MICROSAR Ethernet Driver

Technical Reference

Version 2.1.1

Authors	David Röder, Mark Harsch, Benjamin Groebner
Status	Released

Document Information

History

Author	Date	Version	Remarks
David Röder	2016-12-16	1.00.00	Initial Creation
Mark Harsch	2017-05-10	2.00.00	▶ ESCAN00094983: Extend Core for support of FEC and ENET drivers with extended feature set ▶ Rework to align to most recent technical reference template
Mark Harsch	2017-06-02		Provided information about OS Category 1 interrupt processing
David Röder	2017-07-04	2.01.00	FEATC-1245: FEAT-2151 Extended Ethernet Bus Diagnostic
Benjamin Groebner	2018-02-26	2.01.01	STORY-4103: TASK-65975 Review Integration

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	AUTOSAR_SWS_EthernetDriver.pdf	4.1.1
[2]	AUTOSAR	AUTOSAR_SWS_EthernetDriver.pdf	4.3.0
[3]	AUTOSAR	AUTOSAR_SWS_DET.pdf	4.1.1
[4]	AUTOSAR	AUTOSAR_SWS_DEM.pdf	4.1.1
[5]	AUTOSAR	AUTOSAR_BasicSoftwareModules.pdf	V1.0.0
[6]	Vector	TechnicalReference_Eth_<Driver>.pdf	see delivery
[7]	AUTOSAR	AUTOSAR_SWS_NVRAMManager.pdf	4.1.1
[8]	Vector	TechnicalReference_Asr4Rtm.pdf	see delivery
[9]	Vector	TechnicalReference_Microsar_Os_SafeContext_ASR4.pdf	see delivery

Scope of the Document

This technical reference describes the general use of the Ethernet driver basis software. All aspects which are Ethernet controller specific are described in a separate document [6], which is also part of the delivery.

Caution
We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector´s release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

Contents

Illustrations

Tables

1 Component History

The component history gives an overview over the important milestones that are supported in the different versions of the component.

Component Version	New Features
01.00.xx	Initial Component Release
02.00.xx	Extensions to support VTT specific driver
03.00.xx	Extensions to support FEC and ENET specific drivers with their extended feature set
03.01.xx	FEAT-2151: Extended Ethernet Bus Diagnostic
03.02.xx	Changes due to DrvEth_S6J3xEthAsr migration
03.03.xx	Introduce support for GNU compiler

Table 1-1 Component history

2 Introduction

This document describes the functionality, API and configuration of the AUTOSAR BSW module Ethernet Driver as specified in [1] and the extensions of the Vector Ethernet drivers.

This document describes the functionalities provided by the whole range of Vector Ethernet drivers. Not any driver supports all functionalities. Please refer to [6] to check, which functionality is available and can be used with the specific driver.

Supported AUTOSAR Release*:	4.1.1	
Supported Configuration Variants:	pre-compile	
Vendor ID:	ETH_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	ETH_MODULE_ID	88 decimal (according to ref. [5])

* For the detailed functional specification please also refer to the corresponding AUTOSAR SWS.

The Ethernet Driver provides hardware independent access to configure and control Ethernet Controllers integrated in the microcontroller (MCU) or connected to it via an external interface like for example SPI.

2.1 Architecture Overview

Figure 2-1 shows where the Ethernet Driver is located in the AUTOSAR architecture.

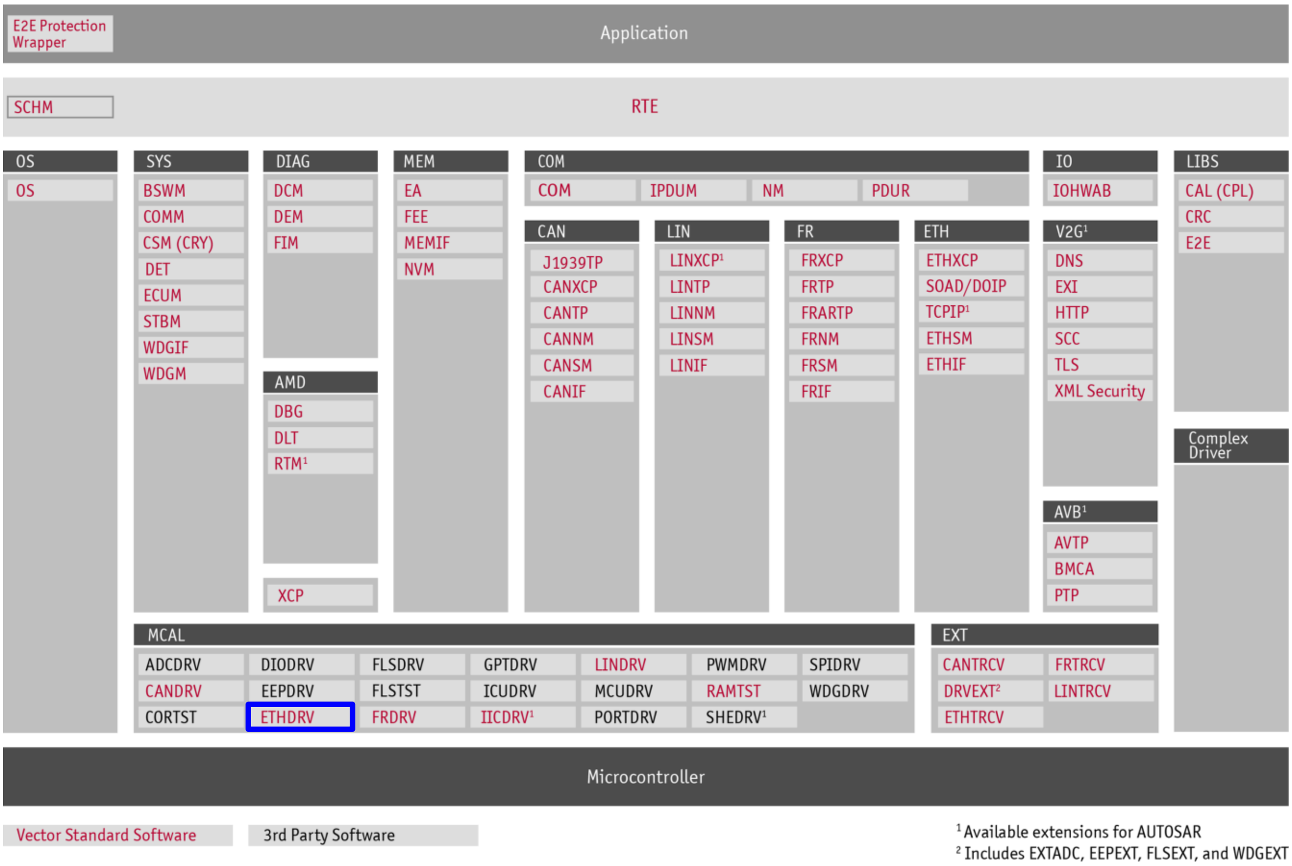

Figure 2-1 AUTOSAR 4.x Architecture Overview

Figure 2-2 shows the interfaces to adjacent modules of the Ethernet Driver. These interfaces are described in chapter 0.

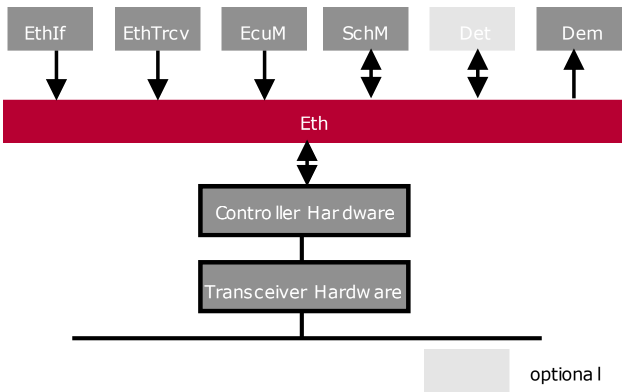

Figure 2-2 Interfaces to adjacent modules of the Ethernet Driver

3 Functional Description

3.1 Features

The features listed in the following tables cover the complete functionality specified for the whole range of Vector Ethernet Drivers.

For extensions beyond or limitation of the described functionalities please see [6] of the specific driver.

The AUTOSAR standard functionality is specified in [1], the corresponding features are listed in the tables

> Table 3-1 Supported AUTOSAR standard conform features

> Table 3-2 Not supported AUTOSAR standard conform features

Vector Informatik provides further Ethernet Driver functionality beyond the AUTOSAR standard. The corresponding features are listed in the tables

> Table 3-3 Features provided beyond the AUTOSAR standard

> Table 3-4 Platform dependent Ethernet features

The following features specified in [1] are supported:

Supported AUTOSAR Standard Conform Features
Initialization of the Ethernet Controller
Activation/Deactivation of the Ethernet Controller during runtime
Reception/Transmission of Ethernet Frames according to IEEE802.3
Retrieve current active MAC address
Set MAC address
Modification of the MAC address filter
Access of hardware connected via MDIO interface according to IEEE802.3 clause 22
Retrieval of statistic counters located in the register space of the Ethernet Controller
Retrieval of the modules version info
Report development errors to DET
Report production errors to DEM
Operation in interrupt mode
Operation in polling mode
Multiple Ethernet controller operation
Multiple Ethernet driver operation

Table 3-1 Supported AUTOSAR standard conform features

3.1.1 Deviations

The following features specified in [1] are not supported:

Not Supported AUTOSAR Standard Conform Features
Link-time configuration variant

Not Supported AUTOSAR Standard Conform Features
Post-build configuration variant

Table 3-2 Not supported AUTOSAR standard conform features

3.1.2 Additions/ Extensions

The following features are provided beyond the AUTOSAR standard:

Features Provided Beyond The AUTOSAR Standard
Storage of MAC address in NV-RAM for re-initialization after MCU power-cycle
Extended Buffer Configuration
Runtime Measurement
Flexible Interrupt configuration
Pre-/Post-ControllerInit User-Callouts
Time Synchronization
Quality of Service
FQTSS (Traffic Shaping)
Checksum Offloading
Receive Buffer Segmentation
Rx and Tx Statistics

Table 3-3 Features provided beyond the AUTOSAR standard

3.1.3 Platform Dependent Features

Table 3-4 lists all driver/hardware dependent features provided by Vector drivers.

Please refer to [6] for details if the noted features are available by the specific driver used.

Features Provided Driver/Hardware Specific
Time Synchronization
Quality of Service
FQTSS (Traffic Shaping)
Checksum Offloading
Receive Buffer Segmentation

Table 3-4 Platform dependent Ethernet features

3.2 Initialization

3.2.1 High-Level Initialization

The Ethernet Driver is initialized by calling the `Eth_30_<Driver>_Init()` service with `NULL_PTR` as configuration pointer due to Pre-Compile-Configuration-Variant only support

The Ethernet controller itself is initialized by calling the `Eth_30_<Driver>_ControllerInit()` service for the respective controller to be initialized.

Note
When using a MICROSAR SIP the initialization is done by the stack itself without any user interaction.

Caution
If startup code doesn't initialize the RAM and therefore some data isn't set to a predefined value the Ethernet Driver relies on (e.g. zero initialized data), the function `Eth_30_<Driver>_InitMemory()` must be called during startup. This function sets the predefined values usually set by the startup code.

3.2.2 Low-Level Initialization

The Ethernet Driver relies on the proper configuration of the MCU, which must be done by the MCAL modules. Table 3-5 contains information about the modules involved and what preconditions must be provided by them for proper functionality of the Ethernet Controller.

AUTOSAR MCAL module	Preconditions
Port	Initialization of the MCU pins for the MII interface to the Ethernet Transceiver/Ethernet Switch Port.
Mcu	Initialization of the clocks provided to the Ethernet Controller.

