL scripts to obtain the CPSW statistics. In this method all nonzero CPSW statistics can be obtained.
- Using the expression window. This expression can be pasted in the *Watch Expressions* window to view the CPSW statistics.

**(CSL_Xge_cpswRegs *) (gEnetSoc_cpsw3g.enetPer.virtAddr)**

The key principle is that the statistics counter signature reveals the failure mode.

Table 4-12 lists some of the common issues that can be uncovered using CPSW statistics.

**Table 4-12. CPSW Statistical Summary**

| Issue Symptom | Primary Counters to Examine |
|---|---|
| The frame counts look correct but data is corrupted | rxCrcErrors, rxAlignCodeErrors, and rxOversizedFrames |
| Intermittent RX loss under high load | rxBottomOfFifoDrop, rxTopOfFifoDrop, and aleOverrunDrop |
| The application sends Ethernet packets but nothing appears on the wire | portMaskDrop, txSofOverrun, and MAC txGoodFrames |
| Packets arrive at the wire but the application receives nothing | portMaskDrop, aleDrop, and rxBottomOfFifoDrop |
| Traffic tagged as VLAN silently drops | aleVidIngressDrop and aleBlockDrop |
| Broadcast and multicast storm—throughput degrades | aleRateLimitDrop, aleUnknownBcast, and aleUnknownMcast |

### *4.3.1 CPSW Statistics Architecture*

In the AM26x (CPSW_3G) devices there are two independent statistics blocks; MAC port and host port statistics. MAC port statistics are available for each MAC port. Figure 4-18 shows the packet flow between the MAC port and the host ports. This figure presents how the port statistics are measured.

