## Enet Integration Guide

# Introduction

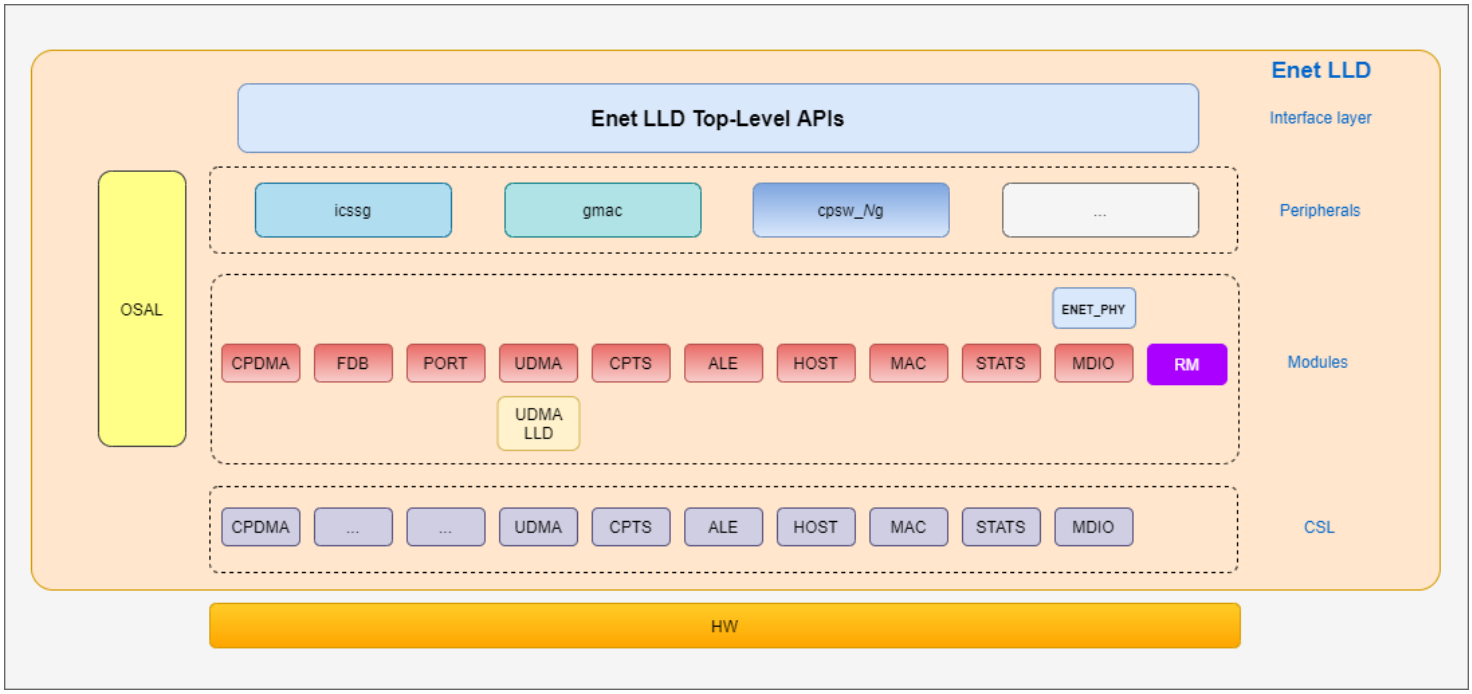


Enet LLD is an unified Ethernet driver that support Ethernet peripherals found in TI SoCs, such as CPSW and ICSSG. Please refer to the SDK release notes to find out what peripherals are currently supported.

The diagram below shows the overall software architecture of the Enet low-level driver. A top-level driver layer provides the interface that the applications can use to configure the switch and to send/receive Ethernet frames.

For instance, the CPSW support in the Enet driver consists of several software submodules that mirror those of the CPSW hardware, like DMA, ALE, MAC port, host port, MDIO, etc. Additionally, the Enet driver also includes PHY driver support as well as a resource manager to administrate the CPSW resources.

Enet LLD relies on other PDK drivers like UDMA for data transfer to/from the Ethernet peripheral's host port to the other processing cores inside the TI SoC devices. For the lower level access to the hardware registers, Enet LLD relies on the Chip Support Library (CSL).