Table 3-5 Lower-Level-Initialization

3.3 States

The Ethernet Driver adopts the states described in the following table:

State	Description
`ETH_STATE_UNINIT`	The driver wasn't initialized yet and all its state variables are set to an undefined value.
`ETH_STATE_INIT`	The driver was initialized and its module global state variables are set to a defined value allowing starting initialization of the Ethernet controller.
`ETH_STATE_ACTIVE`	At least one Ethernet controller and its related states were initialized and ready for operation.

Table 3-6 Ethernet Driver states

The Ethernet Controller adopts the states described in the following table:

State	Description
`ETH_MODE_DOWN`	The Ethernet controller is turned off and can't be used for communication.
`ETH_MODE_ACTIVE`	The Ethernet controller is turned on and able to transmit/receive

	frames.

Table 3-7 Ethernet Controller states

Most APIs of the driver are only allowed to be called in specific states either of the driver or the Ethernet Controller. For details see the service descriptions in 5.

3.4 Main Functions

The Ethernet Driver has one main function described in the following table:

Eth_30_<Driver>_MainFunction()		
Existence	Timing	Description
Always	Scheduled according to the period provided by the configuration parameter `EthMainFunctionPeriod`.	Processes driver specific task described in [6].

Table 3-8 Eth_30_<Driver>_MainFunction() description

Note
The main function isn't declared by the Ethernet Driver itself but declaration is provided by the SchM during generation with DaVinci Configurator PRO.

3.5 Error Handling

3.5.1 Development Error Reporting

By default, development errors are reported to the DET using the service `Det_ReportError()` as specified in [3], if development error reporting is enabled (i.e. pre-compile parameter `ETH_30_<DRIVER>_DEV_ERROR_DETECT==STD_ON`).

If another module is used for development error reporting, the function prototype for reporting the error can be configured by the integrator, but must have the same signature as the service `Det_ReportError()`.

The reported Ethernet Driver ID is 88.

The reported service IDs identify the services which are described in [3]. Table 3-9 presents the service IDs and the related services:

Service ID	Service
0x01	`ETH_30_<DRIVER>_SID_INIT`
0x02	`ETH_30_<DRIVER>_SID_CONTROLLER_INIT`

Service ID	Service
0x03	ETH_30_<DRIVER>_SID_SET_CONTROLLER_MODE
0x04	ETH_30_<DRIVER>_SID_GET_CONTROLLER_MODE
0x05	ETH_30_<DRIVER>_SID_WRITE_MII
0x06	ETH_30_<DRIVER>_SID_READ_MII
0x07	ETH_30_<DRIVER>_SID_GET_COUNTER_STATE
0x08	ETH_30_<DRIVER>_SID_GET_PHYS_ADDR
0x09	ETH_30_<DRIVER>_SID_PROVIDE_TX_BUFFER
0x0A	ETH_30_<DRIVER>_SID_TRANSMIT
0x0B	ETH_30_<DRIVER>_SID_RECEIVE
0x0C	ETH_30_<DRIVER>_SID_TX_CONFIRMATION
0x0D	ETH_30_<DRIVER>_SID_GET_VERSION_INFO
0x0E	ETH_30_<DRIVER>_SID_GET_RX_STATS
0x0F	ETH_30_<DRIVER>_SID_GET_TX_STATS
0x12	ETH_30_<DRIVER>_SID_UPDATE_PHYS_ADDR_FILTER
0x13	ETH_30_<DRIVER>_SID_SET_PHYS_ADDR
0x30	ETH_30_<DRIVER>_TIME_SYNC_SID_GET_GLOBAL_TIME
0x31	ETH_30_<DRIVER>_TIME_SYNC_SID_SET_GLOBAL_TIME
0x32	ETH_30_<DRIVER>_TIME_SYNC_SID_SET_CORRECTION_TIME
0x33	ETH_30_<DRIVER>_TIME_SYNC_SID_ENABLE_EGRESS_TIMESTAMP
0x34	ETH_30_<DRIVER>_TIME_SYNC_SID_GET_EGRESS_TIMESTAMP
0x35	ETH_30_<DRIVER>_TIME_SYNC_SID_SET_EGRESS_TIMESTAMP
0x40	ETH_30_<DRIVER>_SID_SET_BANDWIDTH_LIMIT
0x41	ETH_30_<DRIVER>_SID_GET_BANDWIDTH_LIMIT

Table 3-9 Service IDs

The errors reported to DET are described in Table 3-10:

Error Code		Description
0x00	ETH_30_<DRIVER>_E_NO_ERROR	No error occurred
0x01	ETH_30_<DRIVER>_E_INV_CTRL_IDX	API called with wrong controller index
0x02	ETH_30_<DRIVER>_E_NOT_INITIALIZED	API called while module was not initialized correctly
0x03	ETH_30_<DRIVER>_E_INV_POINTER	API called with wrong pointer parameter (NULL_PTR)
0x04	ETH_30_<DRIVER>_E_INV_PARAM	API called with invalid parameter
0x05	ETH_30_<DRIVER>_E_INV_CONFIG	Initialization triggered for an unknown configuration
0x06	ETH_30_<DRIVER>_E_INV_MODE	API called while module was in an invalid mode
0x07	ETH_30_<DRIVER>_E_INV_ALIGNMENT	Invalid alignment of buffer or descriptor

Table 3-10 Errors reported to DET

3.5.2 Production Code Error Reporting

By default, production code related errors are reported to the DEM using the service `Dem_ReportErrorStatus()` as specified in [4].

The errors reported to DEM are described in Table 3-11:

Error Code	Description
`ETH_E_ACCESS`	Accessing the controller failed

Table 3-11 Errors reported to DEM

3.6 MAC Address

The MAC address of a controller may be configured to a default value via the MAC address configuration parameter within the configuration tool. Additionally, the user may call the API `Eth_30_<Driver>_SetPhysAddr` to change the MAC address at runtime.

Caution
In case the user does not provide a default address within the configuration tool, the address is undefined after ECU startup. Therefore it is essential that the user sets the MAC at runtime before any Ethernet communication starts!

The default behavior of `Eth_30_<Driver>_SetPhysAddr` is non-persistent. The user must provide the MAC address each time after an ECU restart (in case no default is configured within the configuration tool). An extended, persistent behavior can be enabled via the feature “Enable MAC address write access” discussed in the next section.

3.6.1 NV-RAM

The feature “Enable MAC address write access” enables NvM support for the API `Eth_30_<Driver>_SetPhysAddr`. In this case the MAC address gets written into a NV-RAM block and is loaded at controller initialization. This ensures that the MAC address is persistent even after a system restart.

The NV-RAM block for a MAC must have a length of 6 bytes. The initial value is configured via the MAC address configuration parameter within the configuration tool. The NV-RAM block must be managed by the AUTOSAR NV-Manager (NvM) and is addressed by a NvM block descriptor (refer to [7]).

The NV-RAM blocks must be processed during the `NvM_ReadAll()` and `NvM_WriteAll()` function calls.

The symbols for the ROM and RAM mirrors are listed below

> ROM default block: `Eth_30_<Driver>_VPhysSrcAddrRomDefault_<CtrlIdx>`

> RAM mirror: `Eth_30_<Driver>_VPhysSrcAddr_<CtrlIdx>`

where <CtrlIdx> is the index of the controller.

3.7 Hardware Cancellation

The hardware cancellation feature can be applied for different hardware operations by setting the corresponding cycle number. While performing these operations a counter is incremented in a loop in every cycle until either the operation is finished or the configured maximum value of the cycle number has been reached. In this case the operation has failed. The configurable maximum cycle counter values are:

> Controller Reset Loop Cycles: Specifies the number of loop cycles after which the controller reset timeout occurs.
> Controller MII Loop Cycles: Specifies the number of loop cycles after which the timeout for MII accesses occurs.
> Controller Loop Cycles: Specifies the number of loop cycles after which other hardware dependent loop timeouts occur.

Caution
The Loop Cycles Counter is not related to any time base! Therefore the time elapsing is highly platform and configuration dependent (e.g. CPU clock dependent)!

3.8 Runtime Measurement

Runtime measurement allows to measure the processing time of specific functions of the driver. Therefore so called measurement points are used. These points are automatically created in the Rtm module, if the module is contained in the configuration.

This description only shows the Ethernet drivers specific aspect of the runtime measurement. For a more detailed description of the runtime measurement feature and the Rtm module providing the service see [8].

Dependent on the Ethernet controller core integrated on the derivative used and the features enabled, more or less measurement points are provided.

Measurement Point	Description
`Eth_30_<Driver>_ControllerInit`	Measures the runtime of the API `Eth_30_<Driver>_ControllerInit()`. The API is used for controller initialization during runtime.

Table 3-12 Runtime Measurement points

> **Note**
> The runtime measurement is enabled by setting `Runtime Measurement Support` in the general settings of the Ethernet driver.

3.9 Ethernet Switch Frame Management

Some Ethernet Switches provide a functionality called frame management. This feature makes it possible to transfer Meta data related to an Ethernet frame over the MII interface. This Meta data is either embedded into the Ethernet Frame itself or transferred as a consecutive Ethernet frame. To allow the Ethernet Driver to process the Ethernet frames without the knowledge which kind of mechanism is used, an interaction between Ethernet Driver and Ethernet Switch Driver is needed.

For a more detailed description of the interface please refer to the Ethernet Switch Drivers technical reference.

> **Note**
> The feature is implicitly activated if an Ethernet Switch Driver is contained in the configuration and uses the Ethernet Controller as frame management interface.

3.10 Pre-/Post-ControllerInit User-Functions

The driver allows to inject integration code in the processing of the `Eth_30_<Driver>_ControllerInit()` API.

This integration code is either executed before the actual code of the `Eth_30_<Driver>_ControllerInit()` (Pre-ControllerInit User-Function), after the actual code (Post-ControllerInit User-Function) or both.

The functions can be used to integrate derivative specific code that must be executed for a proper operation of the Ethernet Controller. Such code could be setting registers in port or clock modules, which are needed to configure the xMII interface between the Ethernet Controller and the Ethernet PHY properly.

The User-Functions will be generated into the `Eth_30_<Driver>_Lcfg.c` file and contain a so-called User-Block-Comment where integration code can be provided, which is persisted and not overridden if the `Eth_30_<Driver>_Lcfg.c` file is generated again.

The function names are `Eth_30_<Driver>_PreControllerInitCallout()` and `Eth_30_<Driver>_PostControllerInitCallout()`.

3.11 Rx and Tx Statistics

With the functions `Eth_30_<Driver>_GetRxStats` and `Eth_30_<Driver>_GetTxStats` the lists of counter values that can be seen in Table 3-13 and Table 3-14 can be obtained from the hardware. Please note, that if a realistic counter value cannot be obtained from the underlying hardware platform, the API returns a special value. See sections 5.2.1 and 5.2.2 for details on the used data type.

Counter Value	Description
Rx Drop Events	The total number of events in which packets were dropped due to lack of ressources.
Rx Octets	The total number of octets of data (including those in bad packets) received on the network (excluding framing bits but including FCS octets).
Rx Packets	The total number of packets (including bad packets, broadcast packets and multicast packets) received.
Rx Broadcast Packets	The total number of good packets received that were directed to the broadcast address.
Rx Multicast Packets	The total number of good packets received that were directed to a multicast address.
Rx Crc/Alignment Errors	The total number of packets received that had a length of between 64 and 1518 octets that had either a bad Frame Check Sequence (FCS) with an integral number of octets (FCS Error) or a bad FCS with a non-integral number of octets (Alignment Error).
Rx Undersize Packets	The total number of packets received that were less than 64 octets long (excluding framing bits, but including FCS octets) and were otherwise well formed.
Rx Oversize Packets	The total number of packets received that were longer than 1518 octets long (excluding framing bits, but including FCS octets) and were otherwise well formed.
Rx Fragments	The total number of packets received that were less than 64 octets in length (excluding framing bits but including FCS octets) and had either a bad FCS with an integral number of octets (FCS Error) or a bad FCS with a non-integral number of octets (Alignment Error).
Rx Jabbers	The total number of packets received that were longer than 1518 octets in length and had either a bad FCS with an integral number of octets (FCS Error) or a bad FCS with a non-integral number of octets (Alignment Error).
Rx Collisions	The best estimate of the total number of collisions on this Ethernet segment.
Rx Packets 64 Octets	The total number of packets (including bad packets) received that were 64 octets in length.
Rx Packets 65 to 127 Octets	The total number of packets (including bad packets) received that were between 65 and 127 octets in length.
Rx Packets 128 to 255 Octets	The total number of packets (including bad packets) received that were between 128 and 255 octets in length.
Rx Packets 256 to 511 Octets	The total number of packets (including bad packets) received that were between 256 and 511 octets in length.
Rx Packets 512 to 1023 Octets	The total number of packets (including bad packets) received that were between 512 and 1023 octets in length.
Rx Packets 1024 to 1518 Octets	The total number of packets (including bad packets) received that were between 1024 and 1518 octets in length.