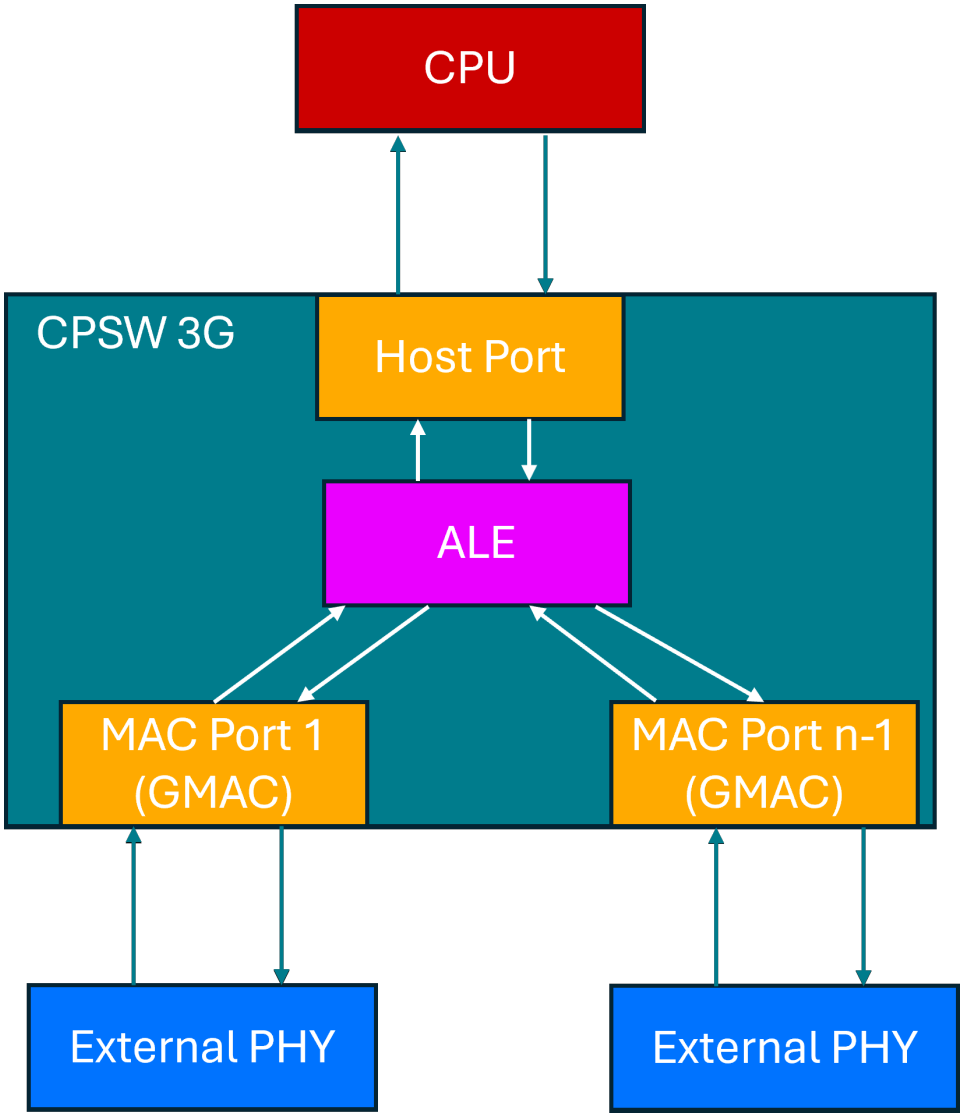


**Figure 4-18. CPSW Ports Diagram**

#### 4.3.1.1 What Each Block Measures

**MAC Port Statistics** – Counters at the GMAC boundary:

- Count frames as seen on the wire, before ALE processing.
- rxGoodFrames increments when a valid Ethernet frame is received from the PHY.
- Error counters (rxCrcErrors, rxJabberFrames, and so forth) reflect the quality of the physical layer.
- The ALE drop counters within the MAC block (aleDrop, portMaskDrop, and so forth) capture frames the ALE discards on ingress from the MAC Port 1.
- The TX counters reflect frames successfully placed on the wire.

**Host Port Statistics** – Counters at the CPPI and DMA boundary:

- Count frames as seen by the CPU DMA interface, after ALE processing.
- rxGoodFrames increments when a frame is delivered to the DMA RX ring.
- txGoodFrames increments when a frame from the DMA TX ring exits the host port.
- The ALE drop counters here capture frames the ALE discards on ingress from the host port.
- The FIFO drop counters (rxBottomOfFifoDrop and rxTopOfFifoDrop) reveal DMA back-pressure.

#### 4.3.1.2 Counter Reference Tables

#### 4.3.1.2.1 MAC Port – RX Counters

**Table 4-13. MAC Port RX Counters Summary**

| Counter | Struct Field | What It Means |
|---|---|---|
| Good frames received | rxGoodFrames | Valid frames received from the PHY, regardless of the outcome of the ALE |
| Broadcast frames | rxBcastFrames | Subset of rxGoodFrames – DA = FF:FF:FF:FF:FF:FF |
| Multicast frames | rxMcastFrames | Subset of rxGoodFrames – DA has multicast bit set |
| Pause frames | rxPauseFrames | IEEE 802.3x flow-control pause frames received |
| CRC errors | rxCrcErrors | Frame received with bad FCS – physical layer noise indicator |
| Alignment \| code errors | rxAlignCodeErrors | Non-integer number of bytes or 8b/10b symbol error |
| Oversized frames | rxOversizedFrames | Frame > configured max length (default 1518 B) with valid CRC |
| Jabber frames | rxJabberFrames | Frame > max length with bad CRC |
| Undersized frames | rxUndersizedFrames | Frame < 64 bytes with valid CRC |
| Fragments | rxFragments | Frame < 64 bytes with bad CRC |
| Received bytes | rxOctets | Byte count of rxGoodFrames only |

#### 4.3.1.2.2 MAC Port – TX Counters

**Table 4-14. MAC Port TX Counters Summary**

| Counter | Struct Field | What It Means |
|---|---|---|
| Good frames transmitted | txGoodFrames | Frames successfully placed on wire |
| Pause frames transmitted | txPauseFrames | Flow-control pause frames sent |
| Deferred frames | txDeferredFrames | Frames deferred due to carrier (half-duplex) |
| Collision frames | txCollisionFrames | Frames that experienced at least one collision |
| Single collision | txSingleCollFrames | Resolved with one retry |
| Multiple collisions | txMultipleCollFrames | Resolved with 2–15 retries |
| Excessive collisions | txExcessiveCollFrames | Discarded after 16 collisions – half-duplex mismatch indicator |
| Late collisions | txLateCollFrames | Collision after 64-byte slot – cable length or duplex mismatch |
| Carrier sense errors | txCarrierSenseErrors | Carrier lost mid-frame |
| Transmitted bytes | txOctets | Byte count of txGoodFrames only |

#### 4.3.1.2.3 MAC Port and Host Port – ALE and FIFO Drop Counters

Both MAC port statistics and host port statistics contain the same ALE and FIFO drop fields. When present in MAC port statistics, the fields describe drops of frames ingressing from MAC port n; when present in host port statistics, the fields describe drops of frames ingressing from the CPU DMA path.