**Enet LLD Software Architecture Block Diagram**

Back To Top

# Examples

Enet LLD comes with a set of examples demonstrating the usage of driver APIs. The examples are:

- `enet_loopback`: Internal (MAC port) or external loopback test.
- `enet_nimu_example`: TCP/IP stack integration using TI NDK package.
- `enet_multiport`: ICSSG and CPSW multiport test app.

## Enet loopback

This example exercises the MAC loopback functionality of the hardware. The example is developed and tested on both bare metal and TI RTOS code base. The CPSW9G hardware is opened with default initialization parameters and the MAC loopback is enabled.

A Tx channel and a Rx flow are opened to enable data transfers. Packets are transmitted from the Switch R5F (Main R5F0_0) to the host port using the Tx channel. These packets are routed back to the host port by the switch hardware as the internal loopback feature is enabled. These packets are then transmitted to the Switch R5F by the Rx flow and the application is notified.

The Tx and Rx functions in the example are set to transmit and receive 10000 packets. After reaching the count of 10000, the application closes the Tx channel, Rx flow, CPSW and restarts the application for a configurable number of times. Restarting the loopback test application ensures that there aren't any memory leaks, and the hardware is closed properly and can be reopened any time.

**Note:** Currently, this example application is supported only in peripherals of the CPSW family, such as CPSW2G, CPSW5G and CPSW9G.

### Enet Multiport

The multiport example is dedicated to demonstrate simultaneous usage of Enet ICSSG peripherals operating in Dual-MAC mode. A total of up to 6 ICSSG MAC ports can be tested simultaneously with this example application.

This example application also supports CPSW2G peripheral present in AM65xx devices.

This example has two components:

- Target-side application running on a Cortex R5F core.
  - One TX channel and one RX flow are opened for each available MAC port.
  - Application receives the packet, copies the payload into a new packet which is then sent back.
  - The application has a menu to enable/disable features, such as packet timestamping. This menu along with application logs are implemented via UART.
- Host-side application.
  - This is a Linux command-line application meant to simplify testing of the multiport example, it uses a raw socket to send and receive packets.
  - Alternatively, other application could be used as well: packETH tool can be used to generate and send packets, Wireshark can be used to receive and verify packet contents.

The data path enabled in this example is as follows:

1. Host side (PC) application sends a packet to an AM65xx IDK MAC port.
   - By default, a broadcast packet is sent but the user can specify any other MAC address.
2. Target side application receives the packet, updates the MAC addresses in the Layer-2 header and sends the packet back.
3. Host side application receives the packet and checks if the payload matched what it had sent in step 1.
4. Packet timestamping can be enabled/disabled at any time via target-side application menu. Enabling timestamping implies that both RX and TX timestamps will be enabled.

**Note:** Currently, this example is supported only in TI AM65xx Industrial Development Kit (IDK) which provides 6 ICSSG MAC ports and 1 CPSW2G MAC port.

### Enet NIMU Example

The Network Developer's Kit (NDK) is a platform for development and demonstration of network enabled applications on TI embedded processors. The NDK stack serves as a rapid prototyping platform for the development of network and packet processing applications. It can be used to add network connectivity to existing applications for communications, configuration, and control. Using the components provided in the NDK, developers can quickly move from development concepts to working implementations attached to the network.

Network Interface Management Unit (NIMU) acts as an intermediate layer between Enet LLD and the TI-NDK (NDK is TI's TCP/IP stack with http server, telnet support, etc).

The NIMU example uses the NIMU layer present in the Enet LLD and gets an IP address using the NDK stack and opens transmit and receive ports.

The send and receive functionalities of the Ethernet peripheral can be tested using the tools provided in NDK Winapps as follows:

- Send test:

```
cd ndk_<version>/packages/ti/ndk/winapps/
./send.x86U <IP address>
```

- Receive test:

```
cd ndk_<version>/packages/ti/ndk/winapps/
./recv.x86U <IP address>
```

Where `IP address` refers to the address of the processing core running the example, i.e. R5F, A72 or A53.

**Back To Top**

# Getting Familiar with Enet LLD APIs

The Enet LLD APIs can be broadly divided into two categories: control and data path. The control APIs can be used to configure all Ethernet hardware submodules like FDB, MAC port, host port, MDIO, statistics, as well as PHY drivers and resource management. The data path APIs are exclusive for the DMA-based data transfers between the TI SoC processing cores and the Ethernet peripheral.