Counter Value	Description
Rx Unicast Frames	The number of subnetwork-unicast packets delivered to a higher level protocol.

Table 3-13 Reception Statistic Counters

Counter Value	Description
Tx Number Of Octets	The total number of octets transmitted out of the interface, including framing characters.
Tx Non-unicast Packets	The total number of packets that higher level protocols requested to be transmitted to a non-unicast (i.e., a subnetwork-broadcast or subnetwort-multicast) address, including those that were discarded or not sent.
Tx Unicast Packets	The total number of packets that higher level protocols requested to be transmitted to a subnetwork-unicast address, including those that were discarded or not sent.

Table 3-14 Transmission Statistic Counters

3.12 Driver/Hardware Dependent Features

The following sections introduce features which are contained in the software provided by Vector but are not supported on each driver or hardware derivative.

Please refer to [6] section 3.1. where following table can be found that summarizing the availability of the features by the respective driver or on the specific hardware derivative used.

Feature	Driver/Hardware support - = not available ■ = available on specific derivatives ■ = available
Time Synchronization	■
Quality of Service	■
FQTSS (Traffic Shaping)	■
Checksum Offloading	■
Safe Context	■
Receive Buffer Segmentation	-

Table 3-15 Vector Driver Feature Set Support

3.12.1 Time Synchronization

The time synchronization allows synchronizing the Ethernet Controllers timer to a respective time base by retrieving timestamps of Ethernet frames on transmission and reception and manipulating the timer.

Note
The feature is enabled by setting ‘Enable PTP’ in the Ethernet Controller configuration in DaVinci Configurator.

3.12.1.1 Timer

For the support of PTP the Ethernet Controller core implements a timer which maintains the local time. This time base is used to timestamp the frames. Additionally to the time stamping feature the timer can be manipulated by multiple APIs, which will be described in the following sections.

Note
For proper operation of the timer an external clock must be provided. Please refer to the peripheral clock configuration chapter of the Technical Reference provided for the microcontroller derivative used.

The clock frequency provided additionally must be defined in the controller configuration in DaVinci Configurator by setting the parameter ‘PTP Reference Clock Frequency’.

3.12.1.2 Timer Manipulation

The local time can be manipulated to be approximated to a global time negotiated between the nodes in the Ethernet network by a time synchronization protocol like PTP.

The API `Eth_30_<Driver>_SetCorrectionTime()` allows this manipulation. It enables the upper layer to correct the time by an offset or to slow down / accelerate the clock.

3.12.1.3 Global Time Retrieving/Setting

An upper layer is able to retrieve and set the global time. Therefor the APIs `Eth_30_<Driver>_GetGlobalTime()` and `Eth_30_<Driver>_SetGlobalTime()` are provided.

Note
These APIs can be called without the restriction to the Rx Indication or TX Confirmation context.

3.12.1.4 Ingress Time Stamping

If an incoming Ethernet frame is time stamped by the Ethernet Controller, an upper layer is able to retrieve this timestamp by calling the API `Eth_30_<Driver>_GetIngressTimestamp()`.

Caution
The API `Eth_30_<Driver>_GetIngressTimestamp()` must be called in the Rx Indication context to retrieve a proper timestamp for the received Ethernet frame. Otherwise the API will return with the negative return value `E_NOT_OK` and the timestamp is invalid.

Figure 3-1 illustrates the sequence with an EthTSyn (GPtp) upper layer:

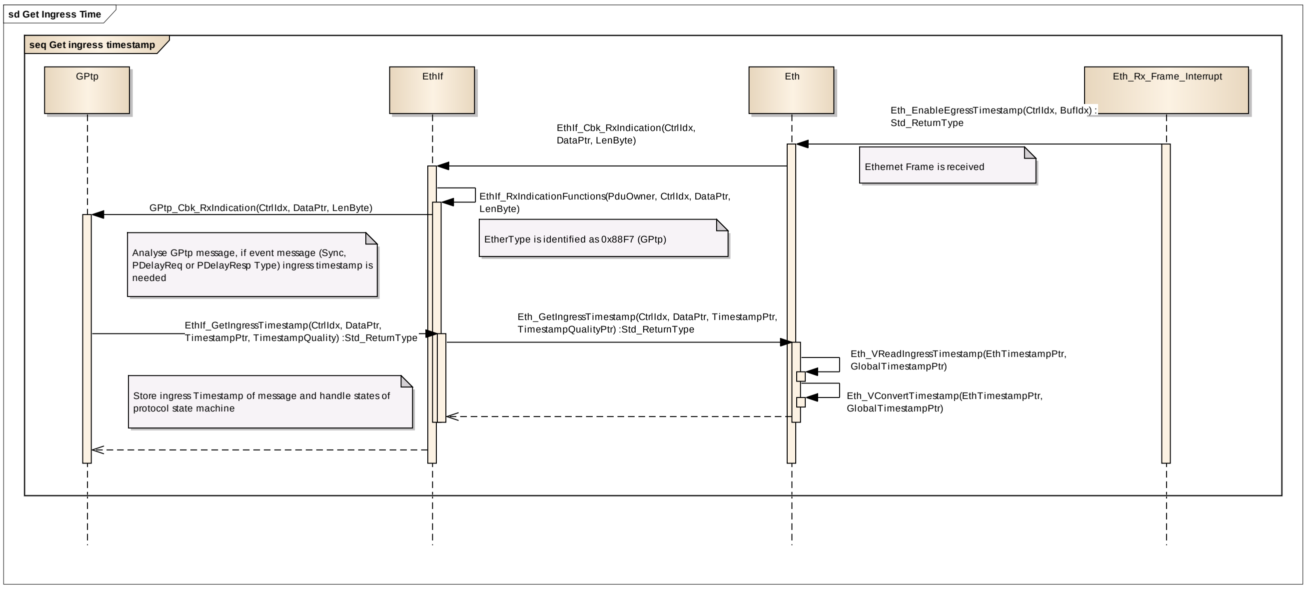

Figure 3-1 Ingress Time-Stamping Sequence Diagram

3.12.1.5 Egress Time Stamping

The time stamping of outgoing Ethernet frames must be triggered by the upper layer itself.

To achieve the time stamping the upper layer must obtain a transmit buffer by calling `Eth_30_<Driver>_ProvideTxBuffer()`. Afterwards the buffer provided is locked and time stamping for this buffer can be enabled by calling `Eth_30_<Driver>_EnableEgressTimestamp()`. The transmission now can be trigger with the API `Eth_30_<Driver>_Transmit()`.

The time stamp itself is retrieved on transmission indication by calling `Eth_30_<Driver>_GetEgressTimestamp()`.

Caution
The API `Eth_30_<Driver>_GetEgressTimestamp()` must be called in the TX Confirmation context to retrieve a proper timestamp for the transmitted Ethernet Frame. Otherwise the API will return with the negative return value `E_NOT_OK` and the timestamp is invalid.

Caution
On transmit of an Ethernet frame to be time-stamped a TX Confirmation must be requested by the upper layer.

Figure 3-2 illustrates the sequence with an EthTSyn (GPtp) upper layer:

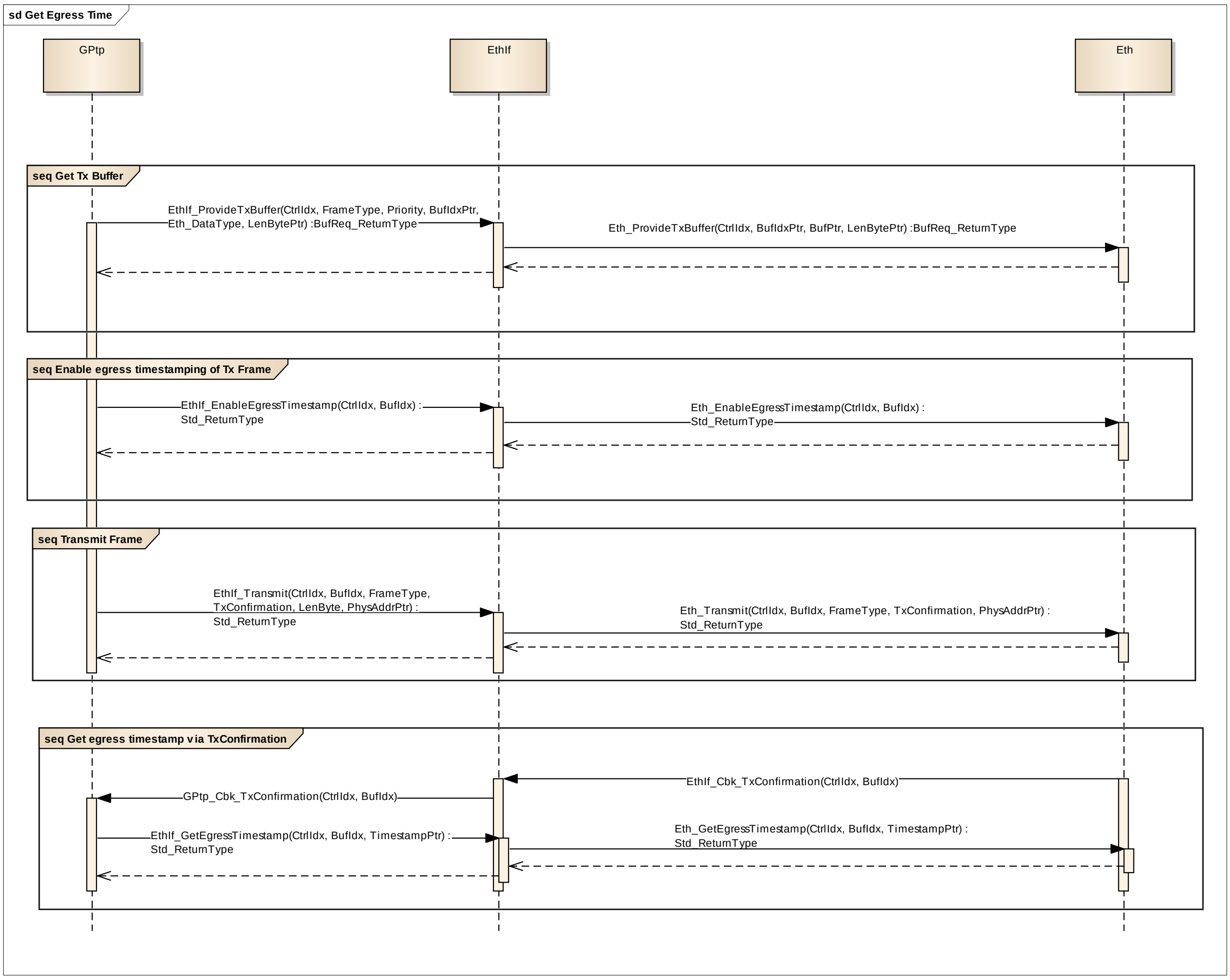

Figure 3-2 Egress Time-Stamping Sequence Diagram

3.12.2 Quality of Service

Quality of Service (QoS) allows prioritizing the Ethernet traffic dependent on the priority of a VLAN tagged frame.