**Table 4-15. ALE and FIFO Statistical Summary**

| Counter | Struct Field | Root Cause |
|---|---|---|
| ALE generic drop | aleDrop | Sum of all ALE policy drops (the below mentioned counters start from ALE overrun drops) |
| ALE overrun drop | aleOverrunDrop | ALE lookup FIFO overran – sustained burst exceeded ALE capacity |
| Port mask drop | portMaskDrop | ALE lookup resolved the DA but the result portmask excluded the destination port |
| RX bottom-of-FIFO drop | rxBottomOfFifoDrop | Ingress FIFO empty when frame arrived – immediate DMA starvation |
| RX top-of-FIFO drop | rxTopOfFifoDrop | Ingress FIFO full when frame arrived – sustained DMA back-pressure |
| ALE rate limit drop | aleRateLimitDrop | ALE policer classified frame as exceeding configured rate limit |
| ALE VID ingress drop | aleVidIngressDrop | VLAN ID of incoming frame not registered on the ingress port |
| ALE DA=SA drop | aleDAEqSADrop | Destination address equals source address – loopback detection |
| ALE block drop | aleBlockDrop | Ingress port state is BLOCKED – most common new-port misconfiguration |
| ALE secure drop | aleSecureDrop | SA of incoming frame bound to a different port in a secure VLAN |
| ALE auth drop | aleAuthDrop | Port requires 802.1x authentication; port is not yet authorized |

#### 4.3.1.2.4 Host Port – ALE Flood and Overrun Counters

These counters exist only in the host port block and reveal ALE flooding behavior.

**Table 4-16. ALE Flood and Overrun Statistical Summary**

| Counter | Struct Field | What It Means |
|---|---|---|
| Unknown unicast count | aleUnknownUcast | Frames with a DA not found in ALE table – flooded to all ports |
| Unknown multicast count | aleUnknownMcast | Multicast DA not found in ALE table – flooded |
| Unknown broadcast count | aleUnknownBcast | Broadcast frames received – always flooded |
| TX SOF overrun | txSofOverrun | Start-of-frame DMA descriptor overrun on host port TX path |
| TX MOF overrun | txMofOverrun | Mid-of-frame DMA descriptor overrun on host port TX path |

#### 4.3.1.2.5 MAC Port RX Issues

Table 4-17 highlights some issue that can be uncovered by the counter in the MAC port RX path. The counter name, debug method, and fix is mentioned in the table.

**Table 4-17. MAC Port RX Statistical Summary**

| Scenario | Nonzero Counter | Root Cause | Debug Method | Fix \| Reference |
|---|---|---|---|---|
| No frames from the PHY; link LED active | rxGoodFrames = 0 | The PHY is not linked, cable fault, or MDIO misconfiguration. | Read ENET_MDIO_IOCTL_IS_ALIVE. Check the PHY address. | PHY or Link Debug |
| Frames arrive but data is corrupted | rxCrcErrors | Cable noise, EMI, bad connector, or far-end duplex mismatch. | Replace the cable. Read negotiated duplex using ENET_PHY_IOCTL_GET_LINK_STATUS. | Force full-duplex using userLinkCaps in EnetPhy_Cfg |
| Signal errors, unreliable link | rxAlignCodeErrors | 8b/10b symbol error or non-octet-aligned frame – signal integrity. | Shorten the cable. Check for EMI sources. | Replace cable |
| All RX frames are dropped, CPU receives nothing | aleBlockDrop | The state of the ALE port 1 is BLOCKED or DISABLED – not set to FORWARD after initialization. | Issue CPSW_ALE_IOCTL_GET_PORT_STATE for port 1. | Set to FORWARD: CPSW_ALE_IOCTL_SET_PORT_STATE |
| Frames tagged as VLAN are silently dropped | aleVidIngressDrop | The 802.1Q VID in the frame is not registered in the ALE VLAN table for port 1. | Dump the ALE table: CPSW_ALE_IOCTL_DUMP_TABLE. Check for VID entry with port 1. | Add entry: CPSW_ALE_IOCTL_ADD_VLAN |
| Frames arrive, none reach the host port; no aleDrop | portMaskDrop | The ALE DA lookup succeeded but host port 0 is excluded from the egress portmask. | Inspect the DA entry portmask using CPSW_ALE_IOCTL_DUMP_TABLE. | Re-add DA entry with portmask including bit 0: CPSW_ALE_IOCTL_ADD_UCAST |
| Known source MAC is dropped; other traffic is OK | aleSecureDrop | The SA registered in the ALE with a secure flag on a different port. | Find the SA entry in the ALE table dump. Check the portNum and secure fields. | Delete stale entry: CPSW_ALE_IOCTL_DEL_UCAST; re-add without secure flag |
| Burst loss under high traffic | aleOverrunDrop | The burst rate exceeded the look-up FIFO capacity of the ALE. | Monitor aleOverrunDrop delta over 1s under load. | Configure ALE rate limiter: CPSW_ALE_IOCTL_SET_PORT_BCAST_MCAST_LIMIT |