The main APIs of the Enet LLD are the following:

- **Enet_open()**
- **Enet_close()**
- **Enet_ioctl()**
- **Enet_periodicTick()**

- **EnetDma_openRxCh()**
- **EnetDma_closeRxCh()**
- **EnetDma_openTxCh()**
- **EnetDma_closeTxCh()**
- **EnetDma_retrieveRxPktQ()**
- **EnetDma_submitRxPktQ()**
- **EnetDma_retrieveTxPktQ()**
- **EnetDma_submitTxPktQ()**

It's worth noting that the control path APIs are mainly IOCTL-based, and the data path APIs are direct functions in order to avoid any additional overhead associated with IOCTL calls as DMA data operations occur highly frequently.

### IOCTL Interface

IOCTLs are system calls that take an argument specifying the command code and can take none or additional parameters via **Enet_IoctlPrms** argument. IOCTL are used by all Enet submodules except for DMA.

The **Enet_IoctlPrms** parameter structure consists of input and output argument pointers and their corresponding size. The following helper macros are provided to help construct the IOCTL params:

- **ENET_IOCTL_SET_NO_ARGS(prms)**. Used for IOCTL commands that take no parameters.
- **ENET_IOCTL_SET_IN_ARGS(prms, in)**. Used for IOCTL commands that take input parameters but don't output any parameter.
- **ENET_IOCTL_SET_OUT_ARGS(prms, out)**. Used for IOCTL commands that don't take input parameters but return output parameters.
- **ENET_IOCTL_SET_INOUT_ARGS(prms, in, out)**. Used for IOCTL commands that take input parameters and also return output parameters.

where `prms` in a pointer to **Enet_IoctlPrms** variable, `in` is the pointer to IOCTL input argument and `out` is the pointer to IOCTL output argument.

It's recommended that the application doesn't set the **Enet_IoctlPrms** members individually, but only through the helper macros listed above.

Please refer to the individual IOCTL command to find out if it requires input and/or output parameters.

**Back To Top**

## Integrating Enet LLD into User's Application

Developers who wish to add network connectivity to the applications running on TI SoCs, will have to integrate Enet LLD by following the below sequence:

1. **Init Sequence**. One-time initialization before using any Enet LLD APIs.
2. **Peripheral Open Sequence**. Opens an Ethernet peripheral, it's called for each peripheral that the application intends to use, i.e. ICSSG0 and CPSW.
3. **Port Open Sequence**. Opens a MAC port. For peripherals that provide multiple MAC port (i.e. CPSW9G that has 8 MAC ports), this sequence must be followed for each MAC port. This section also covers MAC-PHY and MAC-MAC links.
4. **Packet Send/Receive Sequence**. DMA TX and RX channel open sequences as well as functions to use to submit and retrieve queue of packets to/from the driver.
5. **Packet Timestamping Sequence**. TX and RX packet timestamping sequence to enable timestamping and retrieval mechanism.
6. **Port Close Sequence**. Closes a MAC port. Application has to close all MAC ports which were opened in step 3 when no longer needs those ports.
7. **Peripheral Close Sequence**. Closes an Ethernet peripheral when the application no longer needs it. This sequence must be followed for each peripheral that was opened in step 2.
8. **Deinit Sequence**. One-time deinitialization once application is done using Enet LLD.

Each of these sequences will be covered in detail in the following sections.

### Init Sequence

This is a one-time initialization where the application sets the OSAL and utils functions that Enet LLD will use throughout its lifecycle.

At this stage Enet LLD also initializes its SoC layer which contains data about the Ethernet hardware available in the TI device.

The application should follow the next steps:

1. (Optional) **Enet_initOsalCfg()** to initialize the OSAL configuration (see **EnetOsal_Cfg**) with a default implementation. The default implementation of Enet OSAL interface is based on PDK OSAL layer. Typically, applications can directly use the default Enet OSAL implementation, unless Enet is being used on another OS (i.e. QNX) of if OS is not supported by PDK OSAL layer.
2. (Optional) **Enet_initUtilsCfg()** to initialize utils configuration (see **EnetUtils_Cfg**) with a default implementation. The default implementation of Enet Utils interface is UART-based logging and one-to-one address translations (physical-to-virtual and virtual-to-physical). Applications may want to pass their own utils' print function if UART is not available. Similarly, if a MMU is present and used, applications may need to pass their own address translation functions corresponding to the MMU configuration.

