National Semiconductor Ethernet PHYTER®

Software Development Guide

Revision 1.97

September 13, 2011

Revision History

Release	Date	Who	Revisions
1.00	03-31-06	Drex C. Dixon	Initial release for DP83848 & DP83849
1.10	05-04-07	Drex C. Dixon	Added support for DP83640 device
1.20	09-03-07	Drex C. Dixon	Added section on 1588 clock tuning. Added section showing a simple 1588 configuration example.
1.30	11-09-07	Drex C. Dixon Devin Seely	Added PTPEnable function. Added OAIBegin/EndMultiCriticalSection functions. Clarified multi-threaded support. Added details to hardware overview sections.
1.40	11-16-07	Devin Seely	Added details to PHY Control Management Interface section.
1.41	12-18-07	Devin Seely	Added detail to Sec 4.1.3.4. Corrected Table 4.1-2.
1.50	02-22-08	Devin Seely	Removed Sections ALP for PHYTER, Ethernet PHYTER Python Library and Active X Interfaces, replaced phy with PHY, updated sections 3.1.2, 3.1.3,
1.60	05-14-08	Devin Seely	Added/modified text in Sec. 3.1.2.3, corrected title of Sec. 4.8.1
1.7	06-11-08	Devin Seely	Added detail to Sec. 3.1.4.6
1.71	08-14-08	Todd Roberts	Updated IsPhyStatusFrame and GetNextPhyMessage to match code. Clarify PTPSetPhyStatusFrameConfig
1.80	08-26-08	Todd Roberts	Update based on testing of API and updates based on actual code implementation.
1.81	09-05-08	Todd Roberts	Updated section 4 to improve user understanding of the library.
1.90	09-23-08	Todd Roberts	Additional updates after PTP operation test development.
1.91	09-30-08	Devin Seely	Corrections to Table 3.1-1 and Section 3.1.7
1.92	10-02-08	Todd Roberts	Corrections to sections 4.4.4.1, 4.4.5.1, 4.4.5.2, and 4.4.7.1 to clarify what adjustments made to timestamp values.
1.93	10-31-08	Todd Roberts	Updated PTPSetEventConfig and PTPSetTransmitConfig sections.
1.94	03-26-09	Ben Buchanan	
1.95	05-29-09	Patrick O’Farrell	Corrected PTPClockStepAdjustment to indicate necessary caller adjustment.
1.96	06-11-09	Patrick O’Farrell	Add notes to section 3.1.2.1 and section 3.1.3.3 clarifying recommended BYTE0_MASK and BYTE0_DATA settings for PTPv2.
1.97	09-13-11	Patrick O’Farrell	Add support for DP83630 and DP83620. Removed references to engineering samples. Corrected ns magnitude in section 3.1.1.2.3. Updated sections on Transmit (3.1.2.1, 3.1.2.2) and Receive (3.1.3.3) Timestamp Configuration. Corrected Type Value bits in section 3.1.8.4. Removed docs directory from EPL code structure.

List of Tables

1 Introduction

The National Semiconductor Ethernet PHYTER® software development guide is provided to enable customers to easily integrate the DP83848, DP83849 and DP836x0 products into their applications and systems. This document describes the software tools and collateral available with the PHYTER family of products. It also details specific libraries and interfaces that can be used to access the powerful hardware capabilities and performance features of each product.

1.1 Overview of Available Tools

A number of tools have been developed to aid in PHYTER silicon feature demonstration, system integration and troubleshooting. This section provides a brief overview of each software tool.

1.1.1 Analog LaunchPAD (ALP)

National has developed a powerful, feature-rich FPGA based platform for demonstrating National's silicon products. It is called the Analog LaunchPAD (ALP). In its initial release the platform uses a Xilinx Spartan 3E device with four general purpose expansion headers. Silicon products from National are provided on small demonstration boards that then plug into one or more ALP expansion headers. A mock-up of the ALP baseboard is shown below.

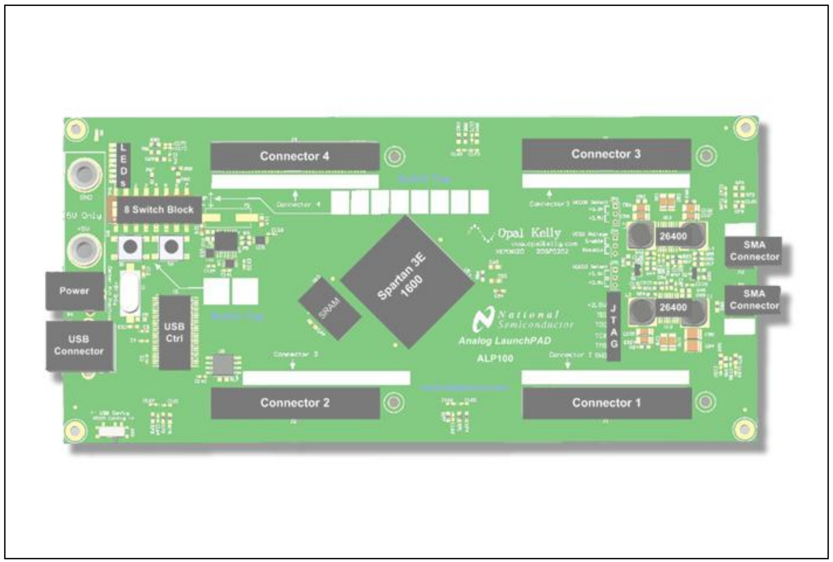
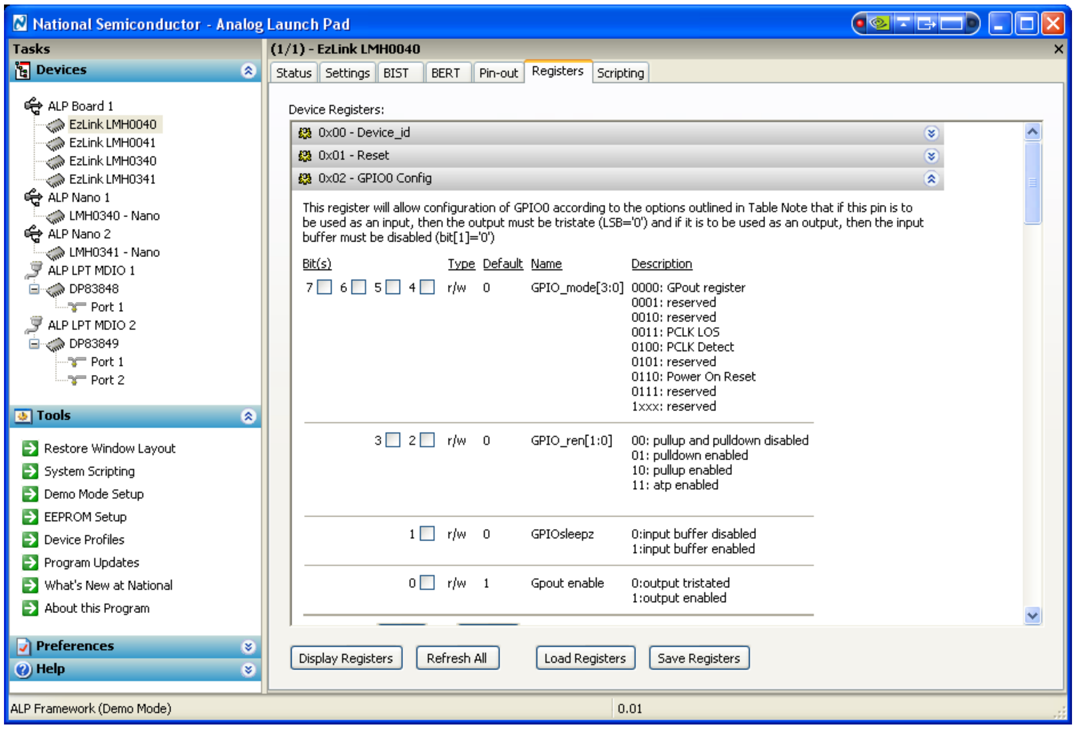


Figure 1 – ALP Baseboard

As a lower cost alternative, ALP "Nano" boards are available that provide management interface access to the PHYTER over a parallel port or USB connection. These adapters do not contain an FPGA.

A GUI application called the ALP Framework (ALPF) provides the user interface needed to interact with the various silicon demonstration boards. An example screen shot of the application is shown below.



Device Registers:

0x00 - Device_id

0x01 - Reset

0x02 - GPIO0 Config

This register will allow configuration of GPIO0 according to the options outlined in Table Note that if this pin is to be used as an input, then the output must be tristate (LSB='0') and if it is to be used as an output, then the input buffer must be disabled (bit[1]='0')

Bit(s)	Type	Default	Name	Description
7 6 5 4	r/w	0	GPIO_mode[3:0]	0000: GPout register 0001: reserved 0010: reserved 0011: PCLK LOS 0100: PCLK Detect 0101: reserved 0110: Power On Reset 0111: reserved 1xxx: reserved
3 2	r/w	0	GPIO_ren[1:0]	00: pullup and pulldown disabled 01: pulldown enabled 10: pullup enabled 11: atp enabled
1	r/w	0	GPIOsleepz	0:input buffer disabled 1:input buffer enabled
0	r/w	1	Gpout enable	0:output tristated 1:output enabled

Figure 2 – ALP Framework Application

1.1.2 Ethernet PHYTER "C" Library (EPL)

A platform independent device software "C" library is available that simplifies PHYTER integration with any target software platform. The library eliminates, in most cases, the need to directly interact with device registers, although direct register access is provided.

The documentation for EPL is provided in Chapter 4.

1.1.3 Ethernet PHYTER Python Library

As part of ALP for PHYTER, a device library is provided that allows one to interact with a PHYTER device using the popular and simple "Python" dynamic scripting language. This is typically used for device feature demonstration and troubleshooting.

1.1.4 ActiveX Interfaces

All of the methods provided by the ALP PHYTER Python Library are exposed through Windows ActiveX. This allows one to use many popular tools, such as Visual Basic for Applications (VBA), Excel, Agilent's VEEPro and National Instrument's LabView to interact with any PHYTER device. In fact, any ActiveX enabled environment can successfully interact with an attached PHYTER device.

2 Device Management Interfaces

Control, status and data access to all PHYTER products is register based and occurs through a device management interface. Depending on the PHYTER device model, up to two hardware management interfaces are supported. The first is the standard IEEE MDIO protocol and the second is exclusive to National and provides high-speed access through the device's MII interface.

2.1 MDIO

For all PHYTER devices, registers can be accessed using standard IEEE MDIO, which is a simple two-wire bit banging protocol. The PHYTER device abstraction software libraries provide support for using MDIO for device access.

2.2 MII PHY Control Management Interface

The PHYTER High Precision devices (DP83640 and DP83630) support a packet-based control mechanism for use in situations where the Serial Management Interface (MDIO) is not available or does not provide enough throughput.

Application software may build a packet, called a PHY Control Frame (PCF), to be passed to the PHY through the MAC Transmit Data interface. The PHY will intercept these packets and use them to assert writes to Management Registers as if they occurred via the usual Management Interface. Multiple register writes may be incorporated in a single frame.

A PCF may also be used to read a register. The read value will be returned in a PHY Status Frame (PSF). Only a single read may be outstanding at a time, therefore only one read should be included in a PCF request packet.

PCFs can be used to generate registers writes without any additional configuration if the device is properly strapped to accept them. PCF_EN/GPIO2 should be pulled high to enable PCFs by default. However PSFs must be properly configured before PCFs can be used to generate register reads.

The provided device abstraction software libraries support using PCF's for device access, however the library must have access to environment dependent MAC frame transmit and frame receive de-multiplexing subsystems.

2.2.1 Command Structure

Each PCF Command is equivalent to a Management access over the Serial Management Interface. As such, the Read or Write Command is similar in format, although it is extended to eliminate the need to change pages. The format for each 32-bit command is as follows:

Field	Bit Location	Description
Command Type	31:30	Set to 01 for Write Set to 10 for Read
PHY Address	29:25	Set to PHY Address or to 0x1F (broadcast)
Page Select	24:21	Register Page Select
Register Address	20:16	Register Address
Write Data	15:0	Write Data

Table 2.2-1 – Command Format

The command packets need to be created in network order (e.g. the MSB first). A command of all 0s indicates no action and is used to terminate a list of commands.

The PHY Address field should be set to the PHY Address of the device, or to a broadcast value of 0x1F. The device can be programmed to ignore the broadcast address by setting the PCF_BC_DIS bit in the PCFCR.

2.2.2 PHY Control Frame Format

Field	Length	Value	Description
Preamble	7 bytes	0x55	Ethernet preamble
SFD	1 byte	0x5D	Ethernet Start-of-Frame Delimiter
Destination Address	6 bytes	0x08 0x00 0x17 0x0B 0x6B 0x0F or 0x08 0x00 0x17 0x00 0x00 0x00	Destination Address. Uses a National Semiconductor assigned Mac address. Multicast versions of the addresses are also allowed. Address is configured by the PCF_DA_SEL bit in the PCFCR register.
Header (Optional)			Optional information to allow embedding within any type of packet
Start Field	6 bytes	0x5F, 0x50, 0x48, 0x59, 0x43, 0x46	Start Field is used to detect the beginning of a Write Command Sequence. The value is ASCII encoding of the string '_PHYCF'.
PCF Commands	N * 4 bytes		Individual commands, each 4 bytes in length.
Termination Field	4 bytes	0x00	
Pad (Optional)		0x00	Pad to 60 bytes for Destination Address through Pad.
CRC	4 bytes		Ethernet Frame Check Sequence

Table 2.2-2 – PHY Control Frame Format

The Preamble, SFD, and CRC fields are usually generated by the Mac layer.

2.2.3 Enabling PHY Control Frames

PHY Control Frames can be enabled through the PCF_Enable bit in the PHY Control Frames Configuration
Register (PCFCR). PHY Control Frames can also be enabled by using the PCF_EN strap option.

2.2.4 PHY Control Frame Interrupts

The PHY Control Frame function may be programmed to interrupt the system on either of two conditions: completion of a PHY Control Frame, or on a PHY Control Frame error.

The PHY Control Frame interrupts may be enabled by setting the PCF_INT_CTL bits in the PCFCR Register. Setting either of these bits will enable control of the PCF Interrupt through bit 0 of the MISR register. Interrupt status will be available at bit 8 at of the MISR. Note that this replaces the Receive Error Counter half-full Interrupt (RHF_INT) and causes the RFH_INT to be combined with the False Carrier Counter half-full Interrupt (FHF_INT).

The cause of the PCF Interrupt may be determined by checking the PCF_STS_OK and PCF_STS_ERR bits in the PCFCR register. Reading the PCFCR register is necessary to rearm the PCF interrupt.

A PCF_STS_ERR interrupt indicates an error was detected on a PHY Control Frame. Any writes in the control frame may have been completed, but may not have completed accurately. Error conditions may include any of the following:

- CRC Error for the frame
- Lack of Termination field
- Unknown command type

If an unknown command type is detected, any subsequent commands in the PHY Control Frame will be ignored.

2.2.5 Control Frame Buffer

Since the PHY must parse a portion of the Control Frame, it will normally send a portion of the frame to the wire while the frame is being parsed. If a Control Frame is detected, the PHY will truncate the frame immediately following the Destination Address. In most Ethernet environments this will be seen as a collision fragment and ignored by receiving Mac layers.

In certain non-standard implementations (such as protocols that rely on isochronous control of the medium), the truncated frame could cause issues. In this case, the PHY can be programmed to buffer up to 15 bytes of each packet. This allows the PHY to determine if the packet is a Control Frame before sending any of the packet on the wire. The size of the Control Frame Buffer is configurable through the PCF_BUF field in the PCFCR Register. Enabling the packet buffering will add delay in the transmit packet propagation through the PHY.

3 DP83640 and DP83630 PHYTER High Precision for IEEE 1588

This chapter provides an overview of the 1588 related features available with the DP83640 and DP83630 PHYTER High Precision devices, information on device configuration and concludes with tips and techniques that can be used to synchronize a slave clock with a 1588 master.

A separate section in the Ethernet PHYTER "C" Library (EPL) chapter provides documentation for the 1588 related software library calls that support these features. One should first become familiar with the hardware functionality presented here prior to reviewing and using the EPL software library functions.

3.1 Hardware Functional Overview

The PHYTER High Precision provides advanced and flexible support for IEEE 1588 for use in a highly accurate IEEE 1588 system. It provides a 1588 digital clock implementation, IEEE 1588 Transmit and Receive packet parsing and Start-of-Frame detection, Transmit and Receive Timestamp units, trigger generation, event timestamp unit and a Pulse-Per-Second (PPS) generator.

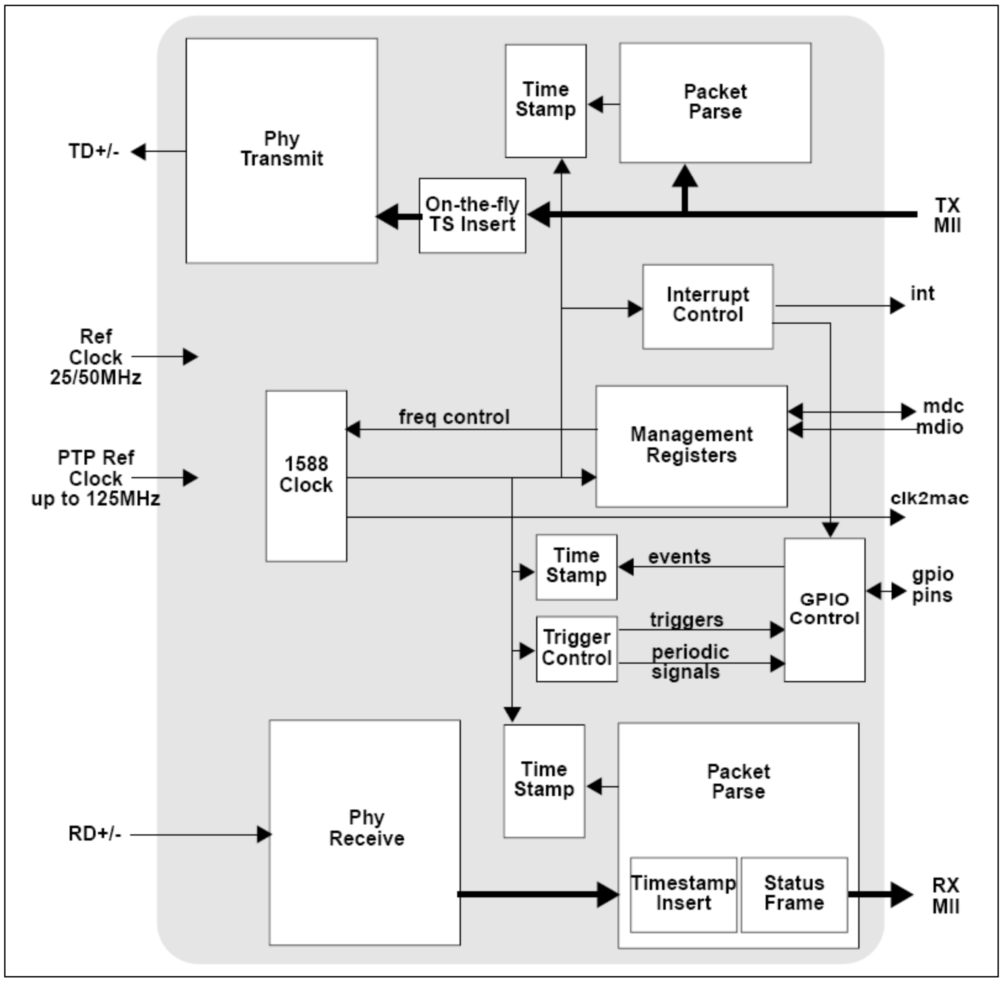


Figure 3 – PHYTER High Precision Hardware Block Diagram

3.1.1 IEEE 1588 Clock

The IEEE 1588 clock has the following features:

- Frequency scalable - Frequency (clock rate) may be adjusted to match the frequency of the master.
- Adjustable by add/subtract - The clock may be adjusted in a step fashion to jump to match the master clock.
- Directly read/writable - Initial setting of the clock may require a direct write of a time value.
- Temporary Frequency control - Allows time correction by running at a modified frequency for a period of time.
- 8 ns resolution (running at 125 MHz).