Note
Quality of Service can only be applied on VLAN tagged traffic because it relies on the priority encoded in the VLAN TCI (Tag Control Information).

The following example provides three traffic classes: The best effort class-, class-1- and class-2-traffic, where best effort has the lowest priority and class 2 has the highest priority.

The different classes can be identified in the configuration tool by the parameter `Priority` (see Figure 3-3) and Table 3-16.

Traffic Class	Priority
Best effort	0
Class 1	1
Class 2	2

Table 3-16 Traffic class to ring priority mapping

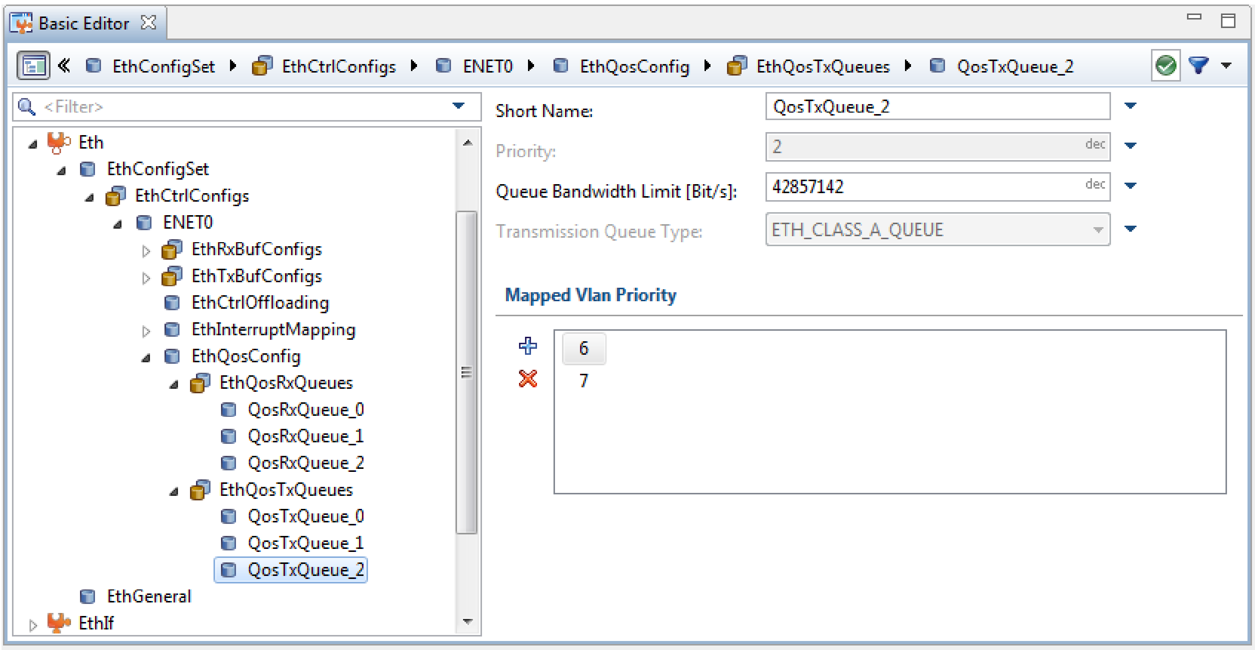

Figure 3-3 QoS traffic class configuration

Like mentioned before, the traffic classification is done with the help of the priority encoded by the VLAN tag. To achieve this, the VLAN priority is used during configuration time to assign traffic to a specific traffic class. This can be done for the Transmission- and for the

Reception-Path as well. In the screenshot, illustrated by Figure 3-3, for example, traffic class 2 on Transmission-Path has assigned all frames sent with the VLAN priority 6 and 7.

3.12.3 FQTSS (Traffic Shaping)

The FQTSS (Forwarding and Queuing Enhancements for Time-Sensitive Streams) support allows applying traffic shaping for Ethernet traffic on the Transmission-Path. The mechanism is based on the QoS traffic classes described in the section 3.12.2.

The traffic classes are assigned to a traffic shaper, which defines an idle slope to shape the Ethernet traffic. The driver allows configuring the shapers in a static way during configuration time by providing the desired bandwidth in [bit/s].

Additionally the current bandwidth limit can be manipulated or retrieved during runtime by using the APIs `Eth_30_<Driver>_SetBandwidthLimit()` and `Eth_30_<Driver>_GetBandwidthLimit()`.

Note
FQTSS (Traffic Shaping) can be activated by configuration.

3.12.4 Checksum Offloading

Checksum Offloading allows calculating and checking checksums of protocols of the TcpIp stack. In principle following checksum calculation can be offloaded to hardware. However not each hardware is capable to calculate all checksums. For details regarding the used driver/hardware please see [6].

Protocol	Description
ICMPv4	Calculation of ICMPv4 checksum
IPv4	Calculation of Ipv4 checksum
UDP over IPv4	Calculation of UDP checksum when embedded in IPv4 protocol
TCP over IPv4	Calculation of TCP checksum when embedded in IPv4 protocol
ICMPv6	Calculation of ICMPv6 checksum
IPv6 with Routing Header Option	Calculation of checksum when IPv6 Routing Header Option is involved
IPv6 with Destination Header Option	Calculation of checksum when IPv6 Destination Header Option is involved
IPv6 with Hop by Hop Header Option	Calculation of checksum when IPv6 Hop by Hop Header Option is involved
UDP over IPv6	Calculation of UDP checksum when embedded in IPv6 protocol
TCP over IPv6	Calculation of TCP checksum when embedded in IPv6 protocol

Table 3-17 Checksum Offloading Capability

Note
Checksum Offloading can be activated by configuration.

3.12.5 Safe Context

CPUs on certain platforms can support different operation modes (e.g. User and Privileged/Supervisor mode). Depending on the used derivative the R/W access to Ethernet hardware registers can be restricted to Privileged mode only. Additionally, depending on the system design, the Ethernet Driver may not be allowed to switch into a Privileged mode by itself. Therefore the protected registers must be accessed from outside of the Ethernet Driver by another software layer (e.g. the Operating System) which is able to switch into the required Privileged mode.

3.12.5.1 Privileged Access with OS Callbacks

The software layer (e.g. the Operating System) which is responsible to handle the privileged access to protected registers must add an area define according to the configured ‘Peripheral Region Os Identifier’ parameter value in the Ethernet controller configuration in DaVinci Configurator Pro.

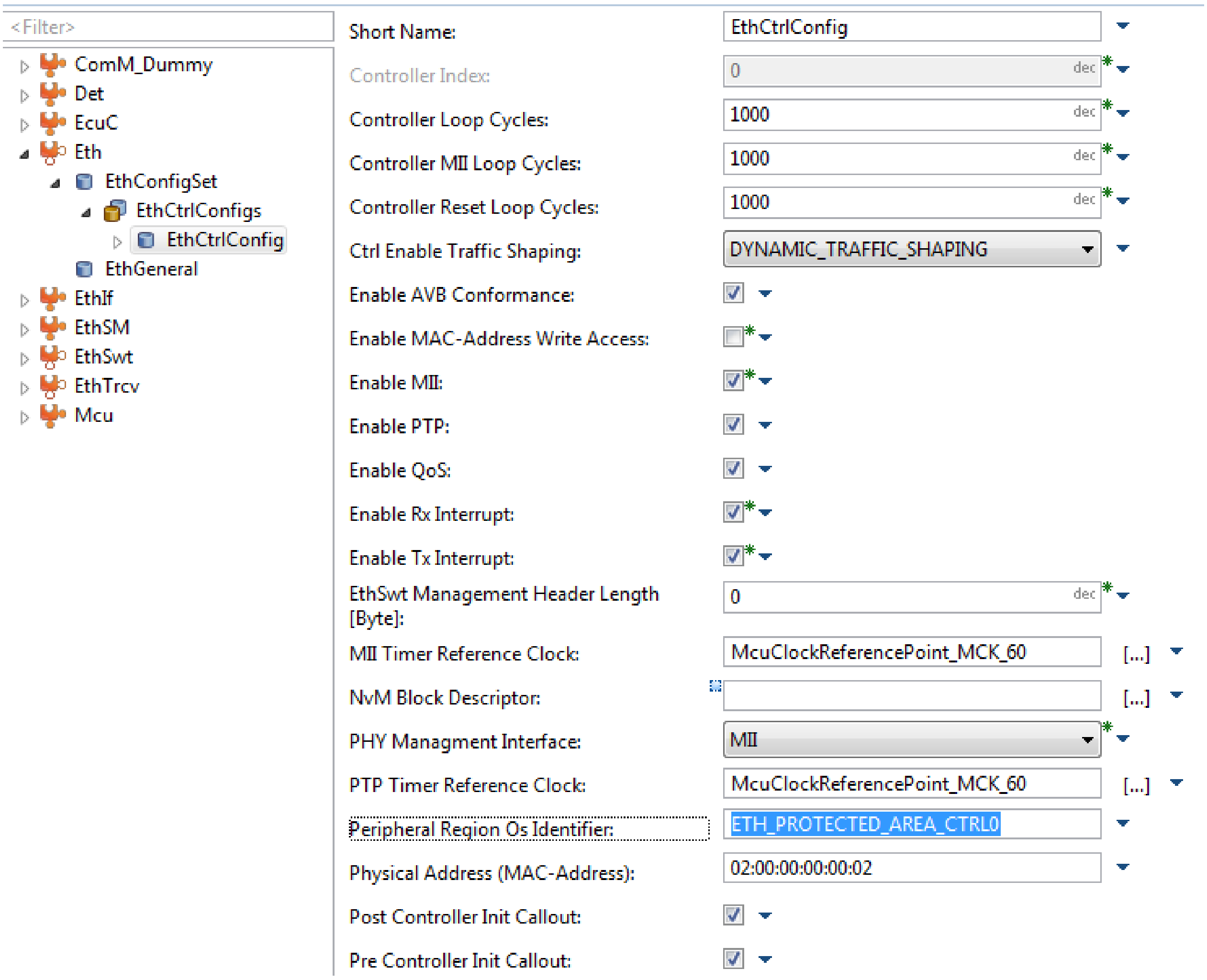

Figure 3-4 Peripheral Region Os Identifier

For each used 'Peripheral Region Os Identifier' the following register address space has to be located inside the privileged region that can be configured within the Operating System module in DaVinci Configurator Pro:

Physical address
ETH base address + 0x0000 to ETH base address + maximum ETH register address offset

Table 3-18 Peripheral Region address range

The OS callbacks that are called by the Ethernet Driver are provided by the Operating System component. Please refer to the Technical Reference [9] of the OS component for further information.

3.12.5.2 Non-Privileged Access by Configuration of a Peripheral Protection Unit

A Peripheral Protection Unit (PPU) which is existing on certain platforms can be configured to permit access to registers which normally require Privileged access mode also for a non-privileged access in User mode. These access rights can be configured on certain platforms individually per peripheral unit, e.g. for a whole Ethernet MAC unit. For further details see e.g. section ‘Peripheral Protection Unit’ of the related hardware manual of the used derivative.

3.12.6 Receive Buffer Segmentation

Some Ethernet MACs provide the ability to receive one frame using multiple descriptors connected to small buffer. When this feature is available and enabled the configuration allows configuring segment sizes different to the maximum frame size, which can be received. The MAC uses as much descriptors per frame as necessary to receive the whole frame.

The implementation of this mechanism is transparent for the upper layers.

Enabling Receive Buffer Segmentation can be activated for a more efficient usage of RAM, which is needed in systems which need to receive large frames only in rare cases, but mainly receive small frames. Instead of having a pool of large buffers available which cause a lot of wasted memory for small frames, the usage of memory scales with the amount of used segments per frame.