   ```
   utilsCfg.print = UART_printf;
   utilsCfg.physToVirt = &myPhysToVirtFxn;
   utilsCfg.virtToPhys = &myVirtToPhysFxn;
   ```

3. **Enet_init()** to pass the OSAL and utils configurations setup in the previous two steps. The application can pass NULL to either the OSAL config or utils config if it intends to use the driver's default implementation.

## Peripheral Open Sequence

This is an initialization that needs to be done for each peripheral in the system. The configuration parameters described in this section are peripheral specific.

Application should follow the next steps:

1. Initialize the peripheral configuration parameters with default values using **Enet_initCfg()**. Although the application can fill all parameters manually, it's recommended tofirst get the driver's default values for all parameters and only overwrite the parameters of interest.
    - CPSW peripheral configuration - **Cpsw_Cfg** structure has the configuration parameters for CPSW Enet Peripheral, such as:
        - Configuration of DMA: **EnetDma_Cfg**.
        - VLAN configuration (inner/outer VLAN, customer/service switch): **Cpsw_VlanCfg**.
        - Max packet length transmitted on egress: **Cpsw_Cfg::txMtu**.
        - Configuration of the host (CPPI) port: **CpswHostPort_Cfg**.
        - Configure of the ALE submodule: **CpswAle_Cfg**.
        - Configure of the CPTS submodule: **CpswCpts_Cfg**.
        - Configuration of the MDIO submodule: **Mdio_Cfg**.
        - Configuration of CPSW Resource Partition: **EnetRm_ResCfg**.
    - ICSSG peripheral configuration - Similarly, ICSSG has its own dedicated configuration structure called **Icssg_Cfg** where the application passes ICSSG specific configuration parameters, such as:
        - Configuration of ICSSG DMA: **EnetUdma_Cfg**.
        - ICSSG firmware: Icssg_FirmwareDownloadCfg.
        - Configuration of the MDIO submodule: **Mdio_Cfg**.
        - Configuration of DMA Resource Partition: **EnetRm_ResCfg**.
2. Once done with the configuration of the parameters, the UDMA driver has to be opened. Enet utils library provides a helper function called EnetAppUtils_udmaOpen() to open the UDMA driver, and returns a handle that can be passed to the peripheral configuration parameters, i.e. **Cpsw_Cfg::dmaCfg**.
3. **Enet_open()** to open a peripheral, passing the configuration parameters previously initialized. **Enet_open()** function takes the following arguments:
    - Peripheral type - This specifies the type or class of the peripheral, i.e. CPSW, ICSSG or other.
    - Instance number - This argument specifies the instance number of the peripheral. Refer to **Enet Peripherals** section for further information about specific peripheral types and instance numbers.
    - Configuration structure - A pointer to the peripheral-specific configuration structure, i.e. **Cpsw_Cfg** or **Icssg_Cfg**.
    - Size of the configuration structure.
4. If the module is opened successfully, the API will return a valid handle pointer to the Enet driver. It's worth noting that there will be a unique Enet handle (**Enet_Handle**) for each peripheral opened.
5. Attach the core with the Resource Manager (RM) using **ENET_PER_IOCTL_ATTACH_CORE**. IOCTL. To use IOCTLs, the application must have the following:
    - Valid handle to the driver.
    - Core ID, which can be obtained using **EnetSoc_getCoreId()**.
    - Valid IOCTL command.
    - Valid pointer to the IOCTL parameters.
6. Once the application attaches the core with Resource Manager (RM), the IOCTL call will return core key which has to be used in all further RM-related calls.
7. A MAC address for the host port is to be obtained using Enet utils helper function EnetAppUtils_allocMac() and the corresponding entry in the ALE table can be added using EnetAppUtils_addHostPortEntry().
8. Allocate memory for Ring Accelerators, Ethernet packets, etc. Enet LLD examples use `EnetAppMemUtils` to take care of all memory allocation and freeing operations. The developer can take this as reference or can implement their own memory allocation functions.