3.1.1.1 Controlling the IEEE 1588 Clock

The PHY provides features for controlling the clock operation in Slave mode. The clock value can be updated to match the Master clock in several ways. In addition, the clock can be programmed to adjust its frequency to compensate for drift.

The clock consists of several fields:

- Seconds: 32-bit field
- Nanoseconds: 30-bit field (maximum value is 10^9 ns)
- Fractional Nanoseconds: Units of 2^{-32} ns.

The clock does not support negative time values. If negative time is required in the system, software will have to make conversions from the PHY clock time to actual time.

The clock also does not support the upper 16-bits of the seconds field as defined by the specification (version 2 specifies a 48-bit seconds field). If this value is required to be greater than 0, it will have to be handled by software. Since a rollover of the second's field will only occur every 136 years, this should not be a significant burden to software.

3.1.1.2 Updating the Clock time value

Several mechanisms can be used to update the PHY's IEEE 1588 clock, based on the results of the synchronization protocol.

Step adjustment - A step adjustment value in nanoseconds may be added to the current value. Note that the adjustment value can be positive or negative.

Time set - A direct set of the time value can be done by setting a new time value.

Rate adjustment - The clock can be programmed to operate at an adjusted frequency value by programming a rate adjustment value. The rate adjustment allows for correction on the order of 2^{-32} ns per reference clock cycle. The frequency adjustment will allow the clock to correct the offset over time, avoiding any potential side-effects caused by a step adjustment in the time value.

Temporary Rate adjustment - The clock can be programmed to operate at a temporary adjusted frequency value by programming a rate adjustment value and duration. The rate adjustment allows for correction on the order of 2^{-32} ns per reference clock cycle. The frequency adjustment will allow the clock to correct the offset over time, avoiding any potential side-effects caused by a step adjustment in the time value.

The method used to update the clock value may depend on the difference in the values. For example, at the initial synchronization attempt, the clocks may be very far apart, and therefore require a Step adjustment or a direct Time set. Later, when clocks are very close in value, the Temporary Rate adjustment method may be the best option.

3.1.1.2.1 Setting Time via PTP Time Data Regiser

Setting time through the PTP Time Data Register involves writing all four time fields to the PTP_TDR and then issuing a PTP_Load_Clk command through the PTP Control Register. The Time value may also be read using the PTP_TDR register by first setting the PTP_Rd_Clk command in the PTP Control Register and then reading all four time fields. In each case, the order of fields is the same.

For example, to set the time:

```
Write Clock_time_ns[15:0] to PTP_TDR
Write Clock_time_ns[31:16] to PTP_TDR
Write Clock_time_sec[15:0] to PTP_TDR
Write Clock_time_sec[31:16] to PTP_TDR
Write to PTP_CTL with the PTP_Load_Clk bit set
```

To read the time:

```
Write to PTP_CTL with the PTP_Rd_Clk bit set
Read Clock_time_ns[15:0] from PTP_TDR
Read Clock_time_ns[31:16] from PTP_TDR
Read Clock_time_sec[15:0] from PTP_TDR
Read Clock_time_sec[31:16] from PTP_TDR
```

3.1.1.2.2 Clock Rate Control

The PTP Rate control registers allow setting two different Rate values, a Normal Rate and a Temporary Rate. The Temporary Rate allows the 1588 Clock to operate at a modified rate for a programmed amount of time (as controlled by the PTP Temporary Rate Duration Registers). The Normal Rate will be selected if a Temporary Rate is not currently active.

When setting a rate, the PTP_TMP_RATE bit in the PTP Rate High Register (PTP_RATEH) indicates the rate is to be temporary. The Temporary Rate will be applied to the clock for the duration set in the PTP_TRD registers. Following completion of the time duration set in PTP_TRD the rate will revert back to the Normal Rate. Note that the Normal Rate may be changed, through the PTP Rate registers, while a Temporary Rate is active. This will have no effect on the Temporary Rate, but the new Normal Rate will be used when the Temporary Rate Duration completes.

3.1.1.2.3 Making Time Corrections using the PTP Time Data Register

Making time adjustments to the PTP Clock through the PTP Time Data Register is similar to setting a time value. The process involves writing all four time fields to the PTP_TDR and then issuing a PTP_Step_Clk command through the PTP Control Register.

To add time using the Step Clock function, the value should be positive. To subtract, both seconds and nanoseconds fields should be 32-bit 2's complement representations. The addition process is a pipelined process that takes two clock cycles at 8 ns each at the default clock rate. When adjusting the clock, the value added should include 16 ns to compensate for the 2-cycle addition. For example, to adjust the clock by +100 ns, the actual value added should be 116 ns. To subtract 100 ns, the actual value subtracted should be 84 ns.

When adding or subtracting using the Step Clock function, the nanosecond value should be less than 10^9 in magnitude.

For example, to add/subtract the time:

```
Write Clock_time_ns[15:0] to PTP_TDR
Write Clock_time_ns[31:16] to PTP_TDR
Write Clock_time_sec[15:0] to PTP_TDR
Write Clock_time_sec[31:16] to PTP_TDR
Write to PTP_CTL with the PTP_Step_Clk bit set
```

3.1.1.2.4 Making Time Corrections by Setting a Temporary Rate

The PTP Rate control registers allow setting two different Rate values, a Normal Rate and a Temporary Rate. The Temporary Rate allows the 1588 Clock to operate at a modified rate for a programmed amount of time (as controlled by the PTP Temporary Rate Duration Registers). The Normal Rate will be selected if a Temporary Rate is not currently active.

When setting a rate, the PTP_TMP_RATE bit in the PTP Rate High Register (PTP_RATEH) indicates the rate is to be temporary. The Temporary Rate will be applied to the clock for the duration set in the PTP_TRD registers. Following completion of the time duration set in PTP_TRD the rate will revert back to the Normal Rate. Note that the Normal Rate may be changed, through the PTP Rate registers, while a Temporary Rate is active. This will have no effect on the Temporary Rate, but the new Normal Rate will be used when the Temporary Rate Duration completes.

To adjust the time value using the Rate control registers, software uses the 1588 protocol to determine the time correction required. The time correction may be spread over multiple clock cycles by programming a Temporary Rate value. To determine the rate setting, software should compute the rate difference as the time correction divided by the time duration in number of 8 ns clock cycles. This value should be multiplied by 2^{32} to convert to the correct units. This rate difference should then be added to the current PTP Rate setting to provide the Temporary Rate.

The Temporary Rate value is a 26-bit value plus a sign bit, providing a range of $-(2^{26}-1)$ to $+(2^{26}-1)$ in units of 2^{-32} ns/cycle. Since each reference clock cycle is 8 ns, this allows for a rate adjustment maximum of approximately +/-1950 ppm.

The Temporary Rate duration is a 26-bit value providing a duration of up to 536 ms.

Example:

Conditions:

Current Rate = +10 ppm, which gives PTP_Rate = 343597 (2^{-32} ns/cycle)
Time_Error = 20 ns, gives Time_Corr = -20 ns
Assume the correction will be done over 10 ms, gives
Temp_Rate_duration = 10 ms/8 ns = 1,250,000

Calculation:

Temp_Rate_delta = (Time_Corr/Temp_Rate_duration) * 2^{32}
= (-20/1250000) * 2^{32} = -68719
Temp_Rate = Current_Rate + Temp_Rate_delta = 343597 + -68719 = 274878

To set the Temporary Rate:

1. Write Temp_Rate_duration[25:16] to PTP Temporary Rate Duration High Register (PTP_TRDH)
2. Write Temp_Rate_duration[15:0] to PTP Temporary Rate Duration Low Register (PTP_TRDL)
3. Write Temp_Rate[25:16] to PTP Rate High Register (PTP_RATEH) with PTP_TMP_RATE bit set to 1.
4. Write Temp_Rate[15:0] to PTP Rate Low Register (PTP_RATEL)

The Temporary Rate will automatically start following the write to the PTP_RATEL register. When the Temporary Rate Duration has expired, the clock rate will revert to the normal PTP Rate. The Temporary Rate Duration registers do not need to be reprogrammed for each setting of the Temporary Rate if the duration is to remain unchanged.

3.1.2 IEEE 1588 Transmit Packet Parser and Timestamp unit

The IEEE 1588 transmit parser monitors transmit packet data to detect IEEE 1588 Version1 and Version2 Event messages. Upon detection of a PTP Event Message, the device will capture the transmit timestamp and provide the timestamp value to software through the PTP Transmit Timestamp Register (PTP_TXTS).

Since software knows the order of packet transmission, only the timestamp is recorded (there is no need to record sequence number or other information). The device can buffer four timestamps. Software should make an adjustment to the transmit timestamp to account for delay from the timestamp point to the wire. The recommended adjustment varies for modes of operation as follows:

- 100Base-TX: 0 ns
- 100Base-FX: 0 ns + transmit latency for fiber transceiver
- 10Base-T: 95 ns

The PTP Status Register (PTP_STS) indicates a transmit timestamp is available to software. After reading the transmit timestamp, software may recheck the PTP_STS register to see if another timestamp is ready.

An interrupt may be generated, if enabled, upon a Transmit Timestamp Ready.

3.1.2.1 Transmit Timestamp Configuration

Transmit Timestamp operations are controlled through the PTP Transmit Configuration Registers (PTP_TXCFG0, and PTP_TXCFG1). The Transmit Timestamp unit may be programmed to detect PTP messages encoded in three main types of packets: UDP/IPv4, UDP/IPv6, and Layer2 Ethernet. The device also directly supports detection of IEEE 1588 messages based on both Version 1 and Version 2 of the specification. To allow compatibility with future versions, a version field may be programmed to accept any version of the standard.

The Timestamp unit can identify PTP packets which include VLAN tags and generate timestamps for those packets. It can also handle multiple VLAN tags if they all have the standard ethertype (0x8100). Similarly, it can detect 802.2 SNAP frames. No special configuration is required to allow detection of PTP messages in VLAN tagged frames or 802.2 SNAP frames.

For UDP/IP operation, the Transmit Timestamp unit may be programmed to detect the four IANA assigned multicast IP destination addresses for IEEE 1588 (224.0.1.129, 224.0.1.130, 224.0.1.131, 224.0.1.132), as well as the defined IP destination address for the Peer Delay Mechanism (224.0.0.117). In most cases, the Transmit Timestamp Unit does not need to filter on IP destination addresses, since any outgoing PTP Event message should be timestamped. The Transmit Timestamp Unit can be configured to filter on IP addresses or it can be configured to capture timestamps for PTP Event messages independent of the IP destination address.

For IPv4 and IPv6, the protocol field of the IP Header is evaluated to determine the next protocol header. Acceptable intermediate headers include: Authentication Header (AH), IPv4 (in IPv4), IPv6 (in IPv6), and IPv6 extension headers. No status is kept for intermediate headers. Multiple intermediate headers are allowed.

To better distinguish between message types, the Timestamp unit may be configured to filter on the first octet of the PTP message. In Version 2, this allows filtering based on the transportSpecific and MessageType fields, and can be used to determine Event messages from General messages. Filtering on the first octet of the PTP message is enabled by setting mask and data values through the PTP_TXCFG1 Register.

For PTPv2, to generate timestamps for PTP Event messages only, the BYTE0_MASK and BYTE0_DATA fields are typically set to 0xF8 and 0x00 respectively.

3.1.2.2 One-Step Operation

In some cases, the Transmitter can be set to operate in a One-Step mode. For Sync Messages, a One-Step device can automatically insert Timestamp information in the outgoing packet. This eliminates the need for software to read the Timestamp and send a Follow_Up Message. The timestamp will not include the delay from the timestamp point to the wire. For IEEE 1588 version 2, the timestamp corrections may be included in the correctionField of the PTP Sync message (see IEEE 1588 Transmit Packet Parser and Timestamp Unit section for correction values).

The full seconds field specified by version 2 is 48 bits, but only 32 bits are stored on the device. Since the hardware does not keep the upper 16-bits of the clock, software must insert the upper 16-bits of the seconds value (bits 47:32) into the outgoing packet. The remainder of the timestamp (32-bits of seconds, 32-bits of nanoseconds) must be set to 0. Hardware will insert these fields of the timestamp, and correct the CRC for the packet. If requested for transport over UDP, the hardware will also correct the UDP checksum by modifying the last two octets of the UDP data. For IPv6, the extra 2 octets are required by the specification. For IPv4, the extra 2 octets are optional, but must be included by software for the algorithm to work correctly.

Overflow of the transmit timestamp queue can affect one-step operation. The timestamp queue must be serviced to prevent overflow if the BYTE0_MASK and BYTE0_DATA fields are not configured to filter out PTP General messages.

The following table indicates the method of insertion:

Message Type	Condition	Handling
V1 sync		Insert timestamp in originTimestamp field. UDP checksum field must be 0.
V2 IPv4/UDP Sync	CHK_1STEP = 0	Insert timestamp in originTimestamp field. UDP checksum field must be 0.
V2 IPv4/UDP Sync	CHK_1STEP = 1	Insert timestamp in originTimestamp field. Modify 2-byte field following Sync message to fix UDP checksum (software must set this field to 0).
V2 IPv6/UDP Sync	CHK_1STEP = 1 (required for IPv6)	Insert timestamp in originTimestamp field. Modify 2-byte field following Sync message to fix UDP checksum (software must set this field to 0).
V2 Layer2 Sync		Insert timestamp in originTimestamp field.

Table 3.1-1 – Transmit One-Step Operation

3.1.2.3 Delay_Req Timestamp Insertion

For slave only operation, the device may be programmed to return Delay_Req timestamps in incoming Delay_Resp messages. In this mode, it is expected that only Delay_Req messages will require transmit timestamps. The most recent transmit timestamp will be inserted into any incoming Delay_Resp message.

Software must still verify that the incoming Delay_Resp message is in response to the most recently sent Delay_Req message. Since all timestamps will be returned in Delay_Resp messages, no timestamps will need to be transferred through the management interface.

To enable Delay_Req timestamp insertion, set the DR_INSERT bit in the PTP_TXCFG register. In addition, the receive logic must also be programmed for timestamp insertion. The timestamp will be inserted at the same offsets as programmed for receive timestamp operation. See Section 3.1.3.4 Receive Timestamp Insertion for a description of the receive operation.

Enabling Delay_Req timestamp insertion modifies operation of the Transmit Timestamp Unit and is not compatible with other transmit timestamp operations. This feature should be disabled when the device transitions from Slave to Master operation. This mode should also not be used when using the Peer Delay mechanism. When disabling Delay_Req timestamp operation, it is possible for the device to indicate a transmit timestamp is available. Software should flush any transmit timestamps using the PTP_STS and PTP_TXTS registers.

3.1.3 IEEE 1588 Receive Packet Parser and Timestamp Unit

The IEEE 1588 receive parser monitors receive packet data to detect IEEE 1588 Version1 and Version2 Event messages. Upon detection of a PTP Event message, the device will capture the receive timestamp and provide the Timestamp value to software. In addition to the Timestamp, the device will record the 16-bit SequenceId, the 4-bit messageType field, and generate a 12-bit hash value for octets 20-29 of the PTP event message. The device will also indicate a timestamp is available by setting a bit in the PTP Status Register. The device can buffer four timestamps. Software should make an adjustment to the receive timestamp to account for delay from the wire to the timestamp point. The recommended adjustment varies for modes of operation as follows:

• 100Base-TX: 215 ns
• 100Base-FX: 120 ns + receive latency for fiber transceiver
• 10Base-T: 300 ns

The RXTS_RDY bit in the PTP Status Register (PTP_STS) indicates a receive timestamp is available to software. After reading the receive timestamp, software may recheck the PTP_STS register to see if another timestamp is ready.

An interrupt will be generated, if enabled, upon a Receive Timestamp Ready.

3.1.3.1 Reported messageType

For each Timestamp captured, the Timestamp unit will record the messageType. For version 1 of the IEEE 1588 specification the Timestamp unit will return the lower four bits of the control field (octet 32 in the message). Otherwise, the Timestamp unit will record the version 2 messageType field which is the least significant bits of the first octet in the PTP message.

3.1.3.2 Source Identification Hash Value

For each Timestamp captured, the Timestamp unit will also record a 12-bit hash value on octets 20-29 of the PTP event message. For version 1 of the IEEE 1588 specification, this corresponds to the messageType, sourceCommunicationTechnology, sourceUuid, and sourcePortId fields. For version 2 of the IEEE 1588 specification, this corresponds to the 10-octet sourcePortIdentity field. The combination of hash value and sequenceId, allows software to correctly match a Timestamp with the correct receive event message.

The hash algorithm used is the CRC function as defined in section 3.2.8 the IEEE 802.3 specification.

The Timestamp unit returns the 12 most significant bits as the CRC computation (the resultant bits are not complemented as done in the 802.3 CRC generation).

To minimize unnecessary timestamp capture, the device may be configured to filter based on the source identification hash value. This value may be programmed by writing to the PTP_RXHASH register. If the source hash value for the incoming PTP event message does not match the programmed hash value, then the message will not be timestamped.

3.1.3.3 Receive Timestamp Configuration

Receive timestamp operations are controlled through the PTP Receive Configuration Registers (PTP_RXCFG0, PTP_RXCFG1, PTP_RXCFG2, and PTP_RXCFG3). The Receive Timestamp unit may be programmed to detect PTP messages encoded in three main types of packets: UDP/IPv4, UDP/IPv6. and Layer2 Ethernet.

The device also directly supports detection of IEEE 1588 messages based on both Version 1 and Version 2 of the specification. To allow compatibility with future versions, a version field may be programmed to Timestamp messages using any version of the standard.

The Timestamp unit can identify PTP packets which include VLAN tags and generate timestamps for those packets. It can also handle multiple VLAN tags if they all have the standard ethertype (0x8100). Similarly, it can detect 802.2 SNAP frames. No special configuration is required to allow detection of PTP messages in VLAN tagged frames or 802.2 SNAP frames.

For UDP/IP operation, the Receive Timestamp unit may be configured to detect each of the four IANA assigned multicast IP destination addresses for IEEE 1588 (224.0.1.129, 224.0.1.130, 224.0.1.131, 224.0.1.132), as well as the defined IP destination address for the Peer Delay Mechanism (224.0.0.117).

The Timestamp unit may also be configured to generate a Timestamp if the IP destination address matches a user-programmed IP address. In IPv4, the full 32-bit address will be matched. In IPv6, the first 16-bits and last 16-bits will be matched.

For IPv4 and IPv6, the protocol field of the IP Header is evaluated to determine the next protocol header. Acceptable intermediate headers include: Authentication Header (AH), IPv4 (in IPv4), IPv6 (in IPv6), and IPv6 extension headers. No status is kept for intermediate headers. Multiple intermediate headers are allowed.

To better distinguish between message types, the Timestamp unit may be configured to filter on the first octet of the PTP message. In Version 2, this allows filtering based on the transportSpecific and MessageType fields.

The Receive Timestamp Unit can also filter on the domainNumber field in the version 2 PTP message. If Domain Field matching is enabled, the domainNumber field must match exactly the user-programmed value in order for a timestamp to be generated for the packet.

An additional control allows the Receive Timestamp Unit to generate a timestamp for PTP messages with the Alternate_Master flag set, otherwise messages with the Alternate_Master flag will not be timestamped.

For PTPv2, to generate timestamps for PTP Event messages only, the BYTE0_MASK and BYTE0_DATA fields are typically set to 0xF8 and 0x00 respectively.