#### 4.3.1.2.6 MAC Port TX Issues

Table 4-18 highlights some issue that can be uncovered by the counter in the MAC port RX path. Frames as placed on the wire by the GMAC. TX error counters reflect half-duplex collision behavior and PHY-level faults. txGoodFrames only increments when a frame exits the MAC without error.

**Table 4-18. MAC Port TX Statistical Summary**

| Scenario | Nonzero Counter | Root Cause | Debug Method | Fix \| Reference |
|---|---|---|---|---|
| macStats.txGoodFrames == 0;<br>host TX is non-zero | hostPort.aleDrop or hostPort.portMaskDrop | The ALE drops frames that originate in the CPU before the frames reach MAC port 1. | Read the host port statistics.<br>Check aleBlockDrop and portMaskDrop. | Fix the state of the host port ALE or the DA entry portmask (see *Host Port – ALE Flood and Overrun Counters*). |
| TX frames disappear;<br>no ALE drops | txExcessiveCollFrames | 16+ collision retries – severe duplex mismatch with the link partner | Read negotiated duplex using ENET_PHY_IOCTL_GET_LINK_STATUS. | Force full-duplex on both ends using userLinkCaps. |
| TX unreliable;<br>occasional frame loss | txLateCollFrames | Collision after 512-bit window – the cable is too long (> 100m) or there is a duplex mismatch. | Check the cable length.<br>Read link mode. | Shorten the cable. Force full-duplex. |
| TX stalls intermittently | txCarrierSenseErrors | The PHY drops the carrier mid-frame – marginal link or intermittent disconnect. | Monitor delta during traffic.<br>Check the state of the PHY link. | Replace the cable. Verify the power and reset sequence of the PHY. |
| TX throughput lower than expected | rxPauseFrames (rising) | Remote host sending IEEE 802.3x PAUSE frames – flow control active. | Check rxPauseFrames delta. | Increase the host-side RX buffer depth or disable flow control. |
| Deferred frames accumulating | txDeferredFrames | Half-duplex: channel busy, frame deferred. | Check if half-duplex was forced intentionally. | Switch to full-duplex.<br>Check automatic negotiation. |

#### *4.3.1.2.7 Host Port RX Issues*

Frames arriving at the CPU DMA interface from the CPSW. rxGoodFrames here are meant to match macPort.rxGoodFrames minus any ALE drops. Any discrepancy is frames lost inside the switch on the path from the ALE to DMA.