## Port Open Sequence

The MAC ports can be opened in MAC-to-PHY or MAC-to-MAC mode. In MAC-to-PHY mode, Enet LLD's PHY driver state machine will be used to configure the Ethernet PHY. In MAC-to-MAC mode, the PHY driver will be bypassed entirely.

The link speed and duplexity in MAC-to-PHY can be fixed or auto-negotiated, while in MAC-to-MAC, both link speed and duplexity can be fixed only and must be provided by application.

The steps to open and configure ports in either MAC-to-PHY or MAC-to-MAC modes are shown below.

### MAC-PHY link

1. Set the port number in **EnetPer_PortLinkCfg** structure.
2. Set the MAC port interface (RMII, RGMII, SGMII, etc) through the layer, sublayer and variant fields of **EnetPer_PortLinkCfg::mii**.
3. The MAC port configuration parameters is peripheral dependent.
    - CPSW MAC Port configuration:
        - Use **CpswMacPort_initCfg()** to initialize the CPSW MAC port configuration parameters to their default values. Overwrite any parameters as needed.
        - If loopback is to be enabled, set enableLoopback flag in MAC configuration to true. For loopback to be functional the secure flag in host port ALE entry must be set to false. The host port entry update can be done using the **CPSW_ALE_IOCTL_ADD_UCAST** command.
    - ICSSG MAC Port configuration:
        - Use **IcssgMacPort_initCfg()** to fill MAC port configuration parameters with default values provided by the ICSSG driver. The application can then overwrite any configuration parameter as needed, i.e. unicast flooding, multicast flooding, etc.
4. Set PHY configuration parameters: generic and model specific.
    a. Use **EnetPhy_initCfg()** to initialize the PHY generic parameters to their default values, such as auto-negotiation capabilities, strap enable/disable, loopback, etc.

   b. Use PHY specific init config functions to initialize model specific parameters. This init config function is provided by the Ethernet PHYs supported in Enet LLD. For example, **Dp83867_initCfg()** is used to initialize config params for DP83867 PHYs.
   c. Application can also use the Enet helper function EnetBoard_getPhyCfg() to get PHY configuration information for PHYs in TI EVMs.
5. Set the link speed and duplexity configuration in **EnetPer_PortLinkCfg::linkCfg**.
   - For auto-negotiation, use **ENET_SPEED_AUTO** or **ENET_DUPLEX_AUTO**.
   - For fixed (manual) configuration, use the fixed speeds and duplexity values in defined in **Enet_Speed** and **Enet_Duplexity** enumerations.
6. Once all the MAC and PHY configurations are done, the ports can be opened by calling the **ENET_PER_IOCTL_OPEN_PORT_LINK** command.
7. *CPSW Note*: To enable RMII on CPSW9G, external 50 MHz RMII clock from PHY is used on SOC RMII_REF_CLOCK pin. On GESI board, this clock is connected as resistor R225 is not populated. To get RMII_50MHZ_CLK, resistor R225 needs to be populated. We need to move R226 to R225 on GESI board to get this clock.
8. If all the above steps succeeded without any errors, then the Enet driver, the Ethernet peripheral and a MAC port have been opened successfully.
9. After opening the port, the driver will run its PHY state machine to configure the PHY and eventually get *link up* status. Application can query the port link using **ENET_PER_IOCTL_IS_PORT_LINK_UP**.
   - Application can check the PHY alive status (i.e. whether PHY is present on the MDIO bus) using the **ENET_MDIO_IOCTL_IS_ALIVE** command.
10. Once link up is detected, application should do the following steps:
    - CPSW:
      - Set the ALE port state to `Forward` state using **CPSW_ALE_IOCTL_SET_PORT_STATE** IOCTL command.
      - Enable the host port of CPSW by calling the **ENET_HOSTPORT_IOCTL_ENABLE** command.
    - ICSSG:
      - Set the MAC port state to `Forward` state using **ICSSG_PER_IOCTL_SET_PORT_STATE** IOCTL command.