3.1.3.4 Receive Timestamp Insertion

The IEEE 1588 Receive Timestamp unit can deliver the timestamp to software by inserting the timestamp in the received packet. This allows for a simple method to deliver the packet to software without having to be concerned with how to match the timestamp to the correct packet. This eliminates the need to provide software with the sequenceId, messageType, and source information. In addition, it eliminates the need to read the Receive Timestamp through the Management Interface.

Receive timestamp insertion is enabled by setting the TS_INSERT bit in the PTP Receive Configuration Register 3 (PTP_RXCFG3). In addition, the TS_APPEND bit will force timestamps to always be appended to the Receive PTP message for Layer2 Ethernet frames. If TS_APPEND is not set, timestamps will be placed in programmable locations within the message. It is recommended that Reserved fields in the version 1 and version 2 specification should be used, since these fields are required to be transmitted on the network as 0. Fields for seconds and nanoseconds should be programmed using the RXTS_SEC_OFF and RXTS_NS_OFF fields of the PTP_RXCFG4 register.
In all cases, the Ethernet CRC will be checked and regenerated. For IPv6, the UDP checksum will be corrected by modifying the last 2 bytes of UDP data, which follow the PTP message. If incoming IPv4 packets contain an additional 2 bytes of UDP data following the PTP message, software may enable modification of the last 2 bytes. Otherwise, the UDP checksum will be cleared to 0 by the device and any UDP checksum failure will be propagated as a CRC failure.

Some of the filtering capabilities are ignored for timestamp insertion. These fields include the source identification hash, domain match, and alternate master flag. If a timestamp is not desired, these messages should be discarded by software anyway.

The length of the seconds field is programmable from 0 to 4 bytes using the TS_SEC_EN and TS_SEC_LEN fields of the PTP_RXCFG4 register. For most applications, one byte should be plenty to determine the actual received time (based on rate of message reception and Timestamps from outgoing messages). If programmed to 0 bytes, the least significant seconds bit will be available in bit 30 of the nanoseconds field.

Bit 31 of the nanoseconds field is used to indicate that a time correction has been completed prior to the capture of this timestamp. This flag will be asserted when either a clock step adjustment or a temporary rate correction has been completed. Software can use this to determine if the timestamp includes the latest time correction. Note: For Revision A1 engineering samples, bit 31 was used to indicate bit 1 of the seconds field if the length of the seconds field was programmed to 0 bytes.

The following table indicates the preferred method of attachment.

Message Type	Condition	Handling
V1 Sync V1 Delay_Req	TS_APPEND = 0	Place 8-bit seconds field in 1st Reserved byte (byte 33). Place nanoseconds field in 4-byte Reserved field (bytes 36-39). Zero out UDP checksum. On UDP checksum failure, force CRC failure.
V2 IPv4/UDP Sync V2 IPv4/UDP Delay_Req V2 IPv4/UDP PDelay_Req V2 IPv4/UDP PDelay_Resp	TS_APPEND = 0	Place 8-bit seconds field in 1st Reserved byte (byte 5). Place nanoseconds field in 4-byte Reserved field (bytes 16-19). Zero out UDP checksum. On UDP checksum failure, force CRC failure.
V2 IPv6/UDP Sync V2 IPv6/UDP Delay_Req V2 IPv6/UDP PDelay_Req V2 IPv6/UDP PDelay_Resp	TS_APPEND = 0	Place 8-bit seconds field in 1st Reserved byte (byte 5). Place nanoseconds field in 4-byte Reserved field (bytes 16-19). Modify last 2 bytes of UDP data to correct checksum. UDP checksum failure will be propagated as a UDP checksum failure.
V2 Layer2 Sync V2 Layer2 Delay_Req V2 Layer2 PDelay_Req V2 Layer2 PDelay_Resp	TS_APPEND = 0	Place 8-bit seconds field in 1st Reserved byte (byte 5). Place nanoseconds field in 4-byte Reserved field (bytes 16-19). Zero out UDP checksum. On UDP checksum failure, force CRC failure.
V2 Layer2 Sync V2 Layer2 Delay_Req V2 Layer2 PDelay_Req V2 Layer2 PDelay_Resp	TS_APPEND = 1	Append to end of packet. Offset for nanoseconds should be set to 0. Offset for seconds should be set to 0. Seconds field will be appended following nanoseconds field. The extended packet length with be equal to 4 bytes plus the size of the seconds field.

Table 3.1-2 – Receive Timestamp Insertion

3.1.4 IEEE 1588 Triggers

The device can be programmed to generate a trigger signal on an output pin based on the IEEE 1588 time value. The trigger can be programmed to generate a one-time rising or falling edge, a single pulse of programmable width, or a periodic signal. The device supports up to 8 trigger signals which can be output on any of the GPIO signal pins.

The triggers are configured through the PTP Trigger Configuration Registers. For each trigger, the following configuration options are available:

- Single/Periodic control - Indicates whether trigger will generate a single edge/pulse or a periodic signal
- Pulse/Edge control - Indicates if trigger is a pulse or single edge
- Trigger-if-late control - allows immediate trigger if start time is earlier than current clock time (available only for Triggers 0 and 1).

- Notify - generate status on Trigger completion or on a trigger error (set too late)
- Toggle mode - toggle from current state (ignore Initial state setting)
- GPIO pin Select - indicates which I/O pin is used for the trigger output signal

The trigger time and width settings are controlled through the PTP Control and Time Data registers. Trigger control information consists of:

- Start Time (32-bit seconds, 30-bit nanoseconds)
- Initial state - Indicates initial state of signal to be set when trigger is armed (0 will cause a signal rise at trigger time, 1 will cause a signal fall at trigger time). This control is ignored in Toggle mode.
- Wait for Rollover - Indicates that the device should not arm the trigger until after the seconds field of the clock time has rolled over from 0xFFFF_FFFF to 0.
- Pulsewidth (2-bit seconds, 30-bit nanoseconds)
- Pulsewidth2 (2-bit seconds, 30-bit nanoseconds for Periodic Pulse) or (16-bit seconds for Periodic Edge)

For Triggers 0 and 1, in a Single or Periodic Pulse type signal, a second Pulsewidth value controls the 2nd pulse width (period is Pulsewidth + Pulsewidth2). For Edge type signals, Pulsewidth2 is interpreted as 16-bit seconds field and Pulsewidth1 is a 30-bit nanoseconds field. For all other triggers, the high and low pulse widths are the same (period is twice Pulsewidth).

3.1.4.1 Initializing a Trigger

To initialize a trigger, the trigger configuration should be set using the appropriate PTP Trigger Configuration Register. This allows for setting of parameters that probably do not need to change if the trigger is to be rearmed at a later time.

Arming a trigger consists of the following steps:

1. Set the Trig_Load bit in the PTP Control Register (PTP_CTL) along with the Trig_Sel setting for the trigger. This will disable the trigger if it was previously enabled.
2. Write to PTP_TDR: Start_time_ns[15:0]
3. Write to PTP_TDR: Initial state, Wait for Rollover, Start_time_ns[29:16]
4. Write to PTP_TDR: Start_time_sec[15:0]
5. Write to PTP_TDR: Start_time_sec[31:16]
6. Write to PTP_TDR: Pulsewidth[15:0]
7. Write to PTP_TDR: Pulsewidth[31:16]
8. Write to PTP_TDR: Pulsewidth2[15:0] (for Triggers 0 and 1 only)
9. Write to PTP_TDR: Pulsewidth2[31:16] (for Triggers 0 and 1 only)
10. Set the Trig_En bit in the PTP_CTL register along with the Trig_Sel setting for the trigger

If fields are not changing from previous settings, the latter writes to the PTP_TDR register may be skipped. Step 10 is not necessary if all appropriate fields are written.

3.1.4.2 Reading Trigger Control information

Reading the trigger control settings is similar to the process for writing these values:

1. Set the Trig_Read bit in the PTP Control Register (PTP_CTL) along with the Trig_Sel setting for the trigger.
2. Read fields from PTP_TDR in same order as written above.

Note that for periodic signals, the time value being read back is the next programmed trigger time rather than the start trigger time (these may or may not be the same value). This capability is essentially for diagnostic purposes only as there should be no need to read back the trigger control setting in normal operation. Care should be taken as the next trigger time may change during the read process. To ensure the time value is stable, the trigger should be disabled prior to initiating the read process.

3.1.4.3 Multiple Trigger assignment

The device supports assigning multiple triggers to the same GPIO pin. This allows for generating multiple similar events on the same I/O signal using multiple trigger controls. The trigger signals are OR'ed together to form a combined signal. The simplest combination is to combine two or more positive pulse triggers on a single I/O.

3.1.4.4 Trigger Notification and Interrupts

Each trigger may be programmed to generate status on completion or on an error. This function is enabled by setting the Trigger Notify control (TRIGn_NOTIFY) in the associated PTP Trigger Configuration Register (PTP_TRIGn).

If notification is enabled, when the trigger completes, the Trigger Ready bit will be set in the PTP Status Register. In addition, a Done bit is set in the PTP Trigger Status Register. If the trigger detects an error due to being armed late, it will also set the Trigger Late indication. For a periodic signal, if the Trigger-if-late control is set, no Notification will be generated. If the Trigger-if-late is not set, the trigger will set both the Done and Late indications.

If trigger interrupts are enabled in the PTP Status Register, an interrupt will be generated upon completion of the trigger.

3.1.4.5 Trigger Special Conditions

The IEEE 1588 specification calls for a 48-bit seconds field. Since the internal IEEE 1588 clock implements only a 32-bit seconds field, software must keep track of the upper 16-bits of the seconds field. Note that the 32-bit seconds field will rollover approximately every 136 years. The rollover occurs at 2^{32} seconds (4.294967296 * 10^9 ns). There are several exceptions to trigger handling when the clock approaches the rollover condition:

1. When programming a trigger time that will be after the rollover, software should set the "Wait For Rollover" control when programming the trigger. This is bit 30 of the second write to PTP_TDR.
2. Software should not set a trigger time within one reference clock period prior to the rollover point, otherwise the trigger may not occur. Instead, software should program a trigger time of 0 and set the "Wait For Rollover" control bit. Alternatively, software may program a trigger time greater than one reference clock period prior to the rollover time.
3. Similarly, a periodic signal may stop if it results in a trigger time within one reference clock period prior to the rollover point.

4. Software should avoid making a step time adjustment that may cause the clock to step forward or backwards across the maximum value. Doing this could cause invalid trigger operation. Adjustments using the Temporary Rate function will not cause any issues with trigger operation.

3.1.4.6 IEEE 1588 Pulse Per Second Output

The device can be programmed to output a Pulse-Per-Second (PPS) signal using the trigger functions. If a 50% duty cycle is acceptable, then any of the triggers may be used. If the PPS signal requires any other duty cycle (for example 200 ms high time) then Trigger0 or Trigger1 must be used.

To configure the trigger, software should enable a periodic trigger using the PTP Trigger Configuration Register (PTP_TRIG). For example, to configure Trigger0 to use GPIO2, write 0xC201 to the PTP_TRIG register. Software should then initialize the trigger as described above with the appropriate start time and pulse width information to allow the trigger to begin at a second boundary in the future.

If a large adjustment to the PTP time is made, the PPS trigger may need to be rearmed to avoid a long period of inactivity or a period of rapid pulsing. For a large positive change in time (greater than one second), the trigger should be rearmed to trigger at a second boundary relative to the new time, prior to making the time adjustment. For a large negative change, the trigger should be disabled prior to making the time adjustment, and then rearmed after making the time adjustment.

3.1.5 IEEE 1588 Event Timestamping

The device can be programmed to timestamp an event by monitoring an input signal. The event can be monitored for rising edge, falling edge, or either. The Event Timestamp Unit can monitor up to eight events which can be set to any of the the GPIO signal pins. PTP event timestamps are stored in a queue which allows storage of up to eight timestamps.

When an event timestamp is available, the device will set the EVENT_RDY bit in the PTP Status Register. The PTP Event Status Register (PTP_ESTS) provides detailed information on the next available event timestamp, including information on the event number, rise/fall direction, and indication of events missed due to overflow of the devices Event queue.

If more than one event occurs at the same time, the MULT_EVNT bit will be set in the PTP_ESTS register. In this case, an extended status value will be available on the first read of the PTP_EDATA register. The extended status value indicates each of the event monitors and rise/fall directions that occurred to generate the timestamp capture. The timestamp values are provided for software through the PTP Event Data Register (PTP_EDATA).

Event timestamp values should be adjusted by 3*reference clock period + 11 ns = +35 ns to compensate for input path and synchronization delays.

3.1.5.1 Enabling Event Monitoring

The Event Timestamp Unit is configured through the PTP Event Configuration Register (PTP_EVNT). Each of the eight event monitors may be individually programmed through this register. Enabling an event monitor is done by setting the EVNT_SEL field to the appropriate event, setting the EVNT_WR bit to 1, setting the EVNT_GPIO field to select the appropriate GPIO, and setting the EVNT_RISE and/or EVNT_FALL enables to 1.

The current state of the GPIO pin may cause an immediate timestamp capture if the enables are set at the same time as the GPIO selection field. For example, if GPIO3 currently is currently at a logic high state when the event monitor is set to that GPIO, the event monitor will see a rising edge. To avoid this, software may program the GPIO selection prior to setting the EVNT_RISE enable.

For example, to enable event monitor 2 to monitor GPIO3 for rising edge detection:

1. Write 0x0305 to PTP_EVNT to select GPIO3 for event monitor 2.
2. Write 0x4305 to PTP_EVNT to enable event rise detection on GPIO3.

3.1.5.2 Single Event Capture

Each event monitor may be placed in a single-event capture mode. In this mode, the event monitor will capture a single event timestamp. The EVNT_RISE and EVNT_FALL enable bits will be cleared upon the event capture. Single-event capture mode may be enabled by setting the EVNT_SINGLE bit in the PTP_EVNT register.

3.1.5.3 Reading Event Timestamps

The process for reading event timestamps is as follows:

1. Read PTP_ESTS to determine if an event timestamp is available.
2. Read from PTP_EDATA: Extended Event Status[15:0] (available only if PTP_ESTS:MULT_EVNT is set to 1)
3. Read from PTP_EDATA: Timestamp_ns[15:0]
4. Read from PTP_EDATA: Timestamp_ns[29:16] (upper 2 bits are always 0)
5. Read from PTP_EDATA: Timestamp_sec[15:0]
6. Read from PTP_EDATA: Timestamp_sec[31:16]
7. Repeat Steps 1-6 until PTP_ESTS = 0

If desired, software may skip reading all or a portion of the timestamp based on the value of the PTP_ESTS:EVNT_TS_LEN field. For example, if the current event timestamp has the same value of seconds as the previous event timestamp, the the EVNT_TS_LEN field will be set to one. This indicates that two fields (both Timestamp_ns fields) contain values that have changed. The seconds values do not need to be read since software can hold those values from the previous event.

3.1.5.4 Event Interrupts

If event interrupts are enabled in the PTP Status Register, an interrupt will be generated upon detection of the event.

3.1.6 PTP Interrupts

The PTP module may interrupt the system using the PWRDN_INTN pin on the device, shared with other interrupts from the PHY. As an alternative, the device may be programmed to use a GPIO pin to generate PTP interrupts separate from other PHY interrupts.

3.1.6.1 Using the Shared Interrupt Pin

To use the PWRDN_INTN pin, software must program the MII Interrupt Control Register (MICR) to select the PTP interrupt. Setting the PTP_INT_SEL bit in the MICR will allow PTP interrupts based on the setting of the mask bit at the MII Interrupt Status Register bit 3 (MISR[3]). The PTP interrupt status will be available at MISR[11]. Software also must configure which PTP functions may generate interrupts using the interrupt enables in the PTP Status Register (PTP_STS).

Once enabled, interrupt handling is as follows:

1. Read MISR to determine if PTP interrupt has occurred
2. Read PTP_STS to determine which PTP function has generated an interrupt
3. Process Trigger, Event, or Timestamp as indicated by PTP_STS
4. Repeat steps 2 and 3 until PTP_STS[11:8] = 0

Note that the PTP interrupt will not be rearmed until a read of PTP_STS returns 0 in the upper octet.

3.1.6.2 Interrupts using a GPIO pin

To use a GPIO pin for interrupts, software must program the PTP_INTCTL register with the GPIO pin to use for interrupts. The interrupt will be an active low signal, implemented as an Open-Drain function (drive low, pulled high via an external pullup resistor). Software also must configure which PTP functions may generate interrupts using the interrupt enables in the PTP Status Register (PTP_STS). This mechanism provides slightly simpler handling of PTP interrupts since there is no need to check the MISR for interrupts.

Once enabled, interrupt handling is as follows:

1. Read PTP_STS to determine which PTP function has generated an interrupt
2. Process trigger, event, or timestamp as indicated by PTP_STS
3. Repeat steps 1 and 2 until PTP_STS[11:8] = 0

Note that the PTP interrupt will not be cleared until a read of PTP_STS returns 0 in the upper octet.

Software may temporarily disable interrupt signaling by clearing the interrupt enable bits in the PTP_STS register to 0.

3.1.7 IEEE 1588 Output Clock Signal

The DP83640 and DP83630 generate controls for providing a synchronized clock signal for use by external devices. The output clock signal can be any frequency which is divisible by N from 250 MHz, where N is in the range of 2 to 255. This provides nominal frequencies from 125 MHz down to 980.4 kHz. The output clock signal is frequency accurate to the 1588 clock time of the device. In addition, if clock time adjustments are made using the Temporary Rate capabilities, then all time adjustments will be tracked by the output clock signal as well. Note that any step adjustment in the 1588 clock time will not be accurately represented on the 1588 clock output signal.

The 250 MHz clock source may be selected from either the internal FCO or PGM. The FCO offers reduced jitter in the output clock, while the PGM offers a wider range of frequency correction in the output clock.

The output clock cannot be used when the PTP logic is using an external reference clock. Note also that the sub-nanosecond step size must be adjusted to correspond to the programmed reference clock period of the PTP logic.

Example Configuration of the Output Clock:

Conditions:

Nominal Output Clock Frequency = 10 MHz (divide-by-25)
250 MHz Clock Source

1. Write 0x8019 to PTP_COC

This enables the clock output using a divide-by-25 (0x19) from the 250 MHz FCO clock.

The maximum sub-nanosecond step size that the output clock can track using the FCO is 0x1555556, which corresponds to +/- 651 ppm assuming a nominal 125 MHz PTP reference clock. Using the PGM, the maximum sub-nanosecond step size that the output clock can track is 0x3FFFFFF, which corresponds to +/- 1953 ppm.

3.1.8 Packet-based Event and Timestamp Delivery

The IEEE 1588 core implements a packet-based status mechanism that allows the PHY to queue up events and pass them to the microcontroller through the receive data interface. The packet, called a PHY Status Frame, may be used to provide IEEE 1588 status for transmit packet timestamps, receive packet timestamps, event timestamps, and trigger conditions. In addition the device can generate status messages indicating packet buffering errors and to return data read using the PHY Control Frame register access mechanism.

Each PHY Status Frame may include multiple status messages. The packet will be framed such that it will look like a form of IEEE 1588 frame to ensure that it will get to the 1588 Software stack. The PHY will provide buffering of any incoming packet to allow the status packet to be passed to the MAC.

Programmable inter-frame gap and preamble length allow the PHY to recover lost bandwidth in the case of heavy receive traffic.

In a PHY Status Frame, status messages are not provided in a chronological order. Instead, they are provided in the following order of priority:

1. PHY Control Frame Read Data
2. Packet Buffer Error
3. Transmit Timestamp
4. Receive Timestamp
5. Trigger Status
6. Event Timestamp

Each of the message types may be individually enabled, allowing options on which functions may be delivered in a PHY Status Frame.

The packet format may be configured to look like a Layer 2 Ethernet frame or a UDP/IPv4 frame.