**Table 4-19. Host Port RX Statistical Summary**

| Scenario | Nonzero Counter | Root Cause | Debug Method | Fix \| Reference |
|---|---|---|---|---|
| hostPort.rxGoodFrames is much less than macPort.rxGoodFrames; no ALE drops on the MAC port | hostPort.rxBottomOfFifoDrop | The DMA RX ring had no free entries. The application is not recycling buffers fast enough. | Compare the wire RX rate versus the application packet counter.<br>Check numRxPkts at initialization. | Increase numRxPkts in the EnetDma_openRxFlow() function. Raise the RX task priority. |
| RX loss only under a sustained high rate | hostPort.rxTopOfFifoDrop | The DMA RX FIFO filled sustained back-pressure from slow processing. | Monitor counter delta over 1s during burst. | Increase the ring depth. Move RX processing to a higher-priority task. |
| The ALE drops visible in host port statistics | hostPort.aleBlockDrop | The state of the host port (port 0) ALE is not FORWARD. | Issue CPSW_ALE_IOCTL_GET_PORT_STATE for port 0. | Set port 0 to FORWARD: CPSW_ALE_IOCTL_SET_PORT_STATE. |
| The ALE drops at the host port; tagged traffic | hostPort.aleVidIngressDrop | The VLAN on the incoming frame is not registered for the host port in the ALE. | Dump the ALE table – check the VLAN entry includes port 0. | Add or update the VLAN entry with the host port in the member list. |
| The CPU is starved by broadcast or multicast | hostPort.aleUnknownBcast or aleUnknownMcast very high | Storm traffic flooded to host port; no ALE rate limiter | Check counter delta over 1s; compare the *Host Port Rx Counter* with the normal traffic baseline. | Configure the rate limiter: CPSW_ALE_IOCTL_SET_PORT_BCAST_MCAST_LIMIT. |
| Policer drops are accumulating | hostPort.aleRateLimitDrop | The policer of the ALE is active – the rate limit is potentially too aggressive. | Verify the configured rate limit value. Check if the traffic is legitimate. | Increase the limit or register multicast groups explicitly with CPSW_ALE_IOCTL_ADD_MCAST. |

#### 4.3.1.2.8 Host Port TX Issues

Frames submitted by the CPU DMA to the CPSW. txGoodFrames increments when a frame exits the host port into the ALE. The ALE then forwards the frame toward MAC port 1 where macPort.txGoodFrames is meant to match.

**Table 4-20. Host Port TX Statistical Summary**

| Scenario | Nonzero Counter | Root Cause | Debug Method | Fix \| Reference |
|---|---|---|---|---|
| hostPort.txGoodFrames = 0; application confirms frames were sent | N/A | The DMA TX ring is not draining; frames are stuck in ring. | Verify the return code of the EnetDma_submitTxPktQ() function. Check the TX confirmation callback is called. | DMA or Packet Flow Debug |
| hostPort.txGoodFrames non-zero; macPort.txGoodFrames = 0; no collision | hostPort.aleBlockDrop | The state of the host port (port 0) ALE is not FORWARD. | Issue CPSW_ALE_IOCTL_GET_PORT_STATE for port 0. | Set port 0 to FORWARD: CPSW_ALE_IOCTL_SET_PORT_STATE. |
| Host TX OK; MAC TX zero; no aleBlockDrop | hostPort.portMaskDrop | The ALE DA lookup excluded MAC port 1 from the egress portmask. | Inspect the DA entry portmask in the ALE table dump. | Re-add the DA entry pointing to MAC port 1: CPSW_ALE_IOCTL_ADD_UCAST. |
| Host TX OK; tagged frames not reaching wire | hostPort.aleVidIngressDrop | The VLAN tag on the TX frame is not registered for the host port in the ALE. | Dump the ALE VLAN entries. Check host port membership. | Add the VLAN entry with the host port in the member list. |
| TX count correct but lower than submitted | hostPort.txSofOverrun or txMofOverrun | DMA TX descriptor ring overrun; the CPU is submitting faster than switch drains. | Compare the TX submit rate versus macPort.txGoodFrames delta. | Increase numTxPkts in the EnetDma_openTxCh() function. Check the TX confirmation task priority. |

## 4.4 Custom Board Enablement in SYSCFG

This section explains how to enable Ethernet features on a custom board. The examples provided in MCU+SDK are tested on TI EVMs like the LaunchPad™ development kit, control-cards, and systems on module devices. Such devices have EEPROMs and IO expanders configured to route Ethernet signals from the SoC to the PHY and to store data like the Ethernet MAC address. These examples cannot be directly ported to custom hardware because custom hardware generally lacks EEROMs and IO expanders.

In scenarios where users want to run SDK examples on custom hardware, the user must adapt the contents of the ti_board_config.c file based on the custom hardware. This file has mainly two board level dependencies:

- **EEPROM Dependency**: An EEPROM dependency is present because the MAC address of the device is stored in EEPROM.
- **IO Expander**: The IO expander dependency is present because EVMs use this to route signals from the SoC to the PHY.

Please use the following steps for enabling Ethernet support on custom hardware.