The following code snippet shows how a MAC-PHY link is opened (step 1-6 above). CPSW MAC port 1 connected to a DP83867 RGMII PHY has been chosen for this example.

```
Enet_IoctlPrms prms;
EnetPer_PortLinkCfg portLinkCfg;
EnetMacPort_LinkCfg *linkCfg = &portLinkCfg.linkCfg;
EnetMacPort_Interface *mii = &portLinkCfg.mii;
CpswMacPort_Cfg cpswMacCfg;
EnetPhy_Cfg *phyCfg = &portLinkCfg.phyCfg;
Dp83867_Cfg dp83867Cfg;

/* Step 1 - MAC port 1 */
portLinkCfg->macPort = ENET_MAC_PORT_1;

/* Step 2 - Set port type to RGMII */
mii->layerType    = ENET_MAC_LAYER_GMII;
mii->sublayerType = ENET_MAC_SUBLAYER_REDUCED;
mii->variantType  = ENET_MAC_VARIANT_FORCED;

/* Step 3 - Initialize MAC port configuration parameters */
CpswMacPort_initCfg(&cpswMacCfg);
portLinkCfg.macCfg = &cpswMacCfg;

/* Step 4a - Set PHY generic configuration parameters */
EnetPhy_initCfg(phyCfg);
phyCfg->phyAddr = 0U;

/* Step 4b - DP83867 PHY specific configuration */
Dp83867_initCfg(&dp83867Cfg);
dp83867Cfg.ledMode[1] = DP83867_LED_LINKED_100BTX;
dp83867Cfg.ledMode[2] = DP83867_LED_RXTXACT;
dp83867Cfg.ledMode[3] = DP83867_LED_LINKED_1000BT;
EnetPhy_setExtendedCfg(phyCfg, &dp83867Cfg, sizeof(dp83867Cfg));

/* Step 5 - Set link speed/duplexity to auto-negotiation */
linkCfg->speed     = ENET_SPEED_AUTO;
linkCfg->duplexity = ENET_DUPLEX_AUTO;

/* Step 6 - Open port link */
ENET_IOCTL_SET_IN_ARGS(&prms, &portLinkCfg);
status = Enet_ioctl(hEnet, coreId, ENET_PER_IOCTL_OPEN_PORT_LINK, &prms);
```

### MAC-to-MAC link

1. Set the port number in **EnetPer_PortLinkCfg::macPort**.
2. Set the MAC port interface (RMII, RGMII, SGMII, etc) through the layer, sublayer and variant fields of **EnetPer_PortLinkCfg::mii**.
3. Initialize the MAC configuration parameters using **CpswMacPort_initCfg()** and manually overwrite any parameters that differ from defaults.
4. Set PHY address to **ENETPHY_INVALID_PHYADDR** to indicate that this is a PHY-less connection.
5. Set the link speed and duplexity configuration in **EnetPer_PortLinkCfg::linkCfg** to match those of the partner MAC port. The speed and duplexity must be fixed values, they can't be **ENET_SPEED_AUTO** or **ENET_DUPLEX_AUTO** as they are used for auto-negotiation which is not relevant in MAC-to-MAC mode.
6. Once all the MAC and PHY configurations are done, the ports can be opened by calling the **ENET_PER_IOCTL_OPEN_PORT_LINK** command.
7. If all the above steps succeeded without any errors, then the Enet driver, the Ethernet peripheral and a MAC port have been opened successfully.
8. Once link up is detected, application should do the following steps:
   - CPSW:
     - Set the ALE port state to `Forward` state using **CPSW_ALE_IOCTL_SET_PORT_STATE** IOCTL command.
     - Enable the host port of CPSW by calling the **ENET_HOSTPORT_IOCTL_ENABLE** command.
   - ICSSG:
     - Set the MAC port state to `Forward` state using **ICSSG_PER_IOCTL_SET_PORT_STATE** IOCTL command.