3.1.8.1 Layer2 Ethernet Format for PHY Status Frames

The Layer2 Ethernet status frame appears as a standard IEEE 1588 frame encapsulated directly in an Ethernet frame. The format is as follows:

Field	Length	Value	Description
MAC Destination Address	6 octets	0x01 0x1B 0x19 0x00 0x00 0x00	Uses the IEEE 1588 defined destination MAC address for transport over Ethernet.
MAC Source Address	6 octets	0x08 0x00 0x17 0x0B 0x6B 0x0F	Uses a National Semiconductor assigned MAC address. Optionally may be configured to other preset values through the PSF_CFG0 register.
EtherType field	2 octets	0x88 0xF7	EtherType value for IEEE 1588.
Status Frame Payload	8 - 114 octets		Status Frame Payload.
Pad	0 - 38 octets	0x00	Pad to minimum Ethernet frame size (if necessary).

Table 3.1-3 – Layer2 Ethernet Format for PHY Status Frames

3.1.8.2 UDP/IPv4 Ethernet Format for PHY Status Frames

The UDP/IPv4 Ethernet status frame appears as a standard IEEE 1588 frame encapsulated in a UDP/IPv4 packet. The format is as follows:

Field	Length	Value	Description
MAC Destination Address	6 octets	0x01 0x00 0x5E 0x00 0x01 0x81	Uses the IEEE 1588 defined destination MAC address for transport over UDP/IPv4.
MAC Source Address	6 octets	0x08 0x00 0x17 0x0B 0x6B 0x0F	Uses a National Semiconductor assigned MAC address. Optionally may be configured to other preset values through the PSF_CFG0 register.
EtherType field	2 octets	0x88 0x00	IPv4 EtherType value.
IPv4 Version/IHL	1 octet	0x45	Version and IHL fields of IP Header. The Header Length is set to the minimum size of 5.
TOS	1 octet	0x00	Type of Service field of IP Header
Total Length	2 octets	0x2E or 0x3E	IP Total Length field. The setting of this field is dependent on whether an packet is arriving from the Physical layer. If no packet is arriving, the longer length will be used. If a packet is arriving, the shorter length will be used.
Identification	2 octets	incrementing	Identification field of the IP Header. This field will increment from 0 starting with the first status packet sent by the device.
Flags/Fragment Offset	2 octets	0x00	Flags and Fragment Offset fields of IP Header
Time to Live	1 octet	0x01	Time to Live field of IP Header
Protocol	1 octet	0x11	Protocol field of IP Header. This field will be set to indicate the IP payload is UDP.

Header Checksum	2 octets		IP Header checksum field. To assist in generating the checksum, software must program the PSF_CFG4 register with a ones-complement sum of all fixed fields of the IP Header. The device will add values for the Identification and Total Length fields.
Source Address	4 octets	programmable	Source Address field of IP Header. This may be programmed through the PSF_CFG2 and PSF_CFG3 registers.
Destination Address	4 octets	0xE0 0x00 0x01 0x81 For 224.0.1.129	Destination Address field of IP Header. This field uses the assigned multicast address for IEEE 1588 default domain.
UDP Source Port	2 octets	0x01 0x3F	UDP Source Port. Assigned to the UDP port value of 319 for IEEE 1588.
UDP Destination Port	2 octets	0x01 0x3F	UDP Destination Port. Assigned to the UDP port value of 319 for IEEE 1588.
UDP Length	2 octets	0x1A or 0x2A	UDP Length field. The setting of this field is dependent on whether a packet is arriving from the Physical layer. If no packet is arriving, the longer length will be used. If a packet is arriving, the shorter length will be used.
UDP Checksum	2 octets	0x00	UDP checksum field. Set to 0 to indicate no checksum has been generated.
Status Frame Payload	8 – 34 octets		Status Frame Payload.
Pad	0 – 38 octets	0x00	Pad to minimum Ethernet frame size (if necessary).

Table 3.1-4 – UDP/IPv4 Ethernet Format for PHY Status Frames

3.1.8.3 PHY Status Frame Payload

Independent of the format of the packet used, the basic payload of a PHY Status Frame will be the same. The first two octets replace the equivalent values in the IEEE 1588 header, and are programmable to allow easy differentiation from standard 1588 messages. The remaining portion of the payload is dedicated to status information and has no relation to standard 1588 message format.

Field	Length	Value	Description
transportSpecific messageType	1 octet	Programmable	Configurable fields to allow differentiation from standard IEEE 1588 frames. This field is programmable through the PSF_CFG1 register.
versionPTP	1 octet	Programmable	Configurable field to allow differentiation from standard IEEE 1588 frames. This field is programmable through the PSF_CFG1 register.
Status Message List	1-7 words		Status message list. Length of each status

	each		message is dependent on the type of status.
Termination Field	4 octets	0x00	The termination field of all 0s indicates the end of the list of status messages.

Table 3.1-5 – Status Frame Payload

3.1.8.4 Status Messages

Each status message includes a 16-bit Status Type field followed by up to seven 16-bit data fields. The length of the message is dependent on the type. The Status Type field has the following bit definitions:

- Status Type [15:12] : Type Value
- Status Type [11:0] : Extended Status

In most cases, the Status Data fields are equivalent to the order in which data will be returned on data reads. For example, the Transmit Timestamp message will return data in the same order that it would be read from the PTP_TXTS register.

The following are definitions for the six different status messages.

Field	Length	Value	Description
Status Type	1 word	0x1000	Status Type field. Type Value is 1. Extended Status is not used.
Timestamp_ns[15:0]	1 word		This field contains the least significant 16-bits of the transmit timestamp nanoseconds field.
Overflow_cnt[1:0], Timestamp_ns[29:16]	1 word		This field contains the most significant 14-bits of the transmit timestamp nanoseconds field. The Overflow_cnt value indicates if timestamps were dropped due to an overflow of the transmit timestamp queue. The overflow counter will stick at a value of three if additional timestamps were missed.
Timestamp_sec[15:0]	1 word		This field contains the least significant 16-bits of the transmit timestamp seconds field.
Timestamp_sec[31:16]	1 word		This field contains the most significant 16-bits of the transmit timestamp seconds field.

Table 3.1-6 – Transmit Timestamp Status Message (length = 5 16-bit words)

Field	Length	Value	Description
Status Type	1 word	0x2000	Status Type field. Type Value is 2. Extended Status is not used.
Timestamp_ns[15:0]	1 word		This field contains the least significant 16-bits of the receive timestamp nanoseconds field.
Overflow_cnt[1:0], Timestamp_ns[29:16]	1 word		This field contains the most significant 14-bits of the receive timestamp nanoseconds field. The Overflow_cnt value indicates if timestamps were dropped due to an overflow of the receive timestamp queue. The overflow counter will stick at a value of three if additional timestamps were missed.
Timestamp_sec[15:0]	1 word		This field contains the least significant 16-bits of the receive timestamp seconds field.
Timestamp_sec[31:16]	1 word		This field contains the most significant 16-bits of the receive timestamp seconds field.
sequenceId[15:0]	1 word		This field contains the sequenceId field from the PTP message.
messageType[3:0], source_hash[11:0]	1 word		This field contains the 4-bit messageType field from the PTP message. The source_hash value is 12-bit Source Identification has value over the sourcePortIdentity fields of the PTP message.

Table 3.1-7 – Receive Timestamp Status Message (length = 7 16-bit words)

Field	Length	Value	Description
Status Type	1 word	0x3000	Status Type field. Type Value is 3. Extended Status is not used.
Trigger Status[15:0]	1 word		This field contains the trigger status value as defined in the PTP_TSTS register.

Table 3.1-8 – Trigger Status Message (length = 2 16-bit words)

Field	Length	Value	Description
Status Type	1 word	0x4xxx	Status Type field. Type Value is 4. Extended Status provides the Event Status as defined in the PTP_ESTS register bits [11:0].
Extended Event Status[15:0]	1 word		OPTIONAL: This field contains the extended status as defined in the PTP_EDATA register. This field is available if the Event Status indicates detection of multiple events.
Timestamp_ns[15:0]	1 word		This field contains the least significant 16-bits of the event timestamp nanoseconds field.
Timestamp_ns[29:16]	1 word		OPTIONAL: This field contains the most significant 14-bits of the event timestamp nanoseconds field. The upper two bits will always be 0. This field is available if the Event Status indicates an event timestamp length of 2 or more (EVNT_TS_LEN >= 1).
Timestamp_sec[15:0]	1 word		OPTIONAL: This field contains the least significant 16-bits of the event timestamp seconds field. This field is available if the Event Status indicates an event timestamp length of 3 or more (EVNT_TS_LEN >= 2).
Timestamp_sec[31:16]	1 word		OPTIONAL: This field contains the most significant 16-bits of the event timestamp seconds field. This field is available if the Event Status indicates an event timestamp length of 4 (EVNT_TS_LEN = 3).

Table 3.1-9 – Event Timestamp Status Message (length = variable)

Field	Length	Value	Description
Status Type	1 word	0x5xxx	Status Type field. Type Value is 5. Extended Status provides an error indication: [0] Packet Buffer data overflow error [1] Packet counter overflow.

Table 3.1-10 – Status Frame Error Status Message (length = 1 16-bit word)

Field	Length	Value	Description
Status Type	1 word	0x6000	Status Type field. Type Value is 6. Extended Status includes the following fields for the PHY Control Frame register read: [4:0] Register Address [7:5] Page Select
Read Data[15:0]	1 word		This field contains the 16-bit data value read from the register by the PHY Control Frame.

Table 3.1-11 – PCF Read Data Status Message (length = 2 16-bit words)

3.1.8.5 PHY Status Frames, RMII, and half-duplex operation

Because PHY Status Frames generate traffic across the MII/RMII that do not relate to traffic on the wire, the CRS_DV signaling does not operate as defined in the RMII specification. In full-duplex mode, the PHY Status Frame function will generate CRS directly from RX_DV. In half-duplex mode, the PHY will pass CRS directly from the receiver to the internal RMII translation function. Thus CRS_DV may not be asserted correctly in half-duplex mode.

For MII operation, PHY Status Frames should work with half-duplex mode operation since CRS will be asserted when the receiver is active, rather than when the Receive MII interface is active.

3.1.9 PTP Reference Clock Modes

The IEEE 1588 PTP logic operates on a nominal 125 MHz reference clock generated by an internal PGM. However, options are available to use a divided-down version of the PGM clock to reduce power consumption at the expense of precision, or to use an external reference clock of up to 125 MHz in the event the 1588 clock is tracked externally.

The PTP_CLKSRC register contains two fields:

- Clock Source (PTP_CLKSRC[15:14]) - selects internal 125 MHz PGM reference (default), divided down PGM reference clock (divide values are 2-15), or external reference clock.
- Clock Period (PTP_CLKSRC[6:0]) - Configures the period, in nanoseconds, of the PTP reference clock. This field will not accept values less than 8 ns. If the divided-down PGM reference clock is used, this field will be interpreted as a multiple of 8, so bits 2:0 will be ignored. If the external reference clock is used, this field is the integer nanosecond period of the external clock.

Example Using Divided-Down PGM Reference Clock:

Conditions:
Desired Reference Clock Frequency = 25 MHz (divide-by-5)
Reference Clock Source = PGM

1. Write 0x4048 to PTP_CLKSRC

This configures the reference clock with a 40 ns period sourced from the internal PGM.

Example Using External Reference Clock:

Conditions:
Desired Reference Clock Frequency = 10 MHz (10 ns period)
Reference Clock Source = External

1. Write 0x8064 to PTP_CLKSRC

This configures the reference clock with a 100 ns period sourced from external PTP reference clock input.

3.1.10 PTP Reset

The entire PTP function, including the 1588 Clock, associated logic, and PTP register space (with two exceptions), can be reset via the PTP_CTL:PTP_RESET bit. The PTP_COC and PTP_CLKSRC registers are not reset in order to preserve the nominal operation of the Clock Output.

3.1.11 Synchronous Ethernet Relation to PTP operation

If the device is configured for 100Mb/s Synchronous Ethernet operation, the entire core including the PTP logic will run synchronous to the recovered 25 MHz Receive Clock (assuming the PGM is the PTP reference clock source). This mode of operation allows the PTP logic to be frequency locked to its partner's transmit clock which significantly reduces clock offset in a slave device.

3.2 1588 Hardware Configuration

There are numerous configuration options related to the 1588 features available with the DP83640 and the DP83630. Many of the defaults are adequate for normal 1588 operation thus simplifying the steps needed by software to initialize the part. This section shows an example of some minimal steps needed to setup 1588 operation using the EPL configuration functions.

```
// Enable 1588 clock, set start time, set rate to 0
PTPEnable( portHandle, FALSE);
PTPClockSetRateAdjustment( portHandle, 0, FALSE, FALSE);
PTPClockSet( portHandle, 1, 0);
PTPSetClockConfig( portHandle, CLKOPT_CLK_OUT_EN, 0x0A, 0x00, 8);
PTPEnable( portHandle, TRUE);

// Configure Trigger 0 for PPS - only perform if using PPS
PTPEnableTriggers( portHandle, FALSE);
PTPSetTriggerConfig( portHandle, 0, TRGOPT_PERIODIC|TRGOPT_NOTIFY_EN, 1);
PTPArmTrigger( portHandle, 0, 2, 0, FALSE, FALSE, 500000000, 0);

// Disable Transmit and Receive Timestamp
PTPSetTransmitConfig( portHandle, 0, 0, 0, 0);
memset( &rxCfgItems, 0, sizeof( RX_CFG_ITEMS));
PTPSetReceiveConfig( portHandle, 0, &rxCfgItems);

// Flush Transmit and Receive Timestamps
while ( (events = PTPCheckForEvents( portHandle)))
{
    if ( events & PTPEVT_TRANSMIT_TIMESTAMP_BIT)
         PTPGetTransmitTimestamp( portHandle, &numSecs, &numNanoSecs, &overflowCount);
    else if ( events & PTPEVT_RECEIVE_TIMESTAMP_BIT)
         PTPGetReceiveTimestamp( portHandle, &numSecs, &numNanoSecs, &overflowCount, &seqId,
&msgType, &hashValue);
}

// Enable Transmit Timestamp operation
PTPSetTransmitConfig( portHandle, TXOPT_IP1588_EN|TXOPT_IPV4_EN|TXOPT_TS_EN,
                          1, 0xFF, 0x00);

// Enable Receive Timestamp operation
rxCfgItems.ptpVersion = 0x01;
rxCfgItems.ptpFirstByteMask = 0xFF;
rxCfgItems.ptpFirstByteData = 0x00;
rxCfgItems.ipAddrData = 0;
rxCfgItems.tsMinIFG = 0x0C;
rxCfgItems.srcIdHash = 0;
rxCfgItems.ptpDomain = 0;
rxCfgItems.tsSecLen = 0;
rxCfgItems.rxTsNanoSecOffset = 0;
rxCfgItems.rxTsSecondsOffset = 0;

rxCfgOpts = RXOPT_IP1588_EN0|RXOPT_IP1588_EN1|RXOPT_IP1588_EN2|
            RXOPT_RX_IPV4_EN|RXOPT_RX_TS_EN|RXOPT_ACC_UDP;
if ( ptpStackCfg->slaveOnly)
     rxCfgOpts |= RXOPT_RX_SLAVE;

PTPSetReceiveConfig( portHandle, rxCfgOpts, &rxCfgItems);
```

For additional details of configuration refer to the PTPTestApp in sections 4 and 5 of this document. The complete initialization sequence can be found in the **PTPTestApp.cpp** (refer to ConfigurePTP() and StartPTP() functions) which contains all the setup for calling into the library and in **ptpControl.c** (see the PTPInitHardware() function) in the library itself.

3.3 1588 Clock Synchronization Techniques

A primary design challenge in developing a highly accurate 1588 slave clock is in correctly adjusting a slave's local clock to closely track a 1588 master's clock with the desired accuracy. The DP83640 and the DP83630 offer a number of unique and powerful hardware assists that aid in meeting this challenge.

There are numerous clock correction algorithms that can be devised for this purpose. The best approach is determined by the desired clock accuracy, stability and overall system design characteristics. This section describes an algorithm developed by National that was used for internal device validation as well as device demonstration purposes. This can be used as a starting point, if desired, to develop a custom algorithm that meets the requirements of a customer's overall 1588 solution.

This algorithm is implemented in the IEEE 1588 open source protocol stack (PTPd) that is provided by National. This stack has been modified to make use of the DP83640 and DP83630 1588 related features and implements the algorithm described below.

When a slave first starts and has selected a best master to synchronize with, its local clock is typically very different then the master's clock. The first part of the algorithm is to set the local slave clock equal to the master's clock value. The master's clock value is obtained from a received sync or follow up message, depending on the 1588 configuration. This adjusts the clock to be fairly close to the master's clock. The EPL function "PTPClockSet" is used to explicitly set the slave's local clock value.

The slave clock is subsequently tuned closer to the master's clock using step adjustments to the local clock. This usually occurs for the next two or three received sync/follow-up messages and usually gets the slave clock to within a few microseconds of the master's clock. The EPL function "PTPClockStepAdjustment" is used to apply step adjustments to the local slave clock.

After the previous steps, the following algorithm can be used to keep the local slave clock very close to the master's clock. Accuracy better then 20ns is possible using these techniques assuming no quickly changing frequency skew occurs due to the clock itself or environment factors such as quickly changing clock temperature. Even better accuracy can be achieved by using Synchronous Ethernet mode. In synchronous mode a slave device uses a clock that is derived from the receive data stream sent by the master device.

The "Master to Slave Delay", "Slave to Master Delay", "One Way Delay" and "Offset from Master" values are calculated as specified in the IEEE 1588 specification and are not detailed in this document.

As part of the 1588 specification the "One Way Delay" (OWD) value is used to determine the "Offset from Master" (ERROFF) value. In these algorithms the OWD is calculated each sync/follow-up cycle and a history of these values is kept of length N_{owd}.

3.3.1 Step Adjustment with Rate Control

This algorithm uses a step adjustment to the local slave clock every sync cycle. A sync cycle occurs when a slave receives a sync message from the master (one-step operation) or after receiving a follow-up message from the master (two-step operation). Although this keeps the clock to within a few microseconds (us) of the master clock a changing rate adjustment to the local clock is used to fine tune the clock's frequency.

The primary disadvantage of this approach is that a step adjustment may be negative, such that the local slave clock value may step back in time. This approach should NOT be used in applications that may be sensitive to this.

A rate adjustment, either positive or negative, is applied to the clock every device clock period (8ns). The appropriate rate adjustment value (RAV) is determined by calculating the difference in actual clock frequencies between the slave and master clocks.

A rate adjustment value (RAV) is calculated each sync cycle. A history of RAV's is kept of length N_{rav}. Every N_{rav} sync cycles these values are averaged together to form a single rate adjustment value, known as an average rate adjustment value (ARAV).

A history of average rate adjustment values (ARAV) is kept of length N_{arav}. Every N_{rav} sync cycles the ARAV history values are averaged together to form a single value, known as the average of average rate adjustment values (AARAV).

The determined AARAV value is then applied to the device's local clock. This is accomplished using the EPL "PTPClockSetRateAdjustment" function and is applied as a non-temporary adjustment. The RAV history is then cleared and the algorithm repeats itself.

If during any sync cycle the "Offset from Master", or error offset, becomes too large, the entire algorithm is started over, thus starting where the clock is explicitly set to the master's clock value. This includes clearing all RAV and ARAV history values. The algorithm makes use of a sync count variable that counts the number of sync cycles initiated by the master. If the algorithm is restarted the sync count is set to 0.

For additional details refer to the PTPd source code (servo.c) where additional tuning refinements have been made. In particular the use of temporary rate adjustments and CLKOUT phase alignment has been implemented. This section will be updated in the future to document the current algorithms that have been developed.