1. Expand the board configuration section.
   - The *Custom Board* option is now visible. The *Custom Board* option is disabled by default. This option must be enabled.
2. Select the check box to enable the *Custom Board* option.
   - The ti_board_config.c and ti_board_config.h files generate when the *Custom Board* check box is enabled.
3. Manually reconstruct the .c file based on the custom board design.
   - The file is responsible for the following:
     - PHY driver function registration
     - PHY driver configuration
     - MAC port configuration
     - Getting MAC address list

**Note**

An example demonstrating the custom board integration is available in SDK. The file is available at source\networking\enet\core\examples\enet_layer2_multi_channel\am263x-lp\r5fss0-0_freertos\enet_custom_board_config.c.

For users, the same file can be chosen as a reference while developing the board_config.c file.

The following is an overview of the contents of the file:

- **const EnetPhy_DrvInfoTbl gEnetPhyDrvTbl**: This is a table of ENET PHY drivers supported on the board. Refer to the *ENET Custom PHY Integration Guide* for details on how to populate this table. This guide can be referenced from the MCU+SDK documentation.
  - If the Ethernet PHY used in the custom board is supported in MCU+SDK out-of-box, include the appropriate header files and populate the table.
  - Pay attention to the data structures mentioned in Figure 4-19 and Figure 4-20 and populate the structures accordingly.

```
/* PHY drivers */
extern Phy_DrvObj_t gEnetPhyDrvDp83869;
extern Phy_DrvObj_t gEnetPhyDrvGeneric;

/*! \brief All the registered PHY specific drivers. */
static const EthPhyDrv_If gEnetPhyDrvs[] =
{
    &gEnetPhyDrvDp83869,    /* DP83869 */
    &gEnetPhyDrvGeneric,    /* Generic PHY - must be last */
};

const EnetPhy_DrvInfoTbl gEnetPhyDrvTbl =
{
    .numHandles = ENET_ARRAYSIZE(gEnetPhyDrvs),
    .hPhyDrvList = gEnetPhyDrvs,
};
```

**Figure 4-19. ENET PHY Driver Table**

```
/*!
 * \brief Common Processor Board (CPB) board's DP83869 PHY configuration.
 */
static const Dp83869_Cfg gEnetCpbBoard_ConfigEnetEthphy1PhyCfg =
{
    .txClkShiftEn         = true,
    .rxClkShiftEn         = true,
    .txDelayInPs          = 2000U,   /* Value in pecosec. Refer to DLL_RX_DELAY_CTRL_SL field in ANA_RGMII_DLL_CTRL register of DP8
    .rxDelayInPs          = 2000U,   /* Value in pecosec. Refer to DLL_TX_DELAY_CTRL_SL field in ANA_RGMII_DLL_CTRL register of DP8
    .txFifoDepth          = 4U,
    .impedanceInMilliOhms = 35000,  /* 35 ohms */
    .idleCntThresh        = 4U,     /* Improves short cable performance */
    .gpio0Mode            = DP83869_GPIO0_LED_2,
    .gpio1Mode            = DP83869_GPIO1_COL, /* Unused */
    .ledMode              =
    {
        DP83869_LED_LINKED_100BTX,         /* Unused */
        DP83869_LED_RXTXACT,
        DP83869_LED_LINKED,
        DP83869_LED_LINKED_1000BT,
    },
};
```