The following code snippet shows how a MAC-to-MAC link is opened (steps 1-6 above). CPSW MAC port 1 connected to partner MAC using RGMII interface at 1 Gbps.

```
Enet_IoctlPrms prms;
EnetPer_PortLinkCfg portLinkCfg;
EnetMacPort_LinkCfg *linkCfg = &portLinkCfg.linkCfg;
```

```
EnetMacPort_Interface *mii = &portLinkCfg.mii;
CpswMacPort_Cfg cpswMacCfg;
EnetPhy_Cfg *phyCfg = &portLinkCfg.phyCfg;

/* Step 1 - MAC port 1 */
portLinkCfg->macPort = ENET_MAC_PORT_1;

/* Step 2 - Set port type to RGMII */
mii->layerType    = ENET_MAC_LAYER_GMII;
mii->sublayerType = ENET_MAC_SUBLAYER_REDUCED;
mii->variantType  = ENET_MAC_VARIANT_FORCED;

/* Step 3 - Initialize MAC port configuration parameters */
CpswMacPort_initCfg(&cpswMacCfg);
portLinkCfg.macCfg = &cpswMacCfg;

/* Step 4 - Indicate that this is a PHY-less MAC-to-MAC connection */
phyCfg->phyAddr = ENETPHY_INVALID_PHYADDR;

/* Step 5 - Link speed/duplexity must be explicit set, cannot be set to auto */
linkCfg->speed    = ENET_SPEED_1GBIT;
linkCfg->duplexity = ENET_DUPLEX_FULL;

/* Step 6 - Open port link */
ENET_IOCTL_SET_IN_ARGS(&prms, &portLinkCfg);
status = Enet_ioctl(hEnet, coreId, ENET_PER_IOCTL_OPEN_PORT_LINK, &prms);
```

## Packet Send/Receive Sequence

1. The DMA channels to enable data transfer are to be opened. It can be done using the below steps:
   - Initialize the Tx Channel and Rx Flow parameters using **EnetDma_initTxChParams()** and **EnetDma_initRxChParams()**, respectively.
   - For Tx Channel set the following parameters:
     - **EnetUdma_OpenTxChPrms::chNum**: Tx Channel number.
     - **EnetUdma_OpenTxChPrms::hUdmaDrv**: UDMA driver handle obtained before.
     - **EnetUdma_OpenTxChPrms::ringMemAllocFxn**: EnetMem_allocRingMem. or equivalent user preferred function.
     - **EnetUdma_OpenTxChPrms::ringMemFreeFxn**: EnetMem_freeRingMem or equivalent user preferred function.
     - **EnetUdma_OpenTxChPrms::numTxPkts**: number of Tx packets used to alloc ring elements, and descriptors.
     - **EnetUdma_OpenTxChPrms::disableCacheOpsFlag**: true/false.
     - **EnetUdma_OpenTxChPrms::dmaDescAllocFxn**: EnetMem_allocDmaDesc or equivalent user preferred function.
     - **EnetUdma_OpenTxChPrms::dmaDescFreeFxn**: EnetMem_freeDmaDesc or equivalent user preferred function.
     - EnetUdma_OpenTxChPrms::hCbArg: Argument to be used for the callback routines.
   - For Rx Flow set the following parameters:
     - **EnetUdma_OpenRxFlowPrms::startIdx**: Rx flow start index.
     - **EnetUdma_OpenRxFlowPrms::flowIdx**: Rx flow number.
     - **EnetUdma_OpenRxFlowPrms::hUdmaDrv**: UDMA driver handle obtained previously.
     - **EnetUdma_OpenRxFlowPrms::ringMemAllocFxn**: EnetMem_allocRingMem or equivalent user preferred function.
     - **EnetUdma_OpenRxFlowPrms::ringMemFreeFxn**: EnetMem_freeRingMem or equivalent user preferred function.
     - **EnetUdma_OpenRxFlowPrms::numRxPkts**: number of Rx packets used to alloc ring elements, and descriptors.
     - **EnetUdma_OpenRxFlowPrms::rxFlowMtu**: Maximum receive packet length for this flow.
     - **EnetUdma_OpenRxFlowPrms::disableCacheOpsFlag**: true/false.
     - **EnetUdma_OpenRxFlowPrms::dmaDescAllocFxn**: EnetMem_allocDmaDesc or equivalent user preferred function.
     - **EnetUdma_OpenRxFlowPrms::dmaDescFreeFxn**: EnetMem_freeDmaDesc. or equivalent user preferred function.
     - EnetUdma_OpenRxFlowPrms::hCbArg: Argument to be used for the callback routines.
   - After setting the parameters open the channel and flow using Enet utils helper functions EnetAppUtils_openTxCh() and EnetAppUtils_openRxFlow(), respectively.
2. Now that the Ethernet peripheral (CPSW or ICSSG) and UDMA are configured, the application can start the data transfer.
   - To transmit data from application:
     - Call **EnetDma_submitTxPktQ()** to submit the packets that are ready to be transmitted to Tx Free Queue.
     - Call **EnetDma_retrieveTxPktQ()** to retrieve back the packets that were successfully transmitted from Tx Completion Queue.
   - To receive packets that are queued up in the CPSW hardware which are ready to be received by the application:
     - Call **EnetDma_retrieveRxPktQ()** to retrieve packets from Rx Completion Queue that are sent to the application.
     - Call **EnetDma_submitRxPktQ()** to submit new packets to Rx Free Queue, so that newly received packets can be copied.