Considering an example of three frames with a size of 128 Bytes and one frame with a size of 1500 Bytes, without Receive Buffer Segmentation 4 x 1500 = 6000 Bytes of receive buffers are needed. Using Receive Buffer Segmentation only 15 x 128 = 1920 Bytes of receive buffers are needed.

4 Integration

This chapter gives necessary information for the integration of the MICROSAR Ethernet Driver into an application environment of an ECU.

4.1 Scope of Delivery

The delivery of the Ethernet Driver contains the files which are described in the chapters 4.1.1, 4.1.2 and 4.1.3.

Files noted in 4.1.2 are only contained if driver supports the respective feature.

If a driver needs additional source files the list of the additions can be found in [6].

4.1.1 Static Files

File Name	Description
Eth_30_<Driver>.c	Driver implementation
Eth_30_<Driver>.h	Driver public API
Eth_30_<Driver>_Int.h	Driver private header file
Eth_30_<Driver>_Types.h	Driver public types
Eth_30_<Driver>_CfgAccess_Int.h	Configuration access private header file
Eth_30_<Driver>_HwAccess_Int.h	Hardware access private header file
Eth_30_<Driver>_IrqHandler.c	Interrupt Handler implementation
Eth_30_<Driver>_IrqHandler_Int.h	Interrupt Handler private header file
Eth_30_<Driver>_LL.c	Driver specific implementation
Eth_30_<Driver>_LL.h	Driver specific header file
Eth_30_<Driver>_LL_Cfg.h	Driver specific configuration of core functionality
Eth_30_<Driver>_LL_Types.h	Driver specific types
Eth_30_<Driver>_LL_Int.h	Driver specific private header file
Eth_30_<Driver>_LL_Regs.h	Driver specific register definitions
Eth_30_<Driver>_LL_IrqHandler.c	Driver specific Interrupt Handler implementation
Eth_30_<Driver>_LL_IrqHandler.h	Driver specific Interrupt Handler header file

Table 4-1 Static files

Do not edit manually
Static source code files must not be edited manually!

4.1.2 Static Files depending on available features

File Name	Description
Eth_30_<Driver>_TimeSync.c	Time Synchronization related implementation
Eth_30_<Driver>_TimeSync.h	Time Synchronization related public API
Eth_30_<Driver>_TimeSync_Int.h	Time Synchronization related private header file
Eth_30_<Driver>_LL_TimeSync.c	Driver specific Time Synchronization related implementation
Eth_30_<Driver>_LL_TimeSync.h	Driver specific Time Synchronization related header file
Eth_30_<Driver>_TrafficHandling.c	FQTSS related implementation
Eth_30_<Driver>_TrafficHandling.h	FQTSS related public API
Eth_30_<Driver>_TrafficHandling_Int.h	FQTSS related private header file
Eth_30_<Driver>_LL_TrafficHandling.h	Driver specific FQTSS related header file

Table 4-2 Static files depending on available features

Do not edit manually
Static source code files must not be edited manually!

4.1.3 Dynamic Files

The dynamic files are generated by the configuration tool DaVinci Configurator PRO.

File Name	Description
Eth_30_<Driver>_Cfg.h	Pre-compile time parameter configuration
Eth_30_<Driver>_Lcfg.c	Link-time parameter configuration
Eth_30_<Driver>_Lcfg.h	Link-time parameter configuration declaration
Eth_30_<Driver>_Irq.c	Interrupt service routine definitions

Table 4-3 Generated files

Do not edit manually
Generated source code files must not be edited manually but can be adjusted according to the configuration elements in the DaVinci Configurator Pro tool!

4.2 Compiler Abstraction and Memory Mapping

The objects (e.g. variables, functions, constants) are declared by compiler independent definitions – the compiler abstraction definitions. Each compiler abstraction definition is assigned to a memory section.

Table 4-4 contains the memory section names and the compiler abstraction definitions which are defined for the Ethernet Driver and illustrates their assignment among each other.

Memory Mapping Sections / Compiler Abstraction Definitions	ETH_30_<DRIVER>_CONST	ETH_30_<DRIVER>_VAR	ETH_30_<DRIVER>_CODE
ETH_30_<DRIVER>_START_SEC_CONST_UNSPECIFIED	■		
ETH_30_<DRIVER _START_SEC_CONST_32BIT	■		
ETH_30_<DRIVER>_START_SEC_CONST_16BIT	■		
ETH_30_<DRIVER>_START_SEC_CONST_8BIT	■		
ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_UNSPECIFIED		■	
ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_32BIT		■	
ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_16BIT		■	
ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_8BIT		■	
ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_BUFFER		■	
ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_DESCRIPTOR		■	
ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_BOOLEAN		■	
ETH_30_<DRIVER>_START_SEC_CODE			■

Table 4-4 Compiler abstraction and memory mapping

4.3 Exclusive Areas

This section describes the exclusive areas utilized by the Ethernet Driver to ensure data consistency. The general exclusive areas used by all drivers are listed in the following table. For exclusive areas needed by a specific driver please see [6].

Note
For better readability the prefix `ETH_30_<DRIVER>_EXCLUSIVE_AREA` of the exclusive area names were removed and must be considered during integration.

When using DaVinci Configurator PRO and the respective MICROSAR stack there is no need to configure these areas manually. It is done by the tool automatically.

Exclusive Area	Descriptor
`MII`	This exclusive area ensures a consistent and exclusive access to the MDIO interface of the Ethernet Controller.
`DATA`	This exclusive are ensures the state and data consistency during reception and transmission of an Ethernet frame.
`RXTX_STATS`	This exclusive area ensures data consistency during reading RX and TX statistic values.

Table 4-5 Exclusive areas

4.4 Memory Mapping

The Ethernet buffers and descriptors must be mapped with the AUTOSAR memory mapping feature. For this purpose the two memory sections `ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_BUFFER` and `ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_DESCRIPTOR` are available.

These sections must be aligned according to a specific restriction dependent on the Ethernet Controller Core contained on the used derivative. Refer to [6] for further information.

Following sections describe how the mapping can be done for specific compilers. The sections `EthRamDesc` (used for the descriptors) and `EthRamBuf` (used for the buffers) must be aligned according to the alignment restrictions given in see [6].

4.4.1 Windriver DiabData Compiler

Example

```
#ifdef ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_DESCRIPTOR
# undef ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_DESCRIPTOR
# define START_SEC_VAR_NOINIT_UNSPECIFIED
# pragma section DATA „.EthRamDesc“ „.EthRamDesc“
#endif
#ifdef ETH_30_<DRIVER>_STOP_SEC_VAR_NOINIT_DESCRIPTOR
# undef ETH_30_<DRIVER>_STOP_SEC_VAR_NOINIT_DESCRIPTOR
# pragma section DATA
# define STOP_SEC_VAR
#endif

#ifdef ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_BUFFER
# undef ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_BUFFER
# define START_SEC_VAR_NOINIT_UNSPECIFIED
# pragma section DATA „.EthRamBuf“ „.EthRamBuf“
#endif
#ifdef ETH_30_<DRIVER>_STOP_SEC_VAR_NOINIT_BUFFER
# undef ETH_30_<DRIVER>_STOP_SEC_VAR_NOINIT_BUFFER
# pragma section DATA
# define STOP_SEC_VAR
#endif
```

4.4.2 Greenhills Compiler

Example

```
#ifdef ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_DESCRIPTOR
# undef ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_DESCRIPTOR
# define START_SEC_VAR_NOINIT_UNSPECIFIED
# pragma ghs section bss=".EthRamDesc"
#endif
#ifdef ETH_30_<DRIVER>_STOP_SEC_VAR_NOINIT_DESCRIPTOR
# undef ETH_30_<DRIVER>_STOP_SEC_VAR_NOINIT_DESCRIPTOR
# pragma ghs section bss=default
# define STOP_SEC_VAR
#endif

#ifdef ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_BUFFER
# undef ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_BUFFER
# define START_SEC_VAR_NOINIT_UNSPECIFIED
# pragma ghs section bss=".EthRamBuf"
#endif
#ifdef ETH_30_<DRIVER>_STOP_SEC_VAR_NOINIT_BUFFER
# undef ETH_30_<DRIVER>_STOP_SEC_VAR_NOINIT_BUFFER
# pragma ghs section bss=default
# define STOP_SEC_VAR
#endif
```

4.4.3 Altium TASKING Compiler

Example

```
#ifdef ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_DESCRIPTOR
# undef ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_DESCRIPTOR
# pragma section all “EthRamDesc”
# define START_SEC_VAR_NOINIT_UNSPECIFIED
#endif
#ifdef ETH_30_<DRIVER>_STOP_SEC_VAR_NOINIT_DESCRIPTOR
# undef ETH_30_<DRIVER>_STOP_SEC_VAR_NOINIT_DESCRIPTOR
# pragma section all restore
# define STOP_SEC_VAR
#endif

#ifdef ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_BUFFER
# undef ETH_30_<DRIVER>_START_SEC_VAR_NOINIT_BUFFER
# pragma section all "EthRamBuf"
# define START_SEC_VAR_NOINIT_UNSPECIFIED
#endif
#ifdef ETH_30_<DRIVER>_STOP_SEC_VAR_NOINIT_BUFFER
# undef ETH_30_<DRIVER>_STOP_SEC_VAR_NOINIT_BUFFER
# pragma section all restore
# define STOP_SEC_VAR
#endif
```

4.5 Interrupts

The Ethernet driver supports the configuration of an interrupt mapping.

In case the interrupts are enabled and an OS is running, there are two different interrupt categories, provided by the OS. The Ethernet driver supports both Category 1 and Category 2 interrupts. Additionally the interrupt operation without an OS is possible.

Caution

When using Category 1 interrupts it is platform dependent how the stack for the interrupt is allocated. Because the stack size needed by an Ethernet interrupt is usually greater than for other interrupts the platform specific behavior must be taken into account. Otherwise this could lead to stack overflows.

For detailed information about how the stack for a Category 1 interrupt is allocated on the used platform please refer to the technical reference of the used AUTOSAR OS.

For usage of the interrupt service routines and their configuration please refer to [6].

Due to support for different derivatives a generic interrupt service routine configuration is provided.

5 API Description

For an interfaces overview please see Figure 2-1.

5.1 Type Definitions

The types defined by the Ethernet Driver are described in this chapter.

Type Name	C-Type	Description	Value Range
Eth_ConfigType	void	Controller configuration	NULL_PTR Ctrl uses Link-time configuration
Eth_ReturnType	uint8	Return type of Eth_30_<Driver>_ReadMii() and Eth_30_<Driver>_WriteMii() APIs	ETH_OK Success
			ETH_E_NOT_OK General failure
			ETH_E_NO_ACCESS Ethernet hardware access failure
Eth_ModeType	uint8	Defines all possible controller modes	ETH_MODE_DOWN Controller inactive
			ETH_MODE_ACTIVE Normal operation mode
			ETH_TX_STATE_NOT_TRANSMITTED Frame not yet transmitted
Eth_FrameType	uint16	Ethernet Frame Type	0x0000 - 0xFFFF Any Ethernet frame type
Eth_DataType	uint32	Defines the Ethernet data type	0x00000000 - 0xFFFFFFFF User data
Eth_RxStatusType	uint8	Out parameter in Eth_30_<Driver>_Receive() to indicate that a frame has been received and if so, whether more frames are available or frames got lost	ETH_RECEIVED Frame received, no further frames available
			ETH_NOT_RECEIVED Frame received, no further frames available
			ETH_RECEIVED_MORE_DATA_AVAILABLE Frame received, more frames available
			ETH_RECEIVED_FRAMES_LOST Frame received, some

Type Name	C-Type	Description	Value Range
			frames lost
Eth_FilterActionType	uint8	Describes the action to be taken for the MAC address given to API Eth_30_<Driver>_UpdatePhysAddrFilter()	ETH_ADD_TO_FILTER Add MAC to the filter, meaning allow reception
			ETH_REMOVE_FROM_FILTER Remove MAC from the filter, meaning reception is blocked in the lower layer
Eth_StateType	uint8	Defines all possible controller Driver states	ETH_STATE_UNINIT Ethernet Controller Driver not initialized
			ETH_STATE_INIT Ethernet Controller Driver initialized
			ETH_STATE_ACTIVE Ethernet Controller Driver enabled
			ETH_STATE_DOWN Ethernet Controller Driver disabled
Eth_TimestampQualityType	uint8	Defines the quality of a timestamp	ETH_TIMESTAMP_VALID Returned timestamp is valid
			ETH_TIMESTAMP_INVALID Returned timestamp is invalid, e.g. because of an hardware error
			ETH_TIMESTAMP_UNCERTAIN Timestamp validity is uncertain. This value is not used by the Tricore Ethernet Driver!
Eth_TimediffType	sint32	Defines an offset to correct the time offset of the Ethernet controllers clock	-2,147,483,648 – 2,147,483,647 [ns]
Eth_RateRatioType	float64	Defines a ratio rate that is used to correct the time drift of the Ethernet controllers clock	0.0 - 2.0

Table 5-1 Type definitions

5.2 Structure Definitions

5.2.1 Eth_RxStatsType

Defines a list of Ethernet statistic counters for reception which is also described in [2] and IETF RFC 2819 MIB.