**Figure 4-20. ENET PHY Driver Configuration for Custom Board**

- **EnetBoard_setupPorts()**: This function sets up any board-level muxes and configures any SoC-level RGMII internal delay or RMII configuration for the specific port. In some AM26x EVMs, like the AM263P CC, there are some board-level muxes that are used to route MDIO and RGMII signals from the SoC to the PHY using an IO expander. This file has the configuration related to that set-up. If the custom board lacks these IO expanders, then the EnetBoard_setMacPort2IOExpanderCfg() function must be removed. If the IO expander is used, then the corresponding logic must be implemented.
- **EnetBoard_getPhyCfg()**: This function returns the ETHPHY specific configuration for a given port including any extended PHY configuration.
- **EnetBoard_getMacAddrList()**: This function returns the board-specific Ethernet MAC addresses that are available. The MAC address can be statically configured in code or can be stored in some non-volatile memory. In any case, the corresponding logic must be implemented in this function.
- **EnetBoard_getId()**: This function returns the board ID. This function is not used anywhere outside this file, so the board ID returned depends on the implementation of the EnetBoard_setupPorts() function.
- **EnetBoard_getPhyCfg()**: This function returns the Ethernet PHY specific configuration for a given port including any extended PHY configuration.
  - The function basically returns the configurations in the structure shown in Figure 4-21. Please pay attention and configure this structure correctly.

```
/*
 * am263px-cc board configuration.
 *
 * RMII/RGMII PHY connected to am263px-cc CPSW_3G MAC port.
 */
static const EnetBoard_PortCfg gEnetCpbBoard_am263px_cc_EthPort[] =
{
    {    /* "CPSW3G" */
        .enetType = ENET_CPSW_3G,
        .instId   = 0U,
        .macPort  = ENET_MAC_PORT_2,
        .mii      = {ENET_MAC_LAYER_GMII, ENET_MAC_SUBLAYER_REDUCED},
        .phyCfg   =
        {
            .phyAddr         = 0,
            .isStrapped      = false,
            .skipExtendedCfg = false,
            .extendedCfg     = &gEnetCpbBoard_ConfigEnetEthphy1PhyCfg,
            .extendedCfgSize = sizeof(gEnetCpbBoard_ConfigEnetEthphy1PhyCfg)
        },
        .flags    = 0U,
    },
};
```

**Figure 4-21. ENET PHY Driver Configuration**

To summarize, to run SDK examples on a custom board, enable the *Custom Board* option in SYSCFG and create a replica of the ti_board_config.c file based on the custom board hardware design.

## 4.5 LwIP Debug Guide

The LwIP stack is an open-source TCP/IP suite specifically designed for embedded systems and microcontrollers. The AM26x family of devices supports a LwIP stack out-of-box in MCU_PLUS_SDK. The stack is supported over both FreeRTOS and NoRTOS, with multiple TCP, UDP, HTTP applications supported in the SDK. The MCU_PLUS_SDK also has MbedTLS integrated to provide TLS security over LwIP examples.

### *4.5.1 LwIP Stack Configuration*

The TI port of the LwIP stack supports a configuration that is tailored to provide a balance between performance and memory footprint. Refer to the *lwipopts.h* file in the *source/networking/lwip/lwip-config/{device}/enet/* path to learn more about TI's support for the out-of-box LwIP feature. If the out-of-box configuration does not fulfill the application requirements, the file can be modified and the LwIP stack can be rebuilt.

To rebuild the stack, use the *makefile* in the SDK. Provide the appropriate build profile (release or debug) and the operating system details to trigger the correct *makefile*. Sample *gmake* commands are shown below:

```
gmake -sj -f makefile.am261x lwipif-cpsw-nortos_r5f.ti-arm-clang PROFILE=debug
 gmake -sj -f makefile.am261x lwip-contrib-freertos_r5f.ti-arm-clang PROFILE=debug
 gmake -sj -f makefile.am261x lwip-freertos_r5f.ti-arm-clang PROFILE=debug
 gmake -sj -f makefile.am261x lwip-nortos_r5f.ti-arm-clang PROFILE=debug
 gmake -sj -f makefile.am261x lwip-contrib-nortos_r5f.ti-arm-clang PROFILE=debug
 gmake -sj -f makefile.am261x lwipif-cpsw-freertos_r5f.ti-arm-clang PROFILE=debug
```

After rebuilding the library with changes, rebuild the application to link the updated libraries.

### 4.5.2 lwip_stats

The LwIP stack also provides statistics for the packets received by the stack. The statistics can be viewed by entering the lwip_stats expression in the CCS.