## Packet Timestamping Sequence

1. RX packet timestamping doesn't require any enable sequence. The timestamps will be returned to the application along with the DMA packet info data, that is, in the **EnetUdma_PktTsInfo::rxPktTs** field of **EnetUdma_PktInfo::tsInfo** after retrieving RX packets using **EnetDma_retrieveRxPktQ()**.
   - CPSW timestamps are returned in clock cycles.
   - ICSSG timestamps returned to the application require an additional conversion step using **ENET_TIMESYNC_IOCTL_TS_TO_NS** command. This IOCTL will return a timestamp value in nanoseconds.
2. Application must register a callback function in advance for TX timestamp event (**ENET_EVT_TIMESTAMP_TX**) via **Enet_registerEventCb()** function. The callback registration can be done at any time before enabling TX timestamping.
3. TX packet timestamping requires to be enabled per packet via DMA packet info **EnetUdma_PktInfo::tsInfo** structure:
   - Enable host TX timestamp by setting **EnetUdma_PktTsInfo::enableHostTxTs** to true.
   - Pass a sequence id which will be used later to correlate the timestamp with the packet it belongs to. The sequence id is passed in **EnetUdma_PktTsInfo::txPktSeqId**.
   - TX packet domain and message type are relevant only for CPSW peripheral. They should be set in **EnetUdma_PktTsInfo::txPktMsgType** and **EnetUdma_PktTsInfo::txPktDomain**.

4. After the packet is submitted for transmission using **EnetDma_submitTxPktQ()**, the timestamp can be retrieved using **Enet_poll()** function.
    - Application calls **Enet_poll()** with event type **ENET_EVT_TIMESTAMP_TX** periodically until the register callback is called.
    - The returned timestamp is a value in clock cycles for CPSW peripheral and a value in nanoseconds for ICSSG peripheral.
    - The sequence id set to before sending packet for transmission in field **EnetUdma_PktTsInfo::txPktSeqId** will be passed to the registered callback. Application can use this value to correlate the packet it comes from.

## IOCTL Sequence

1. Some common IOCTLs have already been mentioned in this document so far. But the application can run any other IOCTL supported by the peripheral. There are two kinds of IOCTLs: synchronous and asynchronous. Asynchronous IOCTLs require an additional completion poll step which is explained in **Synchronous and Asynchronous IOCTLs** section of the **Enet LLD IOCTL interface** document.

## Port Close Sequence

1. MAC ports that were opened earlier can be closed using **ENET_PER_IOCTL_CLOSE_PORT_LINK** IOCTL command. This will close the PHY state machine associated with this MAC port, if the port was not open in MAC-to-MAC mode. Application can choose to reopen the port at any time in the future by following the steps listed in **Port Open Sequence** section.

## Peripheral Close Sequence

1. Disable the host port using the **ENET_HOSTPORT_IOCTL_DISABLE** command
2. Close the opened Tx Channel and Rx flow:
    - Close the Rx flow using EnetAppUtils_closeRxFlow().
    - Close the Tx Channel using EnetAppUtils_closeTxCh().
3. Detach the core from Resource Manager using the **ENET_PER_IOCTL_DETACH_CORE** command.
4. Enet driver can now be closed and de-initialized using **Enet_close()** and **Enet_deinit()**.

## Deinit Sequence

This is one-time deinitialization sequence that must be followed when the application no longer wants to use the Enet LLD.

1. **Enet_deinit()** should be called to deinitialize the driver. No further Enet LLD APIs should be called from this point.
2. Finally, close the UDMA driver using EnetAppUtils_udmaclose()

**Back To Top**