The highest possible value of each counter value range (e.g. value 2^{32}) has the special meaning of 'counter not available'.

The second highest value of each counter value range (e.g. value 2^{32}-1) has the special meaning of 'counter overflow'.

Struct Element Name	C-Type	Description	Value Range
RxStatsDropEvents	uint32	The total number of events in which packets were dropped due to lack of ressources.	[0 ; 2^{32}]
RxStatsOctets	uint32	The total number of octets of data (including those in bad packets) received on the network (excluding framing bits but including FCS octets).	[0 ; 2^{32}]
RxStatsPkts	uint32	The total number of packets (including bad packets, broadcast packets and multicast packets) received.	[0 ; 2^{32}]
RxStatsBroadcastPkts	uint32	The total number of good packets received that were directed to the broadcast address.	[0 ; 2^{32}]
RxStatsMulticastPkts	uint32	The total number of good packets received that were directed to a multicast address.	[0 ; 2^{32}]
RxStatsCrcAlignErrors	uint32	The total number of packets received that had a length of between 64 and 1518 octets that had either a bad Frame Check Sequence (FCS) with an integral number of octets (FCS Error) or a bad FCS with a non-integral number of octets (Alignment Error).	[0 ; 2^{32}]
RxStatsUndersizePkts	uint32	The total number of packets received that were less than 64 octets long (excluding framing bits, but including FCS octets) and were otherwise well formed.	[0 ; 2^{32}]
RxStatsOversizePkts	uint32	The total number of packets received that were longer than 1518 octets long (excluding framing bits, but	[0 ; 2^{32}]

Struct Element Name	C-Type	Description	Value Range
		including FCS octets) and were otherwise well formed.	
RxStatsFragments	uint32	The total number of packets received that were less than 64 octets in length (excluding framing bits but including FCS octets) and had either a bad FCS with an integral number of octets (FCS Error) or a bad FCS with a non-integral number of octets (Alignment Error).	[0 ; 2^{32}]
RxStatsJabbers	uint32	The total number of packets received that were longer than 1518 octets in length and had either a bad FCS with an integral number of octets (FCS Error) or a bad FCS with a non-integral number of octets (Alignment Error).	[0 ; 2^{32}]
RxStatsCollisions	uint32	The best estimate of the total number of collisions on this Ethernet segment.	[0 ; 2^{32}]
RxStatsPkts64 Octets	uint32	The total number of packets (including bad packets) received that were 64 octets in length.	[0 ; 2^{32}]
RxStatsPkts65to127 Octets	uint32	The total number of packets (including bad packets) received that were between 65 and 127 octets in length.	[0 ; 2^{32}]
RxStatsPkts 128to255Octets	uint32	The total number of packets (including bad packets) received that were between 128 and 255 octets in length.	[0 ; 2^{32}]
RxStatsPkts 256to511Octets	uint32	The total number of packets (including bad packets) received that were between 256 and 511 octets in length.	[0 ; 2^{32}]
RxStatsPkts 512to1023Octets	uint32	The total number of packets (including bad packets) received that were between 512 and 1023 octets in length.	[0 ; 2^{32}]
RxStatsPkts 1024to1518Octets	uint32	The total number of packets (including bad packets) received that were between 1024 and 1518 octets in length.	[0 ; 2^{32}]

Struct Element Name	C-Type	Description	Value Range
RxUnicastFrames	uint32	The number of subnetwork-unicast packets delivered to a higher level protocol.	[0 ; 2^{32}]

Table 5-2 Eth_RxStatsType Sructure

5.2.2 Eth_TxStatsType

Defines a list of Ethernet statistic counters for transmission, which is also described in [2] and IETF RFC 2819 MIB.

The highest possible value of each counter value range (e.g. value 2^{32}) has the special meaning of 'counter not available'.

The second highest value of each counter value range (e.g. value 2^{32}-1) has the special meaning of 'counter overflow'.

Struct Element Name	C-Type	Description	Value Range
TxNumberOfOctets	uint32	The total number of octets transmitted out of the interface, including framing characters.	[0 ; 2^{32}]
RxNUcastPkts	uint32	The total number of packets that higher level protocols requested to be transmitted to a non-unicast (i.e., a subnetwork-broadcast or subnetwort-multicast) address, including those that were discarded or not sent.	[0 ; 2^{32}]
TxUniCastPkts	uint32	The total number of packets that higher level protocols requested to be transmitted to a subnetwork-unicast address, including those that were discarded or not sent.	[0 ; 2^{32}]

Table 5-3 Eth_TxStatsType Sructure

5.2.3 Eth_TimeStampType

Defines a timestamp, according to IEEE 1588-2008 (PTP version 2)

Struct Element Name	C-Type	Description	Value Range
nanoseconds	uint32	Nanoseconds part of time	[0 ; 999,999,999] [0x00 ; 0x3B9A_C9ff]
seconds	uint32	32 LSB of the 48 Bit seconds part of time	[0 ; 2^{32}]

Struct Element Name	C-Type	Description	Value Range
secondsHi	uint16	16 MSB of the 48 Bit seconds part of time	[0 ; 2^{16}]

Table 5-4 Eth_TimeStampType Structure

5.3 API Table

5.3.1 AUTOSAR API

1.1.1 Eth_30_Core_InitMemory

Prototype	
`void Eth_30_Core_InitMemory (void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
Inizializes *_INIT_*-variables.	
Particularities and Limitations	
Module is uninitialized Service to initialize module global variables at power up. This function initializes the variables in *_INIT_* sections. Used in case they are not initialized by the startup code.	
Call context	
> TASK > This function is Synchronous > This function is Non-Reentrant	

Table 5-5 Eth_30_Core_InitMemory

1.1.2 Eth_30_Core_Init

Prototype	
`void Eth_30_Core_Init (const Eth_ConfigType *CfgPtr)`	
Parameter	
CfgPtr [in]	Pointer to post-build configuration or null pointer
Return code	
void	none
Functional Description	
Initializes the module.	

Particularities and Limitations
Module's *_INIT*_-variables are initialized either by Eth_30_Core_InitMemory() or startup code Function initializes the module Eth_30_Core. It initializes all variables and sets the module state to initialized.
Call context
> TASK > This function is Synchronous > This function is Non-Reentrant

Table 5-6 Eth_30_Core_Init

1.1.3 Eth_30_Core_ControllerInit

Prototype	
`Std_ReturnType **Eth_30_Core_ControllerInit** (uint8 CtrlIdx, uint8 CfgIdx)`	
Parameter	
CtrlIdx [in]	Identifier of the Ethernet controller
CfgIdx [in]	Identifier of the configuration (only 0 allowed)
Return code	
Std_ReturnType	E_NOT_OK - Initialization of Ethernet controller failed
Std_ReturnType	E_OK - Ethernet controller initialized
Functional Description	
Initializes a Ethernet controller.	
Particularities and Limitations	
Module is initialized Function initializes a Ethernet controller and the related variables so it is possible to set it in operation afterwards.	
Call context	
> TASK > This function is Synchronous > This function is Reentrant	

Table 5-7 Eth_30_Core_ControllerInit

1.1.4 Eth_30_Core_SetControllerMode

Prototype	
`Std_ReturnType **Eth_30_Core_SetControllerMode** (uint8 CtrlIdx, Eth_ModeType CtrlMode)`	
Parameter	
CtrlIdx [in]	Identifier of the Ethernet controller

CtrlMode [in]	Operation mode that shall be applied: ETH_MODE_DOWN - Ethernet controller shall be turned off ETH_MODE_ACTIVE - Ethernet controller shall be tunred on
Return code	
Std_ReturnType	E_NOT_OK - Operation mode couldn't be applied
Std_ReturnType	E_OK - Operation mode succesfully applied
Functional Description	
Sets the operation mode of a Ethernet controller.	
Particularities and Limitations	
Ethernet controller is initialized Function sets the operation mode of the Ethernet controller so it is either turned off (no frame reception and transmission) or turned on (frames can be transmitted and received).	
Call context	
> TASK > This function is Synchronous > This function is Reentrant	

Table 5-8 Eth_30_Core_SetControllerMode

1.1.5 Eth_30_Core_GetControllerMode

Prototype	
`Std_ReturnType` **`Eth_30_Core_GetControllerMode`** `(uint8 CtrlIdx, Eth_ModeType *CtrlModePtr)`	
Parameter	
CtrlIdx [in]	Identifier of the Ethernet controller
CtrlModePtr [out]	Operation mode retrieved
Return code	
Std_ReturnType	E_NOT_OK - Retrieval of operation mode failed
Std_ReturnType	E_OK - Operation mode successfuly retrieved
Functional Description	
Retrieves the current operation mode of a Ethernet controller.	
Particularities and Limitations	
Ethernet controller is initialized -	
Call context	
> TASK > This function is Synchronous > This function is Reentrant	

Table 5-9 Eth_30_Core_GetControllerMode

1.1.6 Eth_30_Core_GetPhysAddr

Prototype	
`void Eth_30_Core_GetPhysAddr (uint8 CtrlIdx, uint8 *PhysAddrPtr)`	
Parameter	
CtrlIdx [in]	Identifier of the Ethernet controller
PhysAddrPtr [out]	Buffer of at least 6 byte to pass the MAC address
Return code	
void	none
Functional Description	
Retrieves the currently active MAC address of a Ethernet controller.	
Particularities and Limitations	
Module is initialized -	
Call context	
> TASK > This function is Synchronous > This function is Reentrant	

Table 5-10 Eth_30_Core_GetPhysAddr

1.1.7 Eth_30_Core_SetPhysAddr

Prototype	
`void Eth_30_Core_SetPhysAddr (uint8 CtrlIdx, const uint8 *PhysAddrPtr)`	
Parameter	
CtrlIdx [in]	Identifier of the Ethernet controller
PhysAddrPtr [in]	Buffer holding the MAC address that shall be applied
Return code	
void	none
Functional Description	
Sets the MAC address of a Ethernet controller.	
Particularities and Limitations	
Module is initialized Function sets the MAC address of a Ethernet controller. Dependent on the configuration of the "Write MAC address" feature the change is persisted in non-volatile RAM and also available after a power- cycle of the MCU.	
Call context	
> TASK	