4 Ethernet PHYTER “C” Library (EPL)

The Ethernet PHYTER Library (EPL) is a software library provided by National Semiconductor that can simplify and speed development of support software for National’s line of PHYTER® Ethernet Physical Layer devices. EPL is easily adapted to various target computing environments and is modular so that only the functionality that is actually used needs to be included in a final product. It aids in device identification, configuration, operational control, event handling and diagnostics.

Use of EPL in a target environment is completely optional and may simply be used as a pseudo code reference when developing custom PHY support software.

4.1 Functional Overview

EPL has been designed to provide the following functionality:

4.1.1 Device Support

The devices lists below are presently supported. The EPL software model will be used to support future Ethernet PHYTER devices from National.

- DP83848
- DP83849
- DP83620
- DP83630
- DP83640

4.1.2 Initialization / Deinitialization

EPL initialization and deinitialization functions are provided that must be called once by higher level software upon system initialization and deinitialization, respectively.

4.1.3 Device Setup / Enumeration

A method is provided that allows higher layers to enumerate the devices available on a specified interface bus.

4.1.4 Device Identification / Capabilities

EPL provides a function that identifies a device as well as providing device capability information. This information can be used by upper level software to know if particular functionality is available on a device, in a device independent fashion.

4.1.5 Number of extended register pages

The number of extended register pages defined and available is device specific. A function is provided that returns the number of extended register pages available on a particular device.

4.1.6 Device Reset

EPL provides a function that soft resets a particular device.

4.1.7 Read / Write Device Registers

EPL provides device port register read and write functions for both MDIO and MII management interfaces.

4.1.8 Set Loopback Mode

EPL provides a function that configures a port’s loopback mode.

4.1.9 Multiple Interface, Device, and Port aware and capable

EPL supports multiple interfaces to communicate with devices. Both MDIO and MII packet based (if hardware support is available) are supported. It can support multiple devices on each interface. In addition it supports devices that contain more then one PHY port (e.g. DP83849). A function is provided that returns the number of ports for a particular device.

4.1.10 Management Interface Error Checking

EPL leverages a error detection feature available with the DP83640 and DP83630 devices to validate register read/write operations through either MDIO or the MII packet based register access mechanisms. This feature is transparent to higher layer software, except for an OAI function that will be called by EPL to signal that data corruption has occurred through the management interface. This would normally indicate a serious system error.

4.1.11 Cable Status

EPL provides an API to obtain the following cable status information. Only a subset of this information is available on the DP83848.

- Polarity
- Cable Swapped
- Cable Length Estimate
- Frequency Offset
- Jitter (Variance)
- Receiver Signal to Noise Ratio (SNR)
- Set SNR Sample Time

4.1.12 Link Quality

With the DP83640, DP83630, DP83620 and DP83849 devices, the following DSP parameters can be queried. Threshold trigger values can be set as well as the ability to query for a trigger condition.

- DEQ C1
- DAGC
- DBLW
- FREQ
- FC

4.1.13 Link Status

The following general link status information is available:

- Up or Down
- Speed
- Duplex
- Polarity
- Auto MDIX
- MDI Crossed
- Auto-negotiation Enabled/Disabled
- PoE Device Present
- Idle Errors
- Low Power Mode
- Local / Remote Capabilities (PAUSE, best speed & duplex)

4.1.14 Link Configuration

EPL provides a method to configure a port's link parameters. The following information can be set:

- Forced or Auto-negotiation
- Duplex
- Speed
- Auto-MDIX
- Energy Detect

4.1.15 TDR Functions

The following TDR functionality are provided when the device is a DP83640, DP83630, DP83620 or DP83849.

- Cable Status (terminated, shorted, open, cross shorted, etc.)
- Cable Length for Tx/Rx pairs
- Obtain Oscilloscope trace of TDR pulses

4.1.16 MII Port Configuration

EPL provides functions to get/set the device's MII configuration. This only applies to the DP83849 dual-port device. The following configurations are supported:

- Normal (straight through)
- Full Port Swap
- Extender/Media Converter
- Broadcast Tx MII Port A
- Broadcast Tx MII Port B
- Mirror Rx Channel A
- Mirror Rx Channel B
- Disable Port A
- Disable Port B

4.1.17 BIST Tx Start/Stop/Status

EPL provides functions to start and stop BIST testing and also obtain BIST status information.

4.1.18 IEEE 1588

Refer to the previous DP83640 and DP83630 PHYTER High Precision IEEE 1588 Hardware Features section for details related to the IEEE 1588 capabilities.

4.1.18.1 IEEE 1588 Configuration

The library provides functions that can be called to configure the DP83640 and DP83630 PHTYER devices for IEEE1588 operation.

4.1.18.2 IEEE 1588 v1 Protocol

The library contains functions that can be used to implement and control the DP83640 and DP83630 PHTYER devices according to the IEEE 1588 v1 Protocol.

4.1.19 Operating System Abstraction Interface (OAI)

A set of OS primitive functions are defined that provide an abstraction for EPL from the underlying OS and processor environment. This interface is referred to as the OS Abstraction Interface (OAI). This interface includes memory management and other OS specific operations. An OS abstraction implementation is provided for the Windows XP OS platform.

4.1.20 Demonstration and Test Code

A sample Windows console application is provided with the EPL development kit. This not only provides a test bench for EPL but can aid a developer in understanding how the various EPL functions operate and how to interpret results, etc.

4.1.21 Python Interface Layer and Example Script

The EPL library contains wrapper code that enables the use of the library from Python. Together with the provided Python code this provides for quick and easy prototyping of the library functionality.

4.2 EPL Architecture and Design Overview

This section provides an overview of the EPL design and details of its implementation.

4.2.1 High Level Design Goals

The following design goals helped shape the implementation of the library:

4.2.1.1 Modular

The EPL software is structured so that it is straightforward to include only the functional areas that are actually in use by higher layer software components. Each main functional area is implemented in its own source file. Cross source file dependencies are minimized.

4.2.1.2 Portability

EPL was designed from the ground up to be as portable as possible. While EPL has only been verified operational on a Windows based platform, it has been designed to ease the porting effort to other environments by separating the OS specific functionality into a separate module.

4.2.1.3 Mutual Exclusion

All EPL functions are reentrant and support multi-threaded environments. This assumption relies on the correct implementation of the OAIBeginMultiCriticalSection and OAIEndMultiCriticalSection OAI functions. Also, EPL functions, except Initialization, Enumeration and Deinitialization, are reentrant if each distinct thread is interacting with different devices on different management interfaces.

4.2.1.4 No "C" library dependencies

To maximize portability, EPL does NOT use any external "C" libraries. No floating point arithmetic is used, thus eliminating dependencies on floating point libraries.

4.2.1.5 Include File Guidelines

A single include file is provided with EPL for use by higher layer software and EPL itself. This file includes a few other files and together provides all the prototypes, data type definitions and general definitions necessary for interfacing to the EPL. All client modules of EPL should include **epl.h**.

```
#include <epl.h>    // Base EPL definitions (required)
```

The **epl.h** include file includes a number of other files for each of the various modules. Access to each of those files from the build environment is necessary to build the library and test application(s). This represents a compromise between having a huge all-in-one header file and having a modular code base where items can be added and removed as needed. Since EPL is intended to be modified and customized for a specific implementation this is an acceptable compromise. If a single include file is needed for a particular use it can easily be created by combining the necessary individual files into a custom **epl.h** for that specific platform.

A **platform.h** file is provided that defines the platform specific definitions that EPL depends on. This file may be customized for a target environment, if needed.

4.2.1.6 Well Known Data Types

The following data types are defined and must be customized for each environment, as needed. They are defined in **epl_types.h** file and can be modified as needed for a particular environment.

```
// Note: On platforms where the natural integer size is less then 32-bits
// in size (eg 16-bit platforms), NS_UINT and NS_SINT must be defined as a
// data type no less than 32-bits in size.
typedef void                NS_VOID;
typedef unsigned int        NS_UINT;    // unsigned variable sized
typedef int                 NS_SINT;    // signed variable sized
typedef unsigned char       NS_UINT8;   // unsigned 8-bit fixed
typedef char                NS_SINT8;   // signed 8-bit fixed
typedef unsigned short int  NS_UINT16;  // unsigned 16-bit fixed
typedef short int           NS_SINT16;  // signed 16-bit fixed
typedef unsigned long int   NS_UINT32;  // unsigned 32-bit fixed
typedef long int            NS_SINT32;  // signed 32-bit fixed
typedef unsigned char       NS_CHAR;
typedef NS_UINT             NS_BOOL;    // TRUE or FALSE
```

4.2.1.7 Standardized Status Return Codes

All EPL functions returning a status back to the caller will be of type *NS_STATUS*, which is as defined below. The return codes applicable to a function are noted in the API description man-page.

```
typedef enum
{
        NS_STATUS_SUCCESS,       // The request operation completed successfully
        NS_STATUS_FAILURE,       // The operation failed
        NS_STATUS_INVALID_PARM,  // An invalid parameter was detected
        NS_STATUS_RESOURCES,     // Failed to allocate resources required
        NS_STATUS_NOT_SUPPORTED, // Operation not supported
        NS_STATUS_ABORTED,       // Operation was interrupted before completion
        NS_STATUS_HARDWARE_FAILURE  // Unexpected hardware error encountered
} NS_STATUS;
```

4.2.2 EPL Code Structure

The EPL code is split into the following directory structure:

```
├───core
├───interface
│   ├───ALP_OK
│   ├───CyUSB
│   └───LPT
├───protocol
│   └───PTP
│       └───PTPStack
│           └───dep
├───OS
│   └───Windows
│       ├───EPLTestApp
│       │   ├───Debug
│       │   └───Release
│       ├───PTPTestApp
│       │   ├───Debug
│       │   └───Release
│       └───phyter1588
│           ├───Debug
│           └───Release
└───tools
    ├───python
    │   ├───EPLTest
    │   └───testscripts
    └───swig
        └───Lib
            ├───python
            ├───std
            ├───typemaps
            └───xml
```

4.2.2.1 core

This directory contains files that implement the core EPL functionality. The majority of the externally visible functions are implemented in this directory. Specific files include:

- **epl.h** – The main header file
- **epl_regs.h** – This defines all of the devices specific registers
- **epl_types.h** – This defines the standard datatypes that are used throughout the library
- **epl_core.c/.h** – These files define and implement the core library functions
- **epl_link.c/.h** – These files define and implement the link related functions
- **epl_bist.c/.h** – These files define and implement the BIST related functions
- **epl_miiconfig.c/.h** – These files define and implement the MII configuration related functions
- **epl_quality.c/.h** – These files define and implement the link quality related functions
- **epl_tdr.c/.h** – These files define and implement the TDR related functions
- **swig_help.h** – This file provides some definitions to help in the generation of the Python interface

4.2.2.2 interface

This directory contains code related to the hardware interfaces that are used to communicate with the device(s). The code in this directory is not normally called from outside the library. Only the core read and write routines call into the interface layer. This allows the library to be used in a consistent manner regardless of the underlying hardware. Currently code for the following interfaces is provided:

4.2.2.2.1 ALP_OK

This directory contains the necessary source and binary files to interface with the ALP 100 board using the Opal Kelly Front Panel interface. The **okMAC.c/.h** files provide the wrapper for EPL to communicate through the **okFrontPanel.dll** interface.

4.2.2.2.2 CyUSB

This directory contains the necessary source and binary files to interface with the ALP Nano board using the Cypress USB interface. Since the hardware only supports the MDIO interface the only functionality provided through this interface is what is available using PHY register reads and writes. The **ifCyUSB.cpp/.h** files provide the wrapper for EPL to communicate through the **CyAPI.lib** interface.

4.2.2.2.3 LPT

This directory contains the necessary source for implementing an MDIO bit banging interface on an LPT port. Depending on the OS environment the use of this code requires additional driver support in order to be able to access the LPT port directly. **The use of this interface is discouraged and provided only as an example of what could be done if other options are not available.**

4.2.2.3 protocol

This directory contains code to support and implement specific protocols from within EPL. Currently only the IEEE 1588 Precision Time Protocol (PTP) v1 is supported. This directory contains the following directories:

4.2.2.3.1 PTP

This directory contains all things PTP. At this level only 2 sets of files exist to form the bridge between EPL and PTP:

- **epl_1588.c/.h** – These files implement the externally visible 1588 functions provided by the library
- **ptpControl.c/.h** – These file define and implement the structures and functions necessary to interface with the **PTPStack**
- **PTPStack** – This is a complete PTP v1 stack. See below for additional detail.

4.2.2.3.2 PTPStack (and subdirectories)

This directory contains a full PTP v1 stack. It was derived from an open source project found at: http://sourceforge.net/projects/ptpd/ The code has been tweaked to operate as part of the EPL binary.
NOTE: The inclusion of this stack is for demonstration and evaluation purposes only. There is no guarantee that the functionality provided is complete or accurate.

4.2.2.4 OS

This directory contains code to support and implement specific operating systems. Currently the only supported OS is Windows. This directory contains the following directories:

4.2.2.4.1 Windows

This directory contains the version of the **oai.c/.h** that implements Windows specific operations that are necessary for EPL to operate correctly. This is also the directory work the final binaries for a specific implementation are created. The following directories are found here:

4.2.2.4.1.1 phyter1588

This directory contains the project files necessary to build the library deliverable **_epl.dll**. For details on the project build see **The EPL Build Environment** section below. In addition the build project files there are some source files that exist at this level:

- **nscphyter1588.cpp** – This file implements the Windows dll initialization code.
- **platform.h** – This file defines some platform specific structures and data that are used to shape the operation of the library
- **epl.i/buildpython.cmd** – These files are used by SWIG to generate the Python interface for the library
- **epl.py/epl_wrap.c** – These files are generated by SWIG and make up the Python interface
- **nscphyter1588.sln/.vcproj** – These files are the Visual Studio project files used to make the library

4.2.2.4.1.2 EPLTestApp

This directory contains the source and project files necessary to build a Windows console application that can be used to exercise and test the library. Refer to section 5 for additional detail about this example. **NOTE: The test is intended to exercise the API only. They do not necessarily indicate recommended values or sequences that would be used in a production system.**

4.2.2.4.1.3 PTPTestApp

This directory contains the source and project files necessary to build a Windows console application that can be used to exercise and test the PTP operations of the library. Refer to section 5 for additional detail about this example. **NOTE: The test is intended to provide a complete working example of the setup and operation of the PTP functionality. It does not necessarily indicate recommended values or sequences that would be used in a production system for optimal performance.**

4.2.2.5 tools

This directory contains tools used to create and or test the library:

4.2.2.5.1 python\EPLTest

This directory contains a test script that can be used from Python to exercise/test the library. It roughly follows the functionality available in the EPLTestApp. **NOTE: The test scripts are intended to exercise the API only. They do not necessarily indicate recommended values or sequences that would be used in a production system.**

4.2.2.5.2 python\testscripts

This directory contains test scripts that can be used from Python to exercise/test the library. **NOTE: The test scripts are intended to exercise the API only. They do not necessarily indicate recommended values or sequences that would be used in a production system.**

4.2.2.5.3 swig

This directory and its subdirectories contain the SWIG tool that is used to generate the Python interface for the library.

4.2.3 The EPL Build Environment

Currently the library and associated programs are built using the following tools:

4.2.3.1 Windows

The following tools are used to build the Windows based tools:

- **Microsoft Visual Studio 2005**

```
Microsoft Visual Studio Microsoft Visual Studio 2005
Version 8.0.50727.42  (RTM.050727-4200)
Microsoft .NET Framework
Version 2.0.50727 SP1
```

- **SWIG (Simple Wrapper Interface Generator)**

```
SWIG Version 1.3.31
Compiled with g++ [i686-pc-mingw32]
Please see http://www.swig.org for reporting bugs and further information
```

4.2.4 Customizing and Porting EPL

As noted earlier it is expected that EPL will not be used exactly as is in a real system. The following provides an overview of how and where most of the modifications may be needed:

4.2.4.1 _epl.dll Build Changes

The most common modification would be to modify the build to add or remove files based on the specific needs of the platform. For example, most real world systems will probably not include hardware that is compatible with the ALP 100/Opal Kelly interface so that code could be excluded from the build. See below for more on creating a new interface module below. You may also want to customize some of the compiler flags based on your specific needs

4.2.4.2 Platform Specific Changes

One of the most common locations to change is **platform.h** level. This file is at the project level and combines information specific to the current implementation. The OAI_DEV_HANDLE_STRUCTURE contains specific members that may or may not be needed in any particular design. If you are implementing a new interface or OS you'd want to add supporting data members here.

4.2.4.3 New Interface Support

As noted above it is very likely that you will need to create a new interface module that supports your specific hardware. To do this you can start by copying one of the existing modules to a new directory and filling in each of the functions with code that works for your specific hardware.

4.2.4.4 New OS Support

Since the only OS that is currently supported is Windows, it is likely that a new OS implementation will need to be created. To do this you'd start by copying the Windows directory over to a new directory under the OS branch. Then each of the files will need to be modified to implement the proper behavior for the new target OS. This includes the following major components:

- **oai.c/.h** – Each of the operations needs to be created to work correctly in the new environment
- **nscphter1588.cpp** – This a Windows DLL specific piece of code so it will need to be replaced with a new file that matches the target OS.
- **build tools** – It is likely that the new OS doesn't support Visual Studio so the build system will need to be created to compile and build a proper binary for the target OS
- **Other –** While every attempt has been made to removed OS specific operations/functionality from the rest of the code it is likely that you will need to make specific tweaks in order to produce working code.

4.2.4.5 Processor/Target Specific Changes

EPL has been developed and tested primarily on an x86 system. While there is very little code that is x86 specific, there are several areas to watch out for if you attempt to port the code to a different architecture:

- Data type sizes – Need to make sure that the basic data types defined in **epl_types.h** result in variables that are big enough to hold the expected data.
- Endianness – x86 is a little endian architecture where the data that is larger than a 8 bits is stored in memory with the least significant byte first. Other platforms are big endian where the most significant byte is stored first. Care must be taken if implementing this code on a big endian platform to ensure that all data is properly handled.

4.3 EPL Core Function Reference

This section provides a reference for the EPL core functionality

4.3.1 General APIs

4.3.1.1 EPLInitialize

Called to initialize the EPL library.

```
NS_STATUS
        EPLInitialize(
                void);
```

Parameters

None

Return Value

NS_STATUS_SUCCESS
EPL initialization was successful.
NS_STATUS_FAILURE
EPL initialization failed, usually due to a memory allocation failure.

Comments

This function must be called prior to calling any other EPL functions. It initializes EPL's internal state. EPLDeinitialize should be called after all interaction with EPL has finished.

4.3.1.2 EPLDeinitialize

Called to de-initialize the EPL library.

```
void
        EPLDeinitialize(
                void);
```

Parameters

None

Return Value

None

Comments

This function must be called after all interaction with EPL has finished. This function will free any memory that the library allocated during operation. The application is responsible for calling OAIFree() to free up any memory that it allocates using the OAIAlloc().

4.3.1.3 EPLEnumDevice

Enumerate a PHYTER device on the specified MDIO bus and returns a deviceHandle object.

```
PEPL_DEV_HANDLE
        EPLEnumDevice(
                IN OAI_DEV_HANDLE oaiDevHandle,
                IN NS_UINT deviceMdioAddress,
                IN EPL_ENUM_TYPE enumType);
```

Parameters

oaiDevHandle

Handle that represents the MDIO bus that the error occurred on. The definition of this is completely up to higher layer software.

deviceMdioAddress

Specifies the device address on the MDIO bus to try and enumerate. To enumerate all devices, start by calling this function with an address of 0, and continue to increment the value up to the maximum MDIO address (typically 31). The increment amount should be the number of ports the device contains.

enumType

This EPL_ENUM_TYPE (defined in platform.h) indicates the type of access functions will be used during the enumeration for register accesses. Options are:

- EPL_ENUM_MDIO_BIT_BANG – This will cause the OAIMdioReadBit function will be used.
- EPL_ENUM_DIRECT – This will cause the OAIDirReadReg function will be used.
- EPL_ENUM_CYUSB_MDIO – This will use the ifCyUSB_ReadMDIO function.