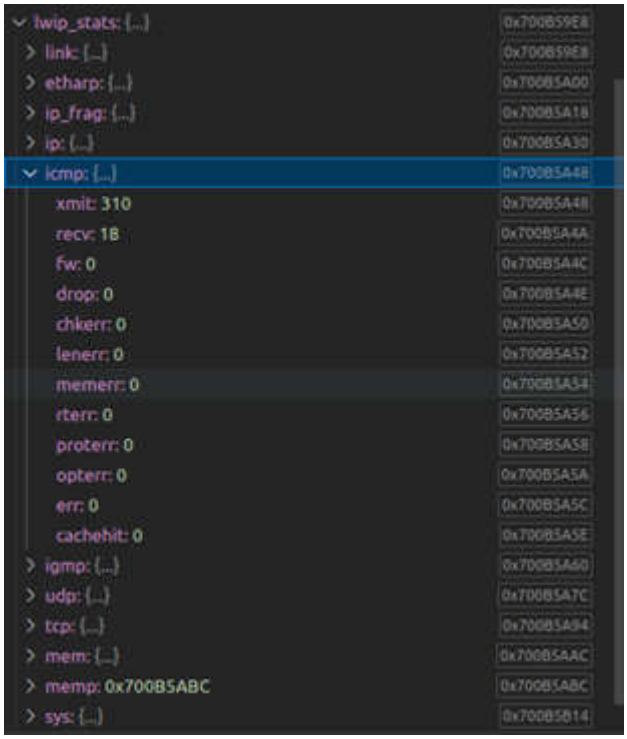


**Figure 4-22. LwIP Statistics Screen Shot**

Figure 4-22 provides crucial information about the packets generated by the stack and the packets received by the stack for application (for example, UDP packets or HTTP packets).

The LwIP statistics can be used in conjunction with CPSW_STATS to understand the behavior of the traffic. For example, if the developer suspects that the application cannot receive packets from an external source, but is able to generate and send packets, the debug flow is to:

**Transmission**:

- Generate a packet through the application and send the packet out through any MAC port (for example, MAC port 1)
- If the application has LwIP integrated, check the LwIP statistics to see the *xmit* count increment. For example, if the application sends x UDP packet, the *xmit* statistic is meant to increment accordingly.
- Check the MAC port and host port statistics of the CPSW. Since the host port generates the packet and sends the packet to the MAC port, and the MAC port sends the packet out of the device to the external source, the count of *RxGoodFrames* increments for the host port and the count of the *TxGoodFrames* increments for the MAC port.

**Reception**:

- The packet generated externally is received in the MCU on the MAC port and the *RxGoodFrames* count increments.
- The host port statistics show an increment in the *TxGoodFrames* count.
- The lwip_stats shows an increment in the *recv* parameter.

If the statistics are not incrementing as expected, this result points to packet drop behavior. At the CPSW level, the packet is either dropped when the packet is malformed, has incorrect headers, or if the ALE is configured to not forward the packet to other MAC port or host port (refer to *CPSW Statistics* section for more details). If the

CPSW receives the packet and the LwIP statistics do not show the packet, this result indicates an issue on the side of the lwIP. Check the memory pools and failures at the LwIP level.

If the packets do not show up at the CPSW statistics or LwIP statistics, revisit the *Hardware Debug* section and make sure the Ethernet hardware, links, PHY, and setup is flawless.

When debugging a LwIP based on a TCP or UDP application on the DUT, verify the link partner is properly running the TCP or UDP client or server. Start the server application before running the client application. Use packet sniffing tools, such as Wireshark®, to see if the packet flow is as expected. Since the LwIP is an open source software stack, not maintained by TI, direct questions specific to the functions and support of the LwIP stack to the LwIP support forum.

# 5 Conclusion

This user manual discussed various software and hardware debug pointers for the CPSW3G switch used on Texas Instruments' microcontrollers and microprocessors. If this user guide does not address or resolve your issue, please reach out to TI experts over the TI E2E™ forums. In the E2E thread, please attach the steps that were followed from this debug guide and results for the sections. This information eases the support process.

Checklist for requesting Ethernet support when posting an inquiry on the TI E2E forum.

- What is the TI SDK version in use?
- What is the TI processor in use?
- Is the device a TI EVM or a custom board?
- What is the test topology?
- If using a custom board, please mention if the schematics have been reviewed by TI.
- Attach the console logs of the application.
- Indicate that this user guide was followed and in what sections were the results as expected.

# 6 References


1. Texas Instruments, *MCU-PLUS-SDK-AM261X: AM261x Software Development Kit (SDK) for Sitara™ Microcontrollers*, tool.
2. Texas Instruments, *MCU-PLUS-SDK-AM263X: AM263x Software Development Kit (SDK) for Sitara™ Microcontrollers*, tool.
3. Texas Instruments, AM26x Academy, learning modules.
4. Texas Instruments, *Designing a Deterministic Scalable Ethernet Backbone for Robotics*, application note.
5. Texas Instruments, AM26x FAQ, frequently asked question webpage.
6. Texas Instruments, *Running Layer-2 Ethernet Examples on AM26x Microcontrollers AM26x* video series.

# IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.



Last updated 10/2025