> This function is Synchronous > This function is Reentrant

Table 5-11 Eth_30_Core_SetPhysAddr

1.1.8 Eth_30_Core_UpdatePhysAddrFilter

Prototype	
`Std_ReturnType` **`Eth_30_Core_UpdatePhysAddrFilter`** `(uint8 CtrlIdx, const uint8 *PhysAddrPtr, Eth_FilterActionType Action)`	
Parameter	
CtrlIdx [in]	Identifier of the Ethernet controller
PhysAddrPtr [in]	Buffer holding the MAC address the filter shall be adapted for
Eth_FilterActionType [in]	Action that shall be applied for the filter: ETH_REMOVE_FROM_FILTER - MAC address shall be blocked ETH_ADD_TO_FILTER - MAC address shall be allowed
Return code	
Std_ReturnType	E_NOT_OK - Filter modification failed
Std_ReturnType	E_OK - Filter successfully updated
Functional Description	
Updates the reception MAC address filter of a Ethernet controller.	
Particularities and Limitations	
Ethernet controller is initialized Functions allows to add or remove MAC address from the reception filter of the Ethernet controller so Ethernet frames addressed to the respective MAC address can be received or will be blocked from reception.	
Call context	
> TASK > This function is Synchronous > This function is Reentrant	

Table 5-12 Eth_30_Core_UpdatePhysAddrFilter

1.1.9 Eth_30_Core_WriteMii

Prototype	
`Eth_ReturnType` **`Eth_30_Core_WriteMii`** `(uint8 CtrlIdx, uint8 TrcvIdx, uint8 RegIdx, uint16 RegVal)`	
Parameter	
CtrlIdx [in]	Identifier of the Ethernet controller
TrcvIdx [in]	Address of the counter part on MDIO interface (MII address)
RegIdx [in]	Address of the register that shall be written

RegVal [in]	Value that shall be written to the register	
Return code		
Eth_ReturnType	ETH_E_NOT_OK - Service call failed due to invalid module state or function parameters	
	ETH_E_NO_ACCESS - Access to the MDIO interface timed out	
	ETH_OK - Write command was triggered succesfully	
Functional Description		
Triggers a write command on the MDIO interface of a Ethernet controller.		
Particularities and Limitations		
Ethernet controller is operational in mode ACTIVE Function triggers a write command on the MDIO interface according to clause 22 of the IEEE specification.		
Call context		
> ANY > This function is Synchronous > This function is Reentrant		

Table 5-13 Eth_30_Core_WriteMii

1.1.10 Eth_30_Core_ReadMii

Prototype	
`Eth_ReturnType` **`Eth_30_Core_ReadMii`** `(uint8 CtrlIdx, uint8 TrcvIdx, uint8 RegIdx, uint16 *RegValPtr)`	
Parameter	
CtrlIdx [in]	Idnetifier of the Ethernet controller
TrcvIdx [in]	Address of the counter part on MDIO interface (MII address)
RegIdx [in]	Address of the register that shall be read
RegValPtr [out]	Buffer to store the result of the read command
Return code	
Eth_ReturnType	ETH_E_NOT_OK - Service call failed due to invalid module state or function paramters
	ETH_E_NO_ACCESS - Access to the MDIO interface timed out
	ETH_OK -
Functional Description	
Triggers a read command on the MDIO interface of a Ethernet controller an provides the result.	
Particularities and Limitations	
Ethernet controller is operational in mode ACTIVE Function triggers a read command on the MDIO interface according to clause 22 of the IEEE specification and provides the result of th read.	
Call context	

> ANY > This function is Synchronous > This function is Reentrant

Table 5-14 Eth_30_Core_ReadMii

1.1.11 Eth_30_Core_GetCounterState

Prototype	
`Std_ReturnType` **`Eth_30_Core_GetCounterState`** `(uint8 CtrlIdx, uint16 CtrOffs, uint32 *CtrValPtr)`	
Parameter	
CtrlIdx [in]	Identifier of the Ethernet controller
CtrOffs [in]	Offset of the counter into the Ethernet statistics counter register space
CtrValPtr [out]	Buffer to store the counter value
Return code	
Std_ReturnType	E_NOT_OK - Counter retrieval failed E_OK - Counter successfully retrieved
Functional Description	
Retrieves the value of a Ethernet statistics counter.	
Particularities and Limitations	
Ethernet controller is initialized Function retrieves the value of a Ethernet statistics counter by addressing the counter with the help of a offset into the Ethernet statistics counter register space.	
Call context	
> ANY > This function is Reentrant	

Table 5-15 Eth_30_Core_GetCounterState

1.1.12 Eth_30_Core_ProvideTxBuffer

Prototype	
`BufReq_ReturnType` **`Eth_30_Core_ProvideTxBuffer`** `(uint8 CtrlIdx, uint8 *BufIdxPtr, Eth_DataType **BufPtr, uint16 *LenBytePtr)`	
Parameter	
CtrlIdx [in]	Identifier of the Ethernet controller
BufIdxPtr [out]	Identifier of the buffer provided on successful buffer provision
BufPtr [out]	Pointer to the buffer provided on successful buffer provision
LenBytePtr [out]	Buffer used to determin the requested and the provide length of the buffer: [in] Length of the data the caller wants to transmit (Payload length) [out] Actual length of the buffer provided

Return code	
BufReq_ReturnType	BUFREQ_E_NOT_OK - Service was called
	BUFREQ_E_OVFL - No buffer with the requested length available by configuration
	BUFREQ_E_BUSY - Any buffer able to hold the requested length is already in use
	BUFREQ_OK - Buffer successfully provided
Functional Description	
Provides a buffer that can be used to transmit a Ethernet frame.	
Particularities and Limitations	
Ethernet controller is operational in mode ACTIVE Function provides a buffer that can be used to transmit a Ethernet frame. The buffer is locked and therefore protected against reuse until the transmission of the frame is confirmed after transmission was triggered (Eth_Transmit() and consecutive Eth_TxConfirmation()) or buffer is intentionally released by calling Eth_Transmit() with LenByte=0.	
Call context	
> ANY > This function is Synchronous > This function is Reentrant	

Table 5-16 Eth_30_Core_ProvideTxBuffer

1.1.13 Eth_30_Core_Transmit

Prototype	
`Std_ReturnType` **`Eth_30_Core_Transmit`** `(uint8 CtrlIdx, uint8 BufIdx, Eth_FrameType FrameType, boolean TxConfirmation, uint16 LenByte, const uint8 *PhysAddrPtr)`	
Parameter	
CtrlIdx [in]	Identifier of the Ethernet controller
BufIdx [in]	Identifier of the buffer provided by Eth_ProvideTxBuffer()
FrameType [in]	Ethernet type, according to type field of IEEE802.3
TxConfirmation [in]	Request for a transmission confirmation: FALSE - No transmission confirmation desired TRUE - Transmission confirmation desired
LenByte [in]	Length of the data to be transmitted (Payload length)
PhysAddrPtr [in]	MAC address the frame shall be sent to
Return code	
Std_ReturnType	E_NOT_OK - Triggering of frame transmission wasn't possible
Std_ReturnType	E_OK - Frame transmission triggered
Functional Description	
Trigger the transmission of an Ethernet frame created from the passed buffer.	

Particularities and Limitations
> Ethernet controller is operational in mode ACTIVEBuffer was acquired by Eth_ProvideTxBuffer() Function takes the buffer previously provided by Eth_ProvideTxBuffer() enhances it with the Ethernet header and triggers the transmission of the Ethernet frame.
Call context
> ANY > This function is Synchronous > This function is Reentrant

Table 5-17 Eth_30_Core_Transmit

1.1.14 Eth_30_Core_Receive

Prototype	
`void Eth_30_Core_Receive (uint8 CtrlIdx, Eth_RxStatusType *RxStatusPtr)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
Triggers the reception of an Ethernet frame.	
Particularities and Limitations	
Call context	
FunctionCallcontext	

Table 5-18 Eth_30_Core_Receive

1.1.15 Eth_30_Core_VTransmit

Prototype	
`Std_ReturnType Eth_30_Core_VTransmit (uint8 CtrlIdx, uint8 BufIdx, Eth_FrameType FrameType, boolean TxConfirmation, uint16 LenByte, const uint8 *PhysAddrPtrDst, const uint8 *PhysAddrPtrSrc)`	
Parameter	
CtrlIdx [in]	Identifier of the Ethernet controller
BufIdx [in]	Identifier of the buffer provided by Eth_ProvideTxBuffer()
FrameType [in]	Ethernet type, according to type field of IEEE802.3
TxConfirmation [in]	Request for a transmission confirmation: FALSE - No transmission confirmation desired TRUE - Transmission confirmation desired
LenByte [in]	Length of the data to be transmitted (Payload length)

PhysAddrPtrDst [in]	Destination MAC address
PhysAddrPtrSrc [in]	Source MAC address
Return code	
Std_ReturnType	E_NOT_OK - Triggering of frame transmission wasn't possible
Std_ReturnType	E_OK - Frame transmission triggered
Functional Description	
Trigger the transmission of an Ethernet frame created from the passed buffer with a specifc source MAC.	
Particularities and Limitations	
> Ethernet controller is operational in mode ACTIVEBuffer was acquired by Eth_ProvideTxBuffer() Function takes the buffer previously provided by Eth_ProvideTxBuffer() enhances it with the Ethernet header (using a specifc source MAC address instead of the Ethernet controllers one) and triggers the transmission of the Ethernet frame.	
Call context	
> ANY > This function is Synchronous > This function is Reentrant	

Table 5-19 Eth_30_Core_VTransmit

1.1.16 Eth_30_Core_TxConfirmation

Prototype	
`void Eth_30_Core_TxConfirmation (uint8 CtrlIdx)`	
Parameter	
CtrlIdx [in]	Identifier of the Ethernet controller
Return code	
void	none
Functional Description	
Triggers the transmission confirmation of a previously Ethernet frame transmitted.	
Particularities and Limitations	
Ethernet controller is operational in mode ACTIVE Function triggers the transmission confirmation of a previously Ethernet frame transmitted and unlocks the buffer associated to the Ethernet frame so it is able to be used for frame transmission again.	
Call context	
> ANY > This function is Synchronous > This function is Reentrant	

Table 5-20 Eth_30_Core_TxConfirmation

1.1.17 Eth_30_Core_GetVersionInfo

Prototype	
`void Eth_30_Core_GetVersionInfo (Std_VersionInfoType *VersionInfoPtr)`	
Parameter	
VersionInfoPtr [out]	Buffer to store the version information
Return code	
void	none
Functional Description	
Retrieves the version information of the component.	
Particularities and Limitations	
- Function retrieves the Vendor ID, Module ID and software version of the component.	
Call context	
> ANY > This function is Synchronous > This function is Reentrant	

Table 5-21 Eth_30_Core_GetVersionInfo

1.1.18 Eth_30_Core_GetRxStats

Prototype	
`Std_ReturnType Eth_30_Core_GetRxStats (uint8 CtrlIdx, Eth_RxStatsType *RxStats)`	
Parameter	
CtrlIdx [in]	Identifier of the Ethernet controller
RxStats [out]	List of values to read statistic values for transmission
Return code	
Std_ReturnType	E_OK: success E_NOT_OK: Rx-statistics could not be obtained
Functional Description	
Returns list of Reception Statistics.	
Particularities and Limitations	
Function returns the list of Reception Statistics out of IETF RFC1213. Configuration Variant(s): ETH_30_CORE_ENABLE_GET_ETHER_STATS_API	
Call context	
> ANY > This function is Synchronous > This function is Reentrant	

Table 5-22 Eth_30_Core_GetRxStats

1.1.19 Eth_30_Core_GetTxStats

Prototype	
`Std_ReturnType Eth_30_Core_GetTxStats (uint8 CtrlIdx, Eth_TxStatsType *TxStats)`	
Parameter	
CtrlIdx [in]	Identifier of the Ethernet controller
TxStats [out]	List of values to read statistic values for transmission
Return code	
Std_ReturnType	E_OK: success E_NOT_OK: Tx-statistics could not be obtained
Functional Description	
Returns list of Transmission Statistics.	
Particularities and Limitations	
Function returns the list of Transmission Statistics out of IETF RFC1213. Configuration Variant(s): ETH_30_CORE_ENABLE_GET_ETHER_STATS_API	
Call context	
> ANY > This function is Synchronous > This function is Reentrant	