Return Value

Returns NULL if no device was found at the specified MDIO device address, otherwise an opaque handle is returned representing the device (deviceHandle). This handle can then be used in other EPL functions to query and interact with the device.

Comments

This function will only match devices that contain National's OUI value. The device can contain any device ID. A device can contain one or more ports. To interact with a port you must use the EPLEnumPort function to obtain the necessary port objects.

4.3.1.4 EPLGetDeviceInfo

Returns descriptive information about the specified device.

```
PEPL_DEV_INFO
        EPLGetDeviceInfo(
                IN PEPL_DEV_HANDLE deviceHandle);
```

Parameters

deviceHandle

Handle that represents the device. This is obtained using the EPLEnumDevice function.

Return Value

A pointer to a EPL_DEV_INFO structure (see below).

Comments

Field Name	**Data Type**	**Description**
deviceType	EPL_DEVICE_TYPE_ENUM	Refer to the definition EPL_DEVICE_TYPE_ENUM for a list of defined device types.
numOfPorts	NS_UINT	The number of PHY Ethernet ports on this device.
deviceModelNum	NS_UINT	The device's silicon model number.
deviceRevision	NS_UINT	The devices' silicon revision number.
numExtRegisterPages	NS_UINT	The number of extended register pages supported by the device.

Table 4.3-1 – EPL_DEV_INFO

4.3.1.5 EPLIsDeviceCapable

Returns whether or not the specified device implements a particular feature.

```
NS_BOOL
        EPLIsDeviceCapable(
                IN PEPL_DEV_HANDLE deviceHandle,
                IN EPL_DEVICE_CAPA_ENUM capability);
```

Parameters

deviceHandle
Handle that represents the device. This is obtained using the EPLEnumDevice function.
Capability
One of the defined values in EPL_DEVICE_CAPA_ENUM (see below).

Return Value

TRUE if the device supports the specified feature, FALSE otherwise.

Comments

This function provides a method to write run-time selected device specific code.

Capability	**Description**
EPL_CAPA_NONE	Device doesn't support any features.
EPL_CAPA_TDR	Device supports TDR features.
EPL_CAPA_LINK_QUALITY	Device supports Link Quality features.
EPL_CAPA_MII_PORT_CFG	Device supports MII port configuration feature.
EPL_CAPA_MII_REG_ACCESS	Device supports PHY Control Frame (PCF) register read/write feature.

Table 4.3-2 – EPL_DEVICE_CAPA_ENUM

4.3.1.6 EPLResetDevice

Resets the device.

```
void
        EPLResetDevice(
                IN PEPL_DEV_HANDLE deviceHandle);
```

Parameters

deviceHandle
Handle that represents the device. This is obtained using the EPLEnumDevice function.

Return Value

None

Comments

For multi-port devices this will cause the device as well as all of its ports to be reset. For single port devices this has the same affect as calling the ResetPort() method. This method does NOT return until the reset operation has completed (approximately 1.2ms).

NOTE: If the PHY Control and Status Frames are used to access the PHY registers calling this function will reset the device configuration and destroy the PHY Status Frame configuration. This will cause PHY register reads to stop working. It is recommended that if a device reset is needed that the entire library be shutdown by calling EPLDeinitialize() and restarted by calling EPLInitialize() to ensure that everything is shutdown cleanly and reconfigured properly.

4.3.1.7 EPLSetMgmtInterfaceConfig

Used to configure EPL to use PHY Control Frames (PCF) or MDIO to read and write PHY registers.

```
void
        EPLSetMgmtInterfaceConfig (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_BOOL usePhyControlFrames);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

usePhyControlFrames
Set to TRUE to configure EPL to use the PCF mechanism for PHY register reads and writes. Set to FALSE to use MDIO (default).

Return Value

None

Comments

None

4.3.1.8 EPLGetMiiConfig

Returns the current MII configuration.

```
EPL_MIICFG_ENUM
        EPLGetMiiConfig(
                IN PEPL_DEV_HANDLE deviceHandle);
```

Parameters

deviceHandle

Handle that represents the device. This is obtained using the EPLEnumDevice function.

Return Value

A value from EPLMII_CFG_ENUM (see below).

Comments

On devices supporting the EPL_CAPA_MII_PORT_CFG feature this method returns the device's current MII port mapping configuration. If the device configuration is undefined, MIIPCFG_UNKNOWN will be returned.

MIIPCFG_UNKNOWN	The MII configuration is set to a non-standard mode.
MIIPCFG_NORMAL	Normal mode. Port A and B are configured for straight through operation.
MIIPCFG_PORT_SWAP	Ports A and B are swapped.
MIIPCFG_EXT_MEDIA_CONVERTER	Repeater mode. Rx from Port A is connected to Port B Tx and PortB’s Rx is connect to Port A’s Tx.
MIIPCFG_BROADCAST_TX_PORT_A	Port A Tx is sent on Port A and Port B Tx channels.
MIIPCFG_BROADCAST_TX_PORT_B	Port B Tx is sent on Port A and Port B Tx channels.
MIIPCFG_MIRROR_RX_CHANNEL_A	Port A’s Rx channel feeds both Port A and B’s Rx MII.
MIIPCFG_MIRROR_RX_CHANNEL_B	Port B’s Rx channel feeds both Port A and B’s Rx MII.
MIIPCFG_DISABLE_PORT_A	Port A is disabled.
MIIPCFG_DISABLE_PORT_B	Port B is disabled.

Table 4.3-3 – EPL_MIICFG_ENUM

4.3.1.9 EPLSetMiiConfig

Configures the device’s MII configuration.

```
void
        EPLSetMiiConfig(
                IN PEPL_DEV_HANDLE deviceHandle,
                IN EPL_MIICFG_ENUM miiPortConfig);
```

Parameters

deviceHandle
: Handle that represents the device. This is obtained using the EPLEnumDevice function.

miiPortConfig
: One of the values defined in EPL_MIICFG_ENUM (see EPLGetMiiPortConfig). This parameter should NOT be set to MIIPCFG_UNKNOWN.

Return Value

None

Comments

On devices supporting the EPL_CAPA_MII_PORT_CFG feature this method sets the device's MII port mapping configuration.

4.3.1.10 EPLEnumPort

Enumerate a device's ports.

```
PEPL_PORT_HANDLE
        EPLEnumPort(
                IN PEPL_DEV_HANDLE deviceHandle,
                IN NS_UINT portIndex);
```

Parameters

deviceHandle
: Handle that represents the device. This is obtained using the EPLEnumDevice function.

portIndex
: Specifies the enumeration port index. To enumerate all ports on a device, start by calling this function with an index of 0, and continue to increment the value up to the number of ports the devices has, minus one. The number of ports can be obtained by calling EPLGetDeviceInfo().

Return Value

Returns NULL if the specified portIndex is invalid, otherwise an opaque handle is returned representing the port (portHandle). This handle can then be used in other EPL functions to query and interact with a particular port on a device.

Comments

None

4.3.1.11 EPLGetDeviceHandle

Given a port handle, returns the associated (parent) device handle.

```
PEPL_DEV_HANDLE
        EPLGetDeviceHandle(
                IN PEPL_PORT_HANDLE portHandle);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

Return Value

Returns the device handle that is associated (parent) of the specified port.

Comments

None

4.3.1.12 EPLReadReg

Reads the contents of the specified port register.

```
NS_UINT
    EPLReadReg(
        IN PEPL_PORT_HANDLE portHandle,
        IN NS_UINT registerIndex);
```

Parameters

portHandle
: Handle that represents a port. This is obtained using the EPLEnumPort function.

registerIndex
: Index of the register to read. Bits 7:5 select the register page (000-pg0, 001-pg1, 010-pg3, 011-pg4, etc.).

Return Value

Value read from the register.

Comments

Refer to the datasheet and **epl_regs.h** file for register definitions.

4.3.1.13 EPLWriteReg

Writes a value to the specified port register.

```
void
        EPLWriteReg(
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT registerIndex,
                IN NS_UINT value);
```

Parameters

portHandle
: Handle that represents a port. This is obtained using the EPLEnumPort function.

registerIndex
: Index of the register to write. Bits 7:5 select the register page (000-pg0, 001-pg1, 010-pg3, 011-pg4, etc.).

value
: The value to write to the register (0x0000 – 0xFFFF).

Return Value

None

Comments

Refer to the datasheet and **epl_regs.h** file for register definitions.

4.3.1.14 EPLGetPortMdioAddress

Returns a port’s MDIO bus address.

```
NS_UINT
        EPLGetPortMdioAddress(
                IN PEPL_PORT_HANDLE portHandle);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

Return Value

Returns a port’s MDIO bus address.

Comments

None

4.3.1.15 EPLSetPortPowerMode

Controls the power to a specified port.

```
void
    EPLSetPortPowerMode(
        IN PEPL_PORT_HANDLE portHandle,
        IN NS_BOOL powerOn);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

powerOn
Set to TRUE to enable power to the port, FALSE to disable power.

Return Value

None

Comments

If the port is powered down, the device is still accessible through the management interface, i.e. this library (registers).

NOTE: If the PHY Control and Status Frames are used to access the PHY registers calling this function will power down the device and destroy the PHY Status Frame configuration. This will cause PHY register reads to stop working. It is not recommended that this function be used in this case except for the situation where the device is no longer needed. If the device is needed after it is powered down the entire library should be shutdown by calling EPLDeinitialize() and restarted by calling EPLInitialize() to ensure that everything is shutdown cleanly and reconfigured properly.

4.3.2 Link Related APIs

4.3.2.1 EPLIsLinkUp

Returns whether or not a valid link exists on the specified port.

```
NS_BOOL
        EPLIsLinkUp (
                IN PEPL_PORT_HANDLE portHandle);
```

Parameters

portHandle

Handle that represents a port. This is obtained using the EPLEnumPort function.

Return Value

TRUE if a valid link exists on the specified port, FALSE otherwise.

Comments

None

4.3.2.2 EPLGetLinkStatus

Returns detailed information regarding the port’s link status.

```
void
        EPLGetLinkStatus (
                IN PEPL_PORT_HANDLE portHandle,
                IN OUT PEPL_LINK_STS linkStatusStruct);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.
linkStatusStruct
Pointer to a caller supplied EPL_LINK_STS (see below) structure that is filled out on return.

Return Value

The fields in the passed in linkStatusStruct will be set accordingly on return.

Comments

Field Name	**Data Type**	**Description**
linkup	NS_BOOL	Set to TRUE if link is currently established, FALSE otherwise.
autoNegEnabled	NS_BOOL	Set to TRUE if auto-negotiation is enabled, FALSE otherwise.
autoNegCompleted	NS_BOOL	Set to TRUE if the auto-negotiation process has completed, FALSE if auto-negotiation is in progress or NOT enabled.
speed	NS_UINT	10 or 100 representing 10 Mbps or 100 Mbps respectively.
duplex	NS_BOOL	Set to TRUE if link is in full-duplex mode, FALSE if it’s in half-duplex mode.
mdixStatus	NS_BOOL	Set to TRUE if pairs are swapped, FALSE otherwise.
autoMdixEnabled	NS_BOOL	Set to TRUE if the auto-MDIX feature is enabled, FALSE if its disabled.
polarity	NS_BOOL	Set to TRUE if an inverted pair polarity was detected, FALSE otherwise.
energyDetectPower	NS_BOOL	Set to TRUE if the energy detect indicates a power up state, FALSE indicates the port is in low-power mode.

Table 4.3-4 – EPL_LINK_STS

4.3.2.3 EPLSetLinkConfig

Configures the port’s link settings.

```
void
        EPLSetLinkConfig (
                IN PEPL_PORT_HANDLE portHandle,
                IN PEPL_LINK_CFG linkConfigStruct);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.
linkConfigStruct
Pointer to a caller supplied EPL_LINK_CFG (see below) structure that specifies link configuration parameters.

Return Value

None

Comments

This function does NOT wait for link establishment to complete (e.g. auto-negotiation).

Field Name	Data Type	Description
autoNegEnable	NS_BOOL	Set to TRUE if auto-negotiation should be enabled, FALSE otherwise (forced mode). In forced mode, the link will be set to the specified speed and duplex configuration. If auto-negotiation mode is enabled, the speed and duplex settings define the best / maximum capabilities of the local PHY.
Speed	NS_UINT	10 or 100 representing 10 Mbps or 100 Mbps respectively.
Duplex	NS_BOOL	Set to TRUE for full-duplex mode, FALSE for half-duplex mode.
pause	NS_BOOL	Set to TRUE to advertise to the link partner that this port supports PAUSE, FALSE otherwise.
autoMdix	EPL_MDIX_ENUM	Set to one of the values defined in EPL_MDIX_ENUM.
energyDetect	NS_BOOL	Set to TRUE to enable the port’s energy detect feature, FALSE to disable it.
energyDetectErrCountThresh	NS_UINT	Threshold to determine the number of energy detect error events that will cause the device to take action. Default is 1. Range is 1 – 15.
energyDetectDataCountThresh	NS_UINT	Threshold to determine the number of energy detect data events that will cause the device to take action. Default is 1. Range is 1 – 15.

Table 4.3-5 – EPL_LINK_CFG

MDIX_AUTO	Automatic MDIX mode.
MDIX_FORCE_NORMAL	Forces MDIX pairs normal (no swap).
MDIX_FORCE_SWAP	Forces MDIX pairs swapped.

Table 4.3-6 – EPL_MDIX_ENUM

4.3.2.4 EPLRestartAutoNeg

Restarts auto-negotiation.

```
void
        EPLRestartAutoNeg (
                IN PEPL_PORT_HANDLE portHandle);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

Return Value

None

Comments

This function does NOT wait for auto-negotiation to finish.

4.3.3 BIST Related APIs

4.3.3.1 EPLSetLoopbackMode

Enables or disables port loopback mode.

```
void
        EPLSetLoopbackMode (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_BOOL enableLoopback);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.
enableLoopback
Set to TRUE to enable port loopback, FALSE otherwise.

Return Value

None

Comments

None

4.3.3.2 EPLBistStartTxTest

Starts Transmit Built-in Self Test (BIST).

```
void
        EPLBistStartTxTest (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_BOOL psr15Flag);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

psr15Flag
Set to TRUE to use 15-bit pseudo random data, FALSE to use 9-bit pseudo random data.

Return Value

None

Comments

NOTE: If the PHY Control and Status Frames are used to access the PHY registers calling this function will prevent access to the registers. It is not recommended that this function be used in this case as there is no way to read the status after it has been started.

4.3.3.3 EPLBistStopTxTest

Halts a previously started BIST test.

```
void
        EPLBistStopTxTest (
                IN PEPL_PORT_HANDLE portHandle);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

Return Value

None

Comments

None

4.3.3.4 EPLBistGetStatus

Returns BIST status information.

```
void
        EPLBistGetStatus (
                IN PEPL_PORT_HANDLE portHandle,
                IN OUT NS_BOOL *bistActiveFlag,
                IN OUT NS_UINT *errDataNibbleCount,
                IN OUT NS_BOOL *receiveDataDetectedFlag);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

bistActiveFlag
Pointer to a Boolean variable that will be set on return to TRUE if transmit BIST is currently enabled, FALSE otherwise.

errDataNibbleCount
Pointer to a NS_UINT variable that will be set on return to the number of errored data nibbles dectected so far.

receiveDataDetectedFlag
Set to TRUE on return if receive BIST traffic has been detected, FALSE otherwise. Only available on the DP83640, DP83630 and DP83620 devices.

Return Value

bistActiveFlag and errDataNibbleCount are set on return.

Comments

None

4.3.4 Cable & Link Quality Related APIs

4.3.4.1 EPLGetCableStatus

Returns various cable status values.

```
NS_STATUS
        EPLGetCableStatus (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT sampleTime,
                IN OUT NS_UINT *cableLength,
                IN OUT NS_SINT *freqOffsetValue,
                IN OUT NS_SINT *freqControlValue,
                IN OUT NS_UINT *varianceValue);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

sampleTime
Defines the length of time in milliseconds that the hardware will sample SNR / variance data. Valid values are: 2, 4, 6 or 8.

cableLength
Pointer to a NS_UINT variable that will be set on return to the estimated length of the cable in meters.

freqOffsetValue
Frequency offset value (see data sheet for details). To obtain frequency offset in ppm you must multiply this value by 5.1562. (Signed value).

freqControlValue
Frequency control value (see data sheet for details). To obtain frequency control in ppm you must multiply this value by 5.1562. (Signed value).

varianceValue
This is the raw variance value obtained from the hardware after sampling for the indicated length of time (sampleTime). It can be used to calculate an SNR value that indicates the quality of the link.

Return Value

NS_STATUS_SUCCESS
Function was successfully executed.

NS_STATUS_FAILURE
Link NOT established or link is NOT at 100 Mbps.

Comments

The device must have the EPL_CAPA_TDR capability to use this function. A call to EPLIsDeviceCapable() can be used to determine if this operation is supported.

EPLInitTDR() and EPLDeinitTDR() should NOT be called when using this function.

A valid 100 Mbps link must be established before calling this function.

The freqOffsetValue and freqControlValue values can be used to calculate jitter in ppm. To do this you must first multiply the returned frequency offset and control values by 5.1562, then take the abs function of the difference between frequency control and frequency offset (e.g. abs(freqControl – freqOffset)).

The SNR can be calculated from the returned varianceValue using the following formula:
varData = (288.0 * ((1024 * 1024 * sampleTime) / 8.0)) / float(varianceValue)
rxSNR = 10.0 * math.log10(varData)

4.3.4.2 EPLGetTDRPulseShape

This procedure uses the TDR feature to obtain an oscilloscope trace of the TDR pulse on the wire.

```
NS_STATUS
    EPLGetTDRPulseShape (
        IN PEPL_PORT_HANDLE portHandle,
        IN NS_BOOL useTxChannel,
        IN NS_BOOL use50nsPulse,
        IN OUT NS_SINT8 *positivePulseResults,
        IN OUT NS_SINT8 *negativePulseResults);
```

Parameters

portHandle
: Handle that represents a port. This is obtained using the EPLEnumPort function.

useTxChannel
: If TRUE a trace will be obtained for the Tx cable pair, if FALSE the trace will be for the Rx cable pair.

use50nsPulse
: If TRUE a 50ns electrical pulse will be used to obtain the trace. If FALSE an 8ns pulse will be used. The negativePulseResults buffer can be NULL when using the 50ns option because the device does NOT support sending a negative 50ns pulse.

positivePulseResults
: This must be at least a 256 byte array that will have its contents set with the sample data that describes the TDR pulse trace that was obtained by sending positive TDR pulses. The values can range from -32 to +32 and represent the relative power level measured at each sampling point. The value at index 0 in the array represents time 0, index 1 at time 8ns, up to index 255 that represents 2.040us.

negativePulseResults
: This must be at least a 256 byte array that will have its contents set with the sample data that describes the TDR pulse trace that was obtained by sending negative TDR pulses. The values can range from -32 to +32 and represent the relative power level measured at each sampling point. The value at index 0 in the array represents time 0, index 1 at time 8ns, up to index 255 that represents 2.040us. This array will NOT be set if use50nsPulse is TRUE.

Return Value

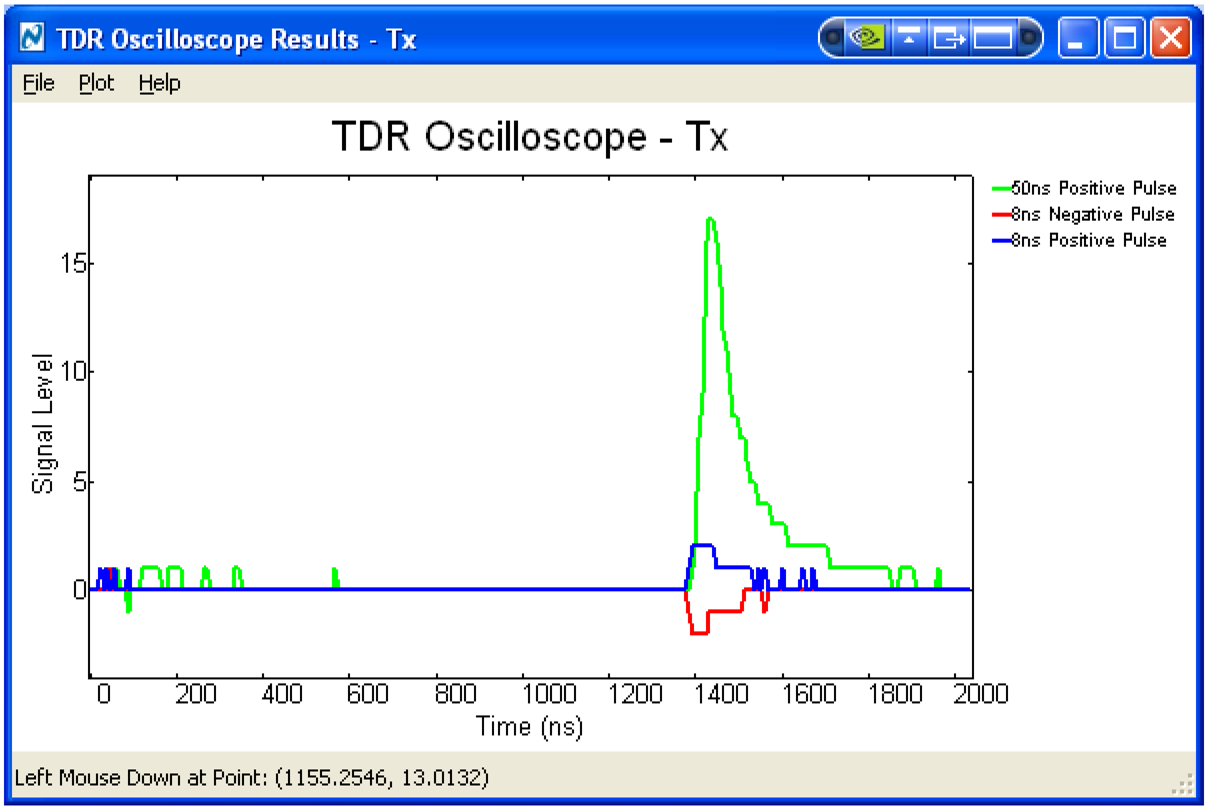
NS_STATUS_SUCCESS
: Function was successfully executed.

Comments

The device must have the EPL_CAPA_TDR capability to use this function. A call to EPLIsDeviceCapable() can be used to determine if this operation is supported.

EPLInitTDR() and EPLDeinitTDR() should NOT be called when using this function.

An example of how the data can be used to graph the pulse is shown below:



Left Mouse Down at Point: (1155.2546, 13.0132)

NOTE: Since the data is only sampled every 8ns, the shape of the curve is only an approximation of the actual signal. That is, some data may not be captured.

4.3.4.3 EPLGetTDRCableInfo

Attempts to determine the length and status of the specified channel on this port.

```
NS_STATUS
    EPLGetTDRCableInfo(
        IN PEPL_PORT_HANDLE portHandle,
        IN NS_BOOL useTxChannel,
        IN OUT EPL_CABLE_STS_ENUM *cableStatus,
        IN OUT NS_UINT *rawCableLength);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

useTxChannel
If TRUE information will be obtained for the Tx cable pair, if FALSE the information will be for the Rx cable pair.

cableStatus
Set on return to a value that indicates the status of the selected wire pair (see enum definitions below).

rawCableLength
Set on return to a raw value indicating the approximate length of the cable or distance to fault. If the cableStatus indicates that the cable is properly terminated then this value is NOT valid. This value must be post processed to obtain the length in meters (see Comments below for formula).

Return Value

NS_STATUS_SUCCESS – If determination was successful.
NS_STATUS_RESOURCES – If memory allocation failure or if call to EPLGatherTDRInfo() fails.

Comments

The device must have the EPL_CAPA_TDR capability to use this function. A call to EPLIsDeviceCapable() can be used to determine if this operation is supported.

EPLInitTDR() and EPLDeinitTDR() should NOT be called when using this function.

To calculate the length of the cable or distance to a fault you can use the following formula:
length = rawCableLength / TDR_CABLE_VELOCITY / 2
where TDR_CABLE_VELOCITY is defined as 4.64. The length value will be in meters.

CABLE_STS_TERMINATED	Indicates that the wire pair is properly terminated with a link partner. rawCableLength is NOT valid in this case.
CABLE_STS_OPEN	The cable pair is open, not connected or shorted.
CABLE_STS_SHORT	The cable pair is shorted together.
CABLE_STS_CROSS_SHORTED	The cable pair is shorted with the other cable pair.
CABLE_STS_UNKNOWN	The function was unable to determine the cable status.

Table 4.3-7 – EPL_CABLE_STS_ENUM

4.3.4.4 EPLGatherTDRInfo

General purpose routine that runs all useful TDR test configurations and returns the results.

```
NS_BOOL
        EPLGatherTDRInfo(
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_BOOL useTxChannel,
                IN NS_BOOL useTxReflectChannel,
                IN NS_UINT thresholdAdjustConstant,
                IN NS_BOOL stopAfterSuccess,
                IN OUT NS_UINT *baseline,
                IN OUT PTDR_RUN_RESULTS posResultsArrayNoInvert,
                IN OUT PTDR_RUN_RESULTS negResultsArrayNoInvert,
                IN OUT PTDR_RUN_RESULTS posResultsArrayInvert,
                IN OUT PTDR_RUN_RESULTS negResultsArrayInvert);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

useTxChannel
Specifies the send channel. Set to TRUE to use Tx channel, FALSE for the Rx channel.

useTxReflectChannel
Specifies the reflect channel. Set to TRUE to use Tx channel, FALSE for the Rx channel.

thresholdAdjustConstant
Defines the integer value that will be added to and substracted from the baseline line levels to determine appropriate transmit and receive thresholds.

stopAfterSuccess
If set to TRUE all configurations are run until a successful reflection (both pos and neg) is obtained, or a pos reflection is obtained using the 10Mbps block. Timings are increased from lowest to highest. If set to FALSE all possible configurations are run with all possible timings.

baseline
Set on return to the quiescent baseline measurement of the wire pair. If NULL this parameter will be ignored.

posResultsArrayNoInvert[10]
negResultsArrayNoInvert[8]
posResultsArrayInvert[8]
negResultsArrayInvert[8]
These are arrays of TDR_RUN_RESULTS structures. These will be set on return with the results of running 8 TDR pulses from 8ns to 64ns (8ns step), followed by one 50ns and one 100ns pulse result (posResults only). If both pos and neg thresholds are met or the 50ns or 100ns threshold is reached and stopAfterSuccess is set to TRUE, any entries that were not run will be set 0. Invert means that a inverse polarity reflection will be tested for. The 50ns and 100ns positive only pulse results are not present in the negative polarity and inverted test results. Entries that were not run will be NULL'ed out.

Return Value

TRUE if both a positive and negative pulse of the same length met their thresholds. TRUE is also returned if a positive 50ns or 100ns pulse reaches its threshold. FALSE is returned if thresholds were not met.

Comments

General purpose routine that runs all useful TDR test configurations and returns the results. This can also used to obtain a complete characteristic dump of all TDR tests determine and troubleshoot TDR behavior.

The device must have the EPL_CAPA_TDR capability to use this function. A call to EPLIsDeviceCapable() can be used to determine if this operation is supported.

EPLInitTDR() and EPLDeinitTDR() should NOT be called when using this function.

Field Name	**Data Type**	**Description**
thresholdMet	NS_BOOL	Specifies whether or not the TDR threshold valuewas met during the sample window.
thresholdTime	NS_UINT	Specifies the time for the first signal that met the TDR threshold. This value is only valid if TDRThresholdMet is set to True. The value is in ns units. If the threshold was NOT met, this value will be 0.
peakValue	NS_UINT	The peak value measured during the TDR sample window. 0x31 – 0x3F are positive threshold values. 0x00 – 0x30 are negative threshold values with 0x30 indicating -1 and so on.
peakTime	NS_UINT	Specifies the time when the first occurrence of the peak value was detected in ns units.
peakLengthRaw	NS_UINT	A raw calculated cable length based on the TDR peak time.
adjustedPeakLengthRaw	NS_UINT	A calculated cable length based on the TDR peak time. This value has the length of the pulse subtracted off the value and gives a more accurate length value when using the peak value for the length.

Table 4.3-8 – TDR_RUN_RESULTS

4.3.4.5 EPLInitTDR

This procedure must be called prior to using most lower-level measurement TDR functions.

```
NS_UINT
        EPLInitTDR(
                IN PEPL_PORT_HANDLE portHandle);
```

Parameters

portHandle

Handle that represents a port. This is obtained using the EPLEnumPort function.

Return Value

Saved link status (must be passed to EPLDeinitTDR()).

Comments

This must be called prior to using most lower-level measurement TDR functions. This method initializes the TDR engine for subsequent calls to the EPLRunTDR(), EPLShortTDRPulseRun(), EPLLongTDRPulseRun() or EPLMeasureTDRBaseline() methods. EPLDeinitTDR() must be called after all calls have been made to the TDR measurement methods.

4.3.4.6 EPLDeinitTDR

This procedure must be called after all desired calls have been made to most low-level TDR measurement functions.

```
void
        EPLDeinitTDR(
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT savedLinkStatus)
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.
savedLinkStatus
Value that was returned by EPLInitTDR() function.

Return Value

Nothing

Comments

This must be called after all desired calls have been made to most low-level TDR measurement functions. This method restores the PHY to the previous link settings.

4.3.4.7 EPLMeasureTDRBaseline

This measures the baseline signal level for the specified channel.

```
NS_UINT
    EPLMeasureTDRBaseline(
        IN PEPL_PORT_HANDLE portHandle,
        NS_BOOL useTxChannel);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.
useTxChannel
Set to TRUE to measure the baseline for the Tx channel, set to FALSE to measure the Rx channel.

Return Value

Baseline level

Comments

This measures the baseline signal level for the specified channel. This can be used to determine appropriate threshold values by using the baseline values as a mid point.

The device must have the EPL_CAPA_TDR capability to use this function. A call to EPLIsDeviceCapable() can be used to determine if this operation is supported.

EPLInitTDR() and EPLDeinitTDR() must be called when using this function.

4.3.4.8 EPLShortTDRPulseRun

This runs through all useful combinations of TDR operations on the specified channel using the chip's 100Mb 8ns pulse generator.

```
NS_BOOL
    EPLShortTDRPulseRun(
        IN PEPL_PORT_HANDLE portHandle,
        IN NS_BOOL useTxChannel,
        IN NS_BOOL useTxReflectChannel,
        IN NS_UINT posThreshold,
        IN NS_UINT negThreshold,
        IN NS_BOOL noninvertedThreshold,
        IN NS_BOOL stopAfterSuccess,
        IN OUT PTDR_RUN_RESULTS posResultsArray,
        IN OUT PTDR_RUN_RESULTS negResultsArray)
```

Parameters

portHandle
: Handle that represents a port. This is obtained using the EPLEnumPort function.

useTxChannel
: Specifies the send channel. Set to TRUE to use Tx channel, FALSE for the Rx channel.

useTxReflectChannel
: Specifies the reflect channel. Set to TRUE to use Tx channel, FALSE for the Rx channel.

posThreshold : Integer (0x21 - 0x3F)
: Specifies the positive threshold value used when listening for a return TDR pulse.

negThreshold : Integer (0x00 - 0x20)
: Specifies the negative threshold value used when listening for a return TDR pulse.

noninvertedThreshold
: Specifies whether or not an non-inverted polarity threshold and peak detection should be used. For example if set to FALSE, after sending a positive(+) TDR pulse, the chip would be set to detect a negative(-) return pulse. Set to FALSE to detect a channel that is electrically shorted.

stopAfterSuccess
: Specifies whether measurements should stop after a successful threshold condition occurs. If FALSE, all possible timing configurations are run.

posResultsArray
: Array of 8 TDR_RUN_RESULTS structures. These will be set on return with the results of running 8 TDR pulses from time 1 to time 8. These represent the results of positive polarity pulses. If both pos and neg thresholds are met and stopAfterSuccess is set to TRUE, any entries that were not run will be set 0.

negResultsArray
: Array of 8 TDR_RUN_RESULTS structures. These will be set on return with the results of running 8 TDR pulses from time 1 to time 8. These represent the results of negative polarity pulses.

Return Value

TRUE if both positive and negative thresholds were met using the same length of TDR pulse, FALSE otherwise. The positiveResultArray and negativeResultsArray array structures will hold the TDR pulse results.

Comments

This runs through all useful combinations of TDR operations on the specified channel using the chip's 100Mb 8ns pulse generator. The positive and negative threshold values can be specified.

The device must have the EPL_CAPA_TDR capability to use this function. A call to EPLIsDeviceCapable() can be used to determine if this operation is supported.

EPLInitTDR() and EPLDeinitTDR() must be called when using this function.

4.3.4.9 EPLLongTDRPulseRun

This runs through all useful combinations of TDR operations on the specified channel using the chip's 10Mb 50ns pulse generator.

```
NS_BOOL
        EPLLongTDRPulseRun(
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_BOOL useTxChannel,
                IN NS_BOOL useTxReflectChannel,
                IN NS_UINT threshold,
                IN NS_BOOL stopAfterSuccess,
                IN OUT PTDR_RUN_RESULTS resultsArray)
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

useTxChannel
Specifies the send channel. Set to TRUE to use Tx channel, FALSE for the Rx channel.

useTxReflectChannel
Specifies the reflect channel. Set to TRUE to use Tx channel, FALSE for the Rx channel.

threshold : Integer (0x21 - 0x3F)
Specifies the positive threshold value used when listening for a return TDR pulse.

stopAfterSuccess
Specifies whether measurements should stop after a successful threshold condition occurs. If FALSE, all possible timing configurations are run.

resultsArray
Array of 2 TDR_RUN_RESULTS structures. These will be set on return with the results of running 2 TDR pulses, 50ns and 100ns.

Return Value

TRUE if the specified threshold was met.The resultsArray structure will hold the TDR pulse results, the first structure will have the results for the 50ns pulse, the second structure will have the results for the 100ns pulse.

Comments

This runs through all useful combinations of TDR operations on the specified channel using the chip's 10Mb 50ns pulse generator. The positive threshold value can be specified.

The device must have the EPL_CAPA_TDR capability to use this function. A call to EPLIsDeviceCapable() can be used to determine if this operation is supported.

EPLInitTDR() and EPLDeinitTDR() must be called when using this function.

4.3.4.10 EPLRunTDR

This low-level function initiates a TDR operation and gathers the results of the operation.

```
NS_STATUS
    EPLRunTDR(
        IN PEPL_PORT_HANDLE portHandle,
        IN PTDR_RUN_REQUEST tdrParms,
        IN OUT PTDR_RUN_RESULTS tdrResults)
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

tdrParms
Pointer to a TDR_RUN_REQUEST structure that defines the parameters that will be used for the TDR operation.

tdrResults
Pointer to a TDR_RUN_RESULTS structure that will be set on return with the results of the TDR operation.

Return Value

NS_STATUS_SUCCESS
Function was successfully executed.

Comments

This low-level function initiates a TDR operation and gathers the results of the operation. The request structure defines the parameters of the operation.

The device must have the EPL_CAPA_TDR capability to use this function. A call to EPLIsDeviceCapable() can be used to determine if this operation is supported.

EPLInitTDR() and EPLDeinitTDR() must be called when using this function.

Field Name	**Data Type**	**Description**
sendPairTx	NS_BOOL	Defines which pair of wires the TDR pulse should be sent on. Send TDR pulse on Tx pair if TRUE, send on Rx pair if FALSE.
reflectPairTx	NS_BOOL	Defines which pair of wires to use to listen for a reflection. Listen for reflection on Tx pair if TRUE, listen on Rx pair if FALSE.
Use100MbTx	NS_BOOL	Specifies which Tx engine to use. If set to True the 100 MB transmit engine will be used. If False, the 10 MB engine will be used to send the TDR pulse.
txPulseTime	NS_UINT (0x00 - 0x07)	Specifies the TDR pulse length. If txUse100Mb is set to True this value is in 8ns units. If txUse100Mb is False this specifies the length in 50ns units, with a max of 2 (100ns).

detectPosThreshold	NS_BOOL	Specifies the polarity of the TDR pulse to detect. If set to TRUE the receive circuitry will detect a positive polarity reflection. If set to FALSE a negative polarity pulse will be detected. This parameter should always be set to TRUE when txUse100Mb is set to FALSE.
rxDiscrimStartTime	NS_UINT (0x00 - 0xFF)	Specifies the amount of time in 8ns units to wait prior to attempting detection of a reflection.
rxDiscrimStopTime	NS_UINT (0x00 - 0xFF)	Specifies the amount of time in 8ns units to wait before closing the window of reflection detection. This value must be greater then or equal to rxDiscrimStartTime.
rxThreshold	NS_UINT (0x00 - 0x3F)	Specifies the relative threshold that has to be met prior to a reflection being detected. If txUse100Mb is set TRUE then this value specifies either a positive or negative threshold with 0x20 being the midpoint. 0x00 - 0x1F are negative thresholds. 0x20 - 0x3F are positive thresholds. Only a positive value should be used when the txUse100Mb is set to FALSE. The threshold should be set according to the expected reflection pulse polarity as specified in the pulsePolarityPos parameter

Table 4.3-9 – TDR_RUN_REQUEST

4.3.4.11 EPLDspGetLinkQualityInfo

This returns the current status of the device's DSP link quality parameters and settings.

```
void
    EPLDspGetLinkQualityInfo(
        IN PEPL_PORT_HANDLE portHandle,
        IN OUT PDSP_LINK_QUALITY_GET linkQualityStruct);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.
linkQualityStruct
Pointer to a link quality get structure that will be filled out on return.

Return Value

None

Comments

The device must have the EPL_CAPA_LINK_QUALITY capability to use this function. A call to EPLIsDeviceCapable() can be used to determine if this is supported.

Field Name	Data Type	Description
linkQualityEnabled	NS_BOOL	Set to TRUE if the DSP Link Quality feature is enabled, FALSE otherwise.
freqCtrlHighWarn	NS_BOOL	Indicates if the frequency control high threshold was reached.
freqCtrlLowWarn	NS_BOOL	Indicates if the frequency control low threshold was reached.
freqOffHighWarn	NS_BOOL	Indicates if the frequency offset high threshold was reached.
freqOffLowWarn	NS_BOOL	Indicates if the frequency offset low threshold was reached.
dblwHighWarn	NS_BOOL	Indicates if the DBLW high threshold was reached.
dblwLowWarn	NS_BOOL	Indicates if the DBLW low zthreshold was reached.
dagcHighWarn	NS_BOOL	Indicates if the DAGC high threshold was reached.
dagcLowWarn	NS_BOOL	Indicates if the DAGC low threshold was reached.
c1HighWarn	NS_BOOL	Indicates if the C1 high threshold was reached.
c1LowWarn	NS_BOOL	Indicates if the C1 low threshold was reached.
freqCtrlLowThresh	NS_SINT	Currently configured frequency control low threshold.
freqCtrlHighThresh	NS_SINT	Currently configured frequency control high threshold.
freqOffLowThresh	NS_SINT	Currently configured frequency offset low threshold.
freqOffHighThresh	NS_SINT	Currently configured frequency offset high threshold.
dblwLowThresh	NS_SINT	Currently configured DBLW low threshold.
dblwHighThresh	NS_SINT	Currently configured DBLW high threshold.
dagcLowThresh	NS_UINT	Currently configured DAGC low threshold.
dagcHighThresh	NS_UINT	Currently configured DAGC high threshold.
c1LowThresh	NS_SINT	Currently configured C1 low threshold.
c1HighThresh	NS_SINT	Currently configured C1 high threshold.
freqCtrlSample	NS_SINT	Sampled frequency control parameter value.
freqOffSample	NS_SINT	Sampled frequency offset parameter value.
dblwCtrlSample	NS_SINT	Sampled DBLW parameter value.

dagcCtrlSample	NS_UINT	Sampled DAGC parameter value.
c1CtrlSample	NS_SINT	Sampled C1 parameter value.
restartOnC1	NS_BOOL	Allow auto restart on C1 threshold violation.
restartOnDAGC	NS_BOOL	Allow auto restart on DAGC threshold violation.
restartOnDBLW	NS_BOOL	Allow auto restart on DBLW threshold violation.
restartOnFreq	NS_BOOL	Allow auto restart on Freq threshold violation.
restartOnFC	NS_BOOL	Allow auto restart on FC threshold violation.
restartOnVar	NS_BOOL	Allow auto restart on Variance threshold violation.
dropLinkStatus	NS_BOOL	Set bit 8 in PCSR to allow PHY to indicate a dropped link if a threshold violation occurs.
varianceEnable	NS_BOOL	Set to TRUE to enable variance data capture
varianceSampleTime	NS_UINT8	Duration of variance sampling. 2, 4, 6, or 8ns
varianceWarn	NS_BOOL	Indicates if the SNR Variance threshold was reached. DP83640, DP83630, and DP83620 devices only.
varianceHighThresh	NS_UINT	SNR Variance high threshold. Valid values are from 0 – 2304. DP83640, DP83630, and DP83620 devices only.
varianceSample	NS_UINT	Sampled variance value. DP83640, DP83630, and DP83620 devices only.

Table 4.3-10 – DSP_LINK_QUALITY_GET

4.3.4.12 EPLDspSetLinkQualityConfig

This returns the current status of the device's DSP link quality parameters and settings.

```
void
        EPLDspSetLinkQualityConfig(
                IN PEPL_PORT_HANDLE portHandle,
                IN PDSP_LINK_QUALITY_SET linkQualityStruct);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.
linkQualityStruct
Pointer to a link quality set structure holding the desired link quality settings.

Return Value

None

Comments

The device must have the EPL_CAPA_LINK_QUALITY capability to use this function. A call to EPLIsDeviceCapable() can be used to determine if this is supported.

Field Name	Data Type	Description
linkQualityEnabled	NS_BOOL	Set to TRUE if the DSP Link Quality feature should be enabled, FALSE otherwise.
freqCtrlLowThresh	NS_SINT	Frequency control low threshold (min=-128, max=+127).
freqCtrlHighThresh	NS_SINT	Frequency control high threshold (min=-128, max=+127).
freqOffLowThresh	NS_SINT	Frequency offset low threshold (min=-128, max=+127).
freqOffHighThresh	NS_SINT	Frequency offset high threshold (min=-128, max=+127).
dblwLowThresh	NS_SINT	DBLW low threshold (min=-128, max=+127).
dblwHighThresh	NS_SINT	DBLW high threshold (min=-128, max=+127).
dagcLowThresh	NS_UINT	DAGC low threshold (min=0, max=+255).
dagcHighThresh	NS_UINT	DAGC high threshold (min=0, max=+255).
c1LowThresh	NS_SINT	C1 low threshold (min=-128, max=+127).
c1HighThresh	NS_SINT	C1 high threshold (min=-128, max=+127).
restartOnC1	NS_BOOL	Allow auto restart on C1 threshold violation.
restartOnDAGC	NS_BOOL	Allow auto restart on DAGC threshold violation.
restartOnDBLW	NS_BOOL	Allow auto restart on DBLW threshold violation.
restartOnFreq	NS_BOOL	Allow auto restart on Freq threshold violation.
restartOnFC	NS_BOOL	Allow auto restart on FC threshold violation.
restartOnVar	NS_BOOL	Allow auto restart on Variance threshold violation.
dropLinkStatus	NS_BOOL	Set bit 8 in PCSR to allow PHY to indicate a dropped link if a threshold violation occurs.
varianceEnable	NS_BOOL	Set to TRUE to enable variance data capture
varianceSampleTime	NS_UINT8	Duration of variance sampling. 2, 4, 6, or 8ns
varianceHighThresh	NS_UINT	SNR Variance high threshold. Valid values are from 0 – 2304. DP83640, DP83630, and DP83620 devices only.

Table 4.3-11 – DSP_LINK_QUALITY_SET

4.4 IEEE 1588 Function Reference

4.4.1 IEEE 1588 Configuration APIs

This section describes the configuration and setup API functions related to the IEEE 1588 features in the DP83640, DP83630, and DP83620 devices. These are used during initialization to setup the device for the desired operational behavior.

Most of the calls defined here are not normally called directly by an application. Most are used internally by the library to get everything configured based on the input parameters defined in RunTimeOpts (see details below in section 5) and in the operation of a PTP stack. The can be very helpful and necessary if you are writing or modifying a PTP stack.

4.4.1.1 PTPEnable

Enables or disables the PHY's PTP 1588 clock.

```
void
        PTPEnable (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_BOOL enableFlag);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.
enableFlag
Set to TRUE to enable the PHY's PTP 1588 clock. Set to FALSE to disable it.

Return Value

None

Comments

This must be called during initialization to enable the 1588 hardware features.

4.4.1.2 PTPThreadC

Starts the initialization process and runs the PTP protocol.

```
void
        PTPThreadC (
                IN PEPL_PORT_HANDLE portHandle,
                IN void *guiObj,
                IN void *stdioCallback,
                IN void *statusUpdateCallback,
                IN RunTimeOpts *ptpStackCfg);
```

Parameters

portHandle
: Handle that represents a port. This is obtained using the EPLEnumPort function.

guiObj
: This is a generic pointer to that can be used to communicate between the application and the library. Currently this is not used by the C interface.

stdioCallback
: This is a pointer to a function that performs output for the library. The library will call this function to display various operational and debug messages. This has no impact on the operation of the stack it is provided as a means to pass status information to the application layer if desired. This can be disabled by passing in a NULL value.

statusUpdateCallback
: This is a pointer to a function that performs status update for the library. The library will call this function to provide operational information to the application. This has no impact on the operation of the stack it is provided as a means to pass status information to the application layer if desired. This can be disabled by passing in a NULL value.

ptpStackCfg
: This is a pointer to a structure of configuration items that control the operation of the stack. For details of the various parameters see the **PTPTestApp** source code and discussion below in section 5.

Return Value

None

Comments

This is the main entry point to the operation of the PTP protocol stack. As its name indicates it is intended to be run as a separate thread from the application. This call will not return until **PTPKillThread()** is called. At that time the thread should be stopped.

stdioCallback is a pointer to a function in the application that is accessible from the thread that calls this function. It should be compatible with the following prototype.

```
void printMsg( int  msgType, char *msgString )
// Display message from stack
// Input: msgType - Type of message provided
//          0 - Debug Message (verbose)
//          1 - Debug Message (normal)
//          2 - Nofication Message
//          3 - Error Message
//        msgString - The message string
```

The callback **msgType** parameter can be used filter messages as needed. Most messages currently in the library are type 1 for normal debug. **NOTE: this function should not perform too much work. If possible it should simply place the messages in a queue to be processed by a different thread otherwise significant delays can be introduced into the stack.**

statusUpdateCallback is a pointer to a function that performs status update for the library. The library will call this function to provide operational information to the application.. It should be compatible with the following prototype.

```
void ptpStatusUpdate( NS_UINT8 stsType, void *stsData )
// Display message from stack
// Input: stsType - Type of status provided
//          STS_PSF_DATA (1) -PHY Status Frame Data
//          STS_OFFSET_DATA (2) - Offset data
//         stsData - pointer to actual data
```

The basic operation of this callback should be as follows:

```
switch( stsType ) {
case STS_PSF_DATA:
    {
    PHYMSG_MESSAGE_TYPE_ENUM msgType;
    PHYMSG_MESSAGE phyMsg;
    NS_UINT8 *nxtMsg;
        // Normally we'd call IsPhyStatusFrame() using a raw packet but
        // since we got here we already know that the data is the 1st
        // of potentially several PSFs so we just process it.
        nxtMsg = (NS_UINT8 *)stsData;
        while( nxtMsg ) {
            nxtMsg = GetNextPhyMessage( devPort, nxtMsg, &msgType, &phyMsg );
            if( !nxtMsg || !bPSF ) {
                continue;
            }
            switch( msgType ) {
            case PHYMSG_STATUS_TX:
            case PHYMSG_STATUS_RX:
            case PHYMSG_STATUS_TRIGGER:
            case PHYMSG_STATUS_EVENT:
            case PHYMSG_STATUS_ERROR:
            case PHYMSG_STATUS_REG_READ:
            default:
                // do something with the PSFs
                break;
            }
        }  // while( nxtMsg )
    }
    break;
```

```
        case STS_OFFSET_DATA:
            {
            STS_OFFSET_DATA_STRUCT *stsODS = (STS_OFFSET_DATA_STRUCT *)stsData;
                if( bSTSUpdate ) {
                    printf( "ptpStatusUpdate %d.%d  %d.%d  %d.%d  %d.%d\n",
                            stsODS->offset_from_master.seconds,
                            stsODS->offset_from_master.nanoseconds,
                            stsODS->master_to_slave_delay.seconds,
                            stsODS->master_to_slave_delay.nanoseconds,
                            stsODS->slave_to_master_delay.seconds,
                            stsODS->slave_to_master_delay.nanoseconds,
                            stsODS->oneWayAvg.seconds,
                            stsODS->oneWayAvg.nanoseconds );
                }
            }
            break;
        default:
            break;
        } // switch( stsType )
```

The operation of this function is designed to support functionality similar to what is available in the ALP Framework demo. If this functionality is used at all it is expected that both the library and the operation of the procedure will be modified to meet the desired needs of the system.

NOTE: this function should not perform too much work. If possible it should simply place the messages/data in a queue to be processed by the application in an application thread.

For additional details about this call and the callbacks see the **PTPTestApp** source code and the discussion below in section 5.

There is a Python version of this call (**PTPThread()**) that operates exactly the same with the exception that the pointers and callbacks are PyObjects and are managed using the Python library calls.

4.4.1.3 PTPKillThread

Call from outside the PTPThread to signal that a shutdown request

```
void
        PTPKillThread (
                IN PEPL_PORT_HANDLE portHandle);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

Return Value

None

Comments

When this is called it simply sets the killThread flag to signal to the protocol that is should shutdown. This will cause the **PTPThreadC()** function to return back to the calling function and allow the application to shutdown properly.

4.4.1.4 PTPSetTriggerConfig

Configures the operational behavior of an individual trigger.

```
void
        PTPSetTriggerConfig (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT trigger,
                IN NS_UINT triggerBehavior,
                IN NS_UINT gpioConnection);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

trigger
The trigger to configure, 0 – 7.

triggerBehavior
A bitmap of configuration options. Zero or more of the bits defined in the TRIGGER_CFG_OPTIONS bit table below OR'ed together.

gpioConnection
The GPIO pin the trigger should be connected to. A value of 0 – 12. If 0 is specified no GPIO pin connection is made.

Return Value

None

Comments

Cfg Bit Name	Description
TRGOPT_PULSE	Causes the Trigger to generate a Pulse rather than a single rising or falling edge.
TRGOPT_PERIODIC	Causes the Trigger to generate a periodic signal. If not set, the Trigger will generate a single Pulse or Edge depending on the Trigger Control settings.
TRGOPT_TRG_IF_LATE	Allow an immediate Trigger in the event the Trigger is programmed to a time value which is less than the current time. This provides a mechanism for generating an immediate trigger or to immediately begin generating a periodic signal. For a periodic signal, no notification will be generated if this bit is set and a Late Trigger occurs.
TRGOPT_NOTIFY_EN	Enables Trigger status to be reported on completion of a Trigger or on an error detection due to late trigger. If Trigger interrupts are enabled, the notification will also result in an interrupt being generated.
TRGOPT_TOGGLE_EN	Puts the trigger into toggle mode. In toggle mode, the initial value will be ignored and the trigger output will be toggled at the trigger time.

Table 4.4-1 – TRIGGER_CFG_OPTIONS

4.4.1.5 PTPSetEventConfig

Configures the operational behavior of an individual event.

```
void
    PTPSetEventConfig (
        IN PEPL_PORT_HANDLE portHandle,
        IN NS_UINT event,
        IN NS_BOOL eventRiseFlag,
        IN NS_BOOL eventFallFlag,
        IN NS_BOOL eventSingle,
        IN NS_UINT gpioConnection);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

event
The event to configure, 0 – 7.

eventRiseFlag
If set to TRUE, enables detection of rising edge on Event input.

eventFallFlag
If set to TRUE, enables detection of falling edge on Event input.

eventSingle
If set to TRUE, enables single event capture operation.

gpioConnection
The GPIO pin the event should be connected to. A value of 0 – 12. If 0 is specified no GPIO pin connection is made.

Return Value

None

Comments

None

4.4.1.6 PTPSetTransmitConfig

Configures the device's 1588 transmit operation.

```
Void
        PTPSetTransmitConfig (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT txConfigOptions,
                IN NS_UINT ptpVersion,
                IN NS_UINT ptpFirstByteMask,
                IN NS_UINT ptpFirstByteData);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

txConfigOptions
A bitmap of configuration options. Zero or more of the bits defined in the TX_CFG_OPTIONS bit table below OR'ed together.

ptpVersion
Enable Timestamp capture for a specific version of the IEEE 1588 specification. This value may be set to any value between 1 and 15 and allows support for future versions of the IEEE 1588 specification. A value of 0 will disable version checking (not recommended).

ptpFirstByteMask
Bit mask to be used for matching Byte0 of the PTP Message. A one in any bit enables matching for the associated data bit. If no matching is required, all bits of the mask should be set to 0.

ptpFirstByteData
Data to be used for matching Byte0 of the PTP Message. This parameter is ignored if ptpFirstByteMask is 0x00.

Return Value

None

Comments

Cfg Bit Name	**Description**
TXOPT_SYNC_1STEP	Enable automatic insertion of timestamp into transmit Sync Messages. Device will automatically parse message and insert the timestamp in the correct location. UPD checksum and CRC fields will be regenerated.
TXOPT_DR_INSERT	Enables insertion of the timestamp from transmitted Delay_Req messages into inbound Delay_Resp messages. The most recent timestamp will be used for any inbound Delay_Resp message. The receive timestamp insertion logic must be enabled using the PTPSetReceiveConfig call.
TXOPT_NTP_TS_EN	Enables insertion of timestamp into NTP Packets.
TXOPT_IGNORE_2STEP	If set the device will insert a timestamp independent of the setting of the Two_Step flag, otherwise the device will not insert a timestamp if the Two_Step bit is set in the flags field of the PTP header.

TXOPT_CRC_1STEP	If this set the device will send the One-Step frame with a valid CRC, even if the incoming CRC is invalid, otherwise the device will force a CRC error for One-Step operation if the incoming frame has a CRC error.
TXOPT_CHK_1STEP	Enables correction of the IPv4 UDP checksum for messages which include insertion of the timestamp. The checksum is corrected by modifying the last two bytes of the UDP data. The last two bytes must be transmitted by the MAC as 0's. This control will have no effect for IPv6/UDP or Layer2 Ethernet messages.
TXOPT_IP1588_EN	Enables filtering of UDP/IP Event messages using the IANA assigned IP Destination addresses. If set, packets with IP Destination addresses which do not match the IANA assigned addresses will not be timestamped. This field affects operation for both IPv4 and IPv6. If not specified the IP destination addresses will be ignored.
TXOPT_L2_EN	Enables detection of IEEE 802.3/Ethernet encapsulated PTP event messages.
TXOPT_IPV6_EN	Enables detection of UDP/IPv6 encapsulated PTP event messages.
TXOPT_IPV4_EN	Enables detection of UDP/IPv4 encapsulated PTP event messages.
TXOPT_TS_EN	Enables Timestamp capture for Transmit.

Table 4.4-2 – TX_CFG_OPTIONS

4.4.1.7 PTPSetPhyStatusFrameConfig

Configures the device's PHY Status Frame (PSF) operational configuration.

```
void
        PTPSetPhyStatusFrameConfig (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT statusConfigOptions,
                IN MAC_SRC_ADDRESS_ENUM srcAddrToUse,
                IN NS_UINT minPreamble,
                IN NS_UINT ptpReserved,
                IN NS_UINT ptpVersion,
                IN NS_UINT transportSpecific,
                IN NS_UINT8 messageType,
                IN NS_UINT32 sourceIpAddress);
```

Parameters

portHandle

Handle that represents a port. This is obtained using the EPLEnumPort function.

statusConfigOptions

A bitmap of configuration options. Zero or more of the bits defined in the STS_CFG_OPTIONS bit table below OR'ed together.

srcAddrToUse

Specifies the MAC source address to use in Phy Status Frames (PSF). Must be one of the following values.

STS_SRC_ADDR_1 - [00 00 00 00 00 00]
STS_SRC_ADDR_2 - [08 00 17 0B 6B 0F]
STS_SRC_ADDR_3 - [08 00 17 00 00 00]
STS_SRC_ADDR_USE_MC - Use MAC multicast destination address

minPreamble

Determines the minimum number of preamble bytes required for sending PHY Status Frames (PCF) on the MII interface. It is recommended that this be set to the smallest value the MAC will tolerate.

ptpReserved

PTP v2 reserved field: This field contains the reserved 4-bit field (at offset 1) to be sent in status packets from the PHY to the local MAC using the MII receive data interface.

ptpVersion

The PTP version number to set in the PTP version field of the status frame. Typically this is set to a value of 0x02.

transportSpecific

The value to use for the transportSpecific field for status frames from the PHY to the local MAC using the MII receive data interface. A value of 0x0F ensures the frame will not be interpreted as a valid PTP message.

messageType

The value to use for the messageType field for status frames from the PHY to the local MAC using the MII receive data interface. The default value of 0x0F ensures the frame will not be interpreted as a valid PTP message.

sourceIpAddress

4-byte source IP address to use in frames created by the PHY. The address is expected to be in network order. 224.0.1.129 is E0.00.01.81 in hex and is encode as 0x810100E0.

Return Value

None

Comments

Cfg Bit Name	Description
STSOPT_LITTLE_ENDIAN	For each 16-bit field in a Status Message, the data will normally be presented in network byte order (Most significant byte first). If this is set, the byte data fields will be reversed so that the least significant byte is first (little endian).
STSOPT_IPV4	If set, IPv4 frames will be used to deliver PHY Status Frames (PSF), otherwise Layer2 Ethernet frames will be used.
STSOPT_PCFR_EN	Enables frame based status delivery of PHY Control Frame Read reponses.
STSOPT_ERR_EN	Enables frame based status delivery of errors.
STSOPT_TXTS_EN	Enables frame based status delivery of Transmit Timestamps.
STSOPT_RXTS_EN	Enables frame based status delivery of Receive Timestamps.
STSOPT_TRIG_EN	Enables frame based status delivery of Trigger Status.
STSOPT_EVENT_EN	Enables frame based status delivery of Event Timestamps.

Table 4.4-3 – STS_CFG_OPTIONS

4.4.1.8 PTPSetReceiveConfig

Configures the device's receive operation.

```
void
        PTPSetReceiveConfig (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT rxConfigOptions,
                IN RX_CFG_ITEMS *rxConfigItems );
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

rxConfigOptions
A bitmap of configuration options. Zero or more of the bits defined in the RX_CFG_OPTIONS bit table below OR'ed together.

rxConfigItems
This structure of configuration values must be filled out prior to making this call. See the RX_CFG_ITEMS structure definition below.

Return Value

None

Comments

Cfg Bit