Table 5-23 Eth_30_Core_GetTxStats

5.3.2 Time Synchronization API

5.3.2.1 Eth_30_Core_GetGlobalTime

Prototype	
`Std_ReturnType Eth_30_Core_GetGlobalTime (uint8 CtrlIdx, Eth_TimeStampType *TimestampPtr, Eth_TimestampQualityType *TimestampQualityPtr)`	
Parameter	
CtrlIdx [in]	Identifier of the Eth controller
TimestampPtr [out]	Retrieve time stamp
TimestampQualityPtr [out]	Quality of the time stamp ETH_TIMESTAMP_VALID: Time stamp is valid ETH_TIMESTAMP_INVALID: Time stamp is not valid ETH_TIMESTAMP_UNCERTAIN: Time stamp is uncertain
Return code	
Std_ReturnType	E_OK - Time stamp successfully retrieved
Std_ReturnType	E_NOT_OK - Time stamp couldn't be retrieved

Functional Description
Retrieves the current time of the Eth controllers timer.
Particularities and Limitations
Ethernet controller is operational in mode ACTIVE - Configuration Variant(s): ETH_30_CORE_TIME_SYNC_ENABLED
Call context
> ANY > This function is Synchronous > This function is Reentrant

Table 5-24 Eth_30_Core_GetGlobalTime

5.3.2.2 Eth_30_Core_SetGlobalTime

Prototype	
`Std_ReturnType` **`Eth_30_Core_SetGlobalTime`** `(uint8 CtrlIdx, const Eth_TimeStampType *TimestampPtr)`	
Parameter	
CtrlIdx [in]	Identifier of the Eth controller
TimestampPtr [in]	Time the Eth controllers timer shall be set to
Return code	
Std_ReturnType	E_OK - Timer successfully set
Std_ReturnType	E_NOT_OK - Timer couldn't be set
Functional Description	
Sets the timer of the Eth controller to the given time.	
Particularities and Limitations	
Ethernet controller is operational in mode ACTIVE - Configuration Variant(s): ETH_30_CORE_TIME_SYNC_ENABLED	
Call context	
> ANY > This function is Synchronous > This function is Reentrant	

Table 5-25 Eth_30_Core_SetGlobalTime

5.3.2.3 Eth_30_Core_SetCorrectionTime

Prototype	
`Std_ReturnType **Eth_30_Core_SetCorrectionTime** (uint8 CtrlIdx, const Eth_TimediffType *OffsetTimePtr, const Eth_RateRatioType *RateRatioPtr)`	
Parameter	
CtrlIdx [in]	Identifier of the Eth controller
OffsetTimePtr [in]	Offset that shall be corrected in nanoseconds 0: No offset correction shall be done other values: positive/negative offset correction shall be done
RateRatioPtr [in]	Rate correction to compensate the drift of the timer 1.0: No rate correction shall be done value in [0.0, 2.0] without 1.0: Correct drift by given value
Return code	
Std_ReturnType	E_OK - Correction successfully applied
Std_ReturnType	> E_NOT_OK - Correction couldn't be applied due to > - Wrong value given for offset and/or rate correction > - Corrections couldn't be applied due to hardware limitations
Functional Description	
Corrects the Eth controllers timer by the given offset and/or rate correction.	
Particularities and Limitations	
Ethernet controller is operational in mode ACTIVE - Configuration Variant(s): ETH_30_CORE_TIME_SYNC_ENABLED	
Call context	
> ANY > This function is Synchronous > This function is Reentrant	

Table 5-26 Eth_30_Core_SetCorrectionTime

5.3.2.4 Eth_30_Core_EnableEgressTimestamp

Prototype	
`Std_ReturnType **Eth_30_Core_EnableEgressTimestamp** (uint8 CtrlIdx, uint8 BufIdx)`	
Parameter	
CtrlIdx [in]	Identifier of the Eth controller
BufIdx [in]	Identifier of the buffer holding the frame to be transmitted
Return code	
Std_ReturnType	E_OK - Time stamping of frame successfully enabled
Std_ReturnType	E_NOT_OK - Time stamping of the frame wasn't enabled
Functional Description	
Enables time stamping for the given frame on transmission.	

Particularities and Limitations
> Ethernet controller is operational in mode ACTIVEBuffer was previously allocated by Eth_30_Core_ProvideTxBuffer() and no transmission was triggerd by Eth_30_Core_Transmit() yet - Configuration Variant(s): ETH_30_CORE_TIME_SYNC_ENABLED
Call context
> ANY > This function is Synchronous > This function is Reentrant

Table 5-27 Eth_30_Core_EnableEgressTimestamp

5.3.2.5 Eth_30_Core_GetEgressTimestamp

Prototype	
`Std_ReturnType` **`Eth_30_Core_GetEgressTimestamp`** `(uint8 CtrlIdx, uint8 BufIdx, Eth_TimeStampType *TimestampPtr, Eth_TimestampQualityType *TimestampQualityPtr)`	
Parameter	
CtrlIdx [in]	Identifier of the Eth controller
BufIdx [in]	Identifier of the buffer used to transmit the frame
TimestampPtr [out]	Time stamp taken on transmission
TimestampQualityPtr [out]	Quality of the time stamp ETH_TIMESTAMP_VALID: Time stamp is valid ETH_TIMESTAMP_INVALID: Time stamp is not valid ETH_TIMESTAMP_UNCERTAIN: Time stamp is uncertain
Return code	
Std_ReturnType	E_OK - Time samp successfully retrieved
Std_ReturnType	E_NOT_OK - Time stamp couldn't be retrieved
Functional Description	
Retrieves the time stamp for a frame transmitted.	
Particularities and Limitations	
Ethernet controller is operational in mode ACTIVE - Configuration Variant(s): ETH_30_CORE_TIME_SYNC_ENABLED	
Call context	
> <Eth_UL>_TxConfirmation() > This function is Synchronous > This function is Reentrant	

Table 5-28 Eth_30_Core_GetEgressTimestamp

5.3.2.6 Eth_30_Core_GetIngressTimestamp

Prototype	
`Std_ReturnType` **`Eth_30_Core_GetIngressTimestamp`** `(uint8 CtrlIdx, Eth_DataType *DataPtr, Eth_TimeStampType *TimestampPtr, Eth_TimestampQualityType *TimestampQualityPtr)`	
Parameter	
CtrlIdx [in]	Identifier of the Eth controller
DataPtr [in]	Memory space the frame is located (as given in <Eth_UL>_RxIndication)
TimestampPtr [out]	Time stamp taken on reception
TimestampQualityPtr [out]	Quality of the time stamp ETH_TIMESTAMP_VALID: Time stamp is valid ETH_TIMESTAMP_INVALID: Time stamp is not valid ETH_TIMESTAMP_UNCERTAIN: Time stamp is uncertain
Return code	
Std_ReturnType	E_OK - Time samp successfully retrieved
Std_ReturnType	E_NOT_OK - Time stamp couldn't be retrieved
Functional Description	
Retrieves the time stamp for a frame received.	
Particularities and Limitations	
Ethernet controller is operational in mode ACTIVE - Configuration Variant(s): ETH_30_CORE_TIME_SYNC_ENABLED	
Call context	
> <Eth_UL>_RxIndication() > This function is Synchronous > This function is Reentrant	

Table 5-29 Eth_30_Core_GetIngressTimestamp

5.3.3 FQTSS (Traffic Shaping) API

5.3.3.1 Eth_30_Core_SetBandwidthLimit

Prototype	
`Std_ReturnType` **`Eth_30_Core_SetBandwidthLimit`** `(uint8 CtrlIdx, uint8 QueuePrio, uint32 BandwidthLimit)`	
Parameter	
CtrlIdx [in]	Controller Index
QueuePrio [in]	Queue Priority
BandwidthLimit [in]	Bandwidth limit which shall be assigned for the Tx queue (in [bits/s])

Return code	
Std_ReturnType	E_NOT_OK - New bandwidth limit couldn't be set.
Std_ReturnType	E_OK - New bandwidth limit set.
Functional Description	
Reconfigures the bandwidth limit set for a transmission queue.	
Particularities and Limitations	
Ethernet controller is operational in mode ACTIVE -	
Call context	
> ANY > This function is Synchronous > This function is Reentrant	

Table 5-30 Eth_30_Core_SetBandwidthLimit

5.3.3.2 Eth_30_Core_GetBandwidthLimit

Prototype	
`Std_ReturnType` **`Eth_30_Core_GetBandwidthLimit`** `(uint8 CtrlIdx, uint8 QueuePrio, uint32 *BandwidthLimitPtr)`	
Parameter	
CtrlIdx [in]	Controller Index
QueuePrio [in]	Queue Priority
BandwidthLimitPtr [out]	Pointer to where to store the currently configured bandwidth limit (in [bit/s])
Return code	
Std_ReturnType	E_NOT_OK - Current bandwidth limit couldn't be retrieved.
Std_ReturnType	E_OK - Current bandwidth limit retrieved.
Functional Description	
Retrieves the currently configured bandwidth limit of a transmission queue.	
Particularities and Limitations	
Ethernet controller is operational in mode ACTIVE -	
Call context	
> ANY > This function is Synchronous > This function is Reentrant	

Table 5-31 Eth_30_Core_GetBandwidthLimit

5.4 Services used by Ethernet Driver

In the following table services provided by other components, which are used by the Ethernet Driver are listed. For details about prototype and functionality refer to the documentation of the providing component.

Component	API
Det	`Det_ReportError()`
Dem	`Dem_SetEventStatus()`
EthIf	▶ `EthIf_RxIndication()` ▶ `EthIf_TxConfirmation()`
EthSwt	▶ `EthSwt_EthAdaptBufferLength()` ▶ `EthSwt_EthPrepareTxFrame()` ▶ `EthSwt_EthProcessTxFrame()` ▶ `EthSwt_EthTxFinishedIndication()` ▶ `EthSwt_EthProcessRxFrame()` ▶ `EthSwt_EthRxFinishedIndication()`
NvM (optional)	▶ `NvM_GetErrorStatus()` ▶ `NvM_SetRamBlockStatus()`
Os (optional)	▶ `osReadPeripeheral32()` ▶ `osWritePeripheral32()` ▶ `osModifyPeripheral32()`

Table 5-32 Services used by the Ethernet Driver

5.5 Callback Functions

The Ethernet Driver does not provide callback functions.

6 Configuration

The Ethernet Driver is configured with the help of the following Vector tools:

> DaVinci Configurator PRO

6.1 Configuration Variants

The Ethernet Driver supports the configuration variants

> `VARIANT-PRE-COMPILE`

The configuration classes of the Ethernet Driver parameters depend on the supported configuration variants. For their definitions please see the Eth_30_<Driver>_bswmd.arxml file.

6.2 Driver specific configuration

For configuration aspects of the specific driver please see [6].

7 Glossary and Abbreviations

7.1 Glossary

Term	Description
DaVinci Configurator PRO	Generation tool for MICROSAR components

Table 7-1 Glossary

7.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
DEM	Diagnostic Event Manager
DET	Development Error Tracer
ECU	Electronic Control Unit
FCS	Frame Check Sequence
FQTSS	Forwarding and Queuing of Time Sensitive Streams
HIS	Hersteller Initiative Software
ISR	Interrupt Service Routine
MAC	Media Access Control
MCU	Microcontroller Unit
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
MII	Media Independent Interface
QoS	Quality of Service
RTE	Runtime Environment
SRS	Software Requirement Specification
SWC	Software Component
SWS	Software Specification

Table 7-2 Abbreviations

8 Contact

Visit our website for more information on

> News
> Products
> Demo software
> Support
> Training data
> Addresses

www.vector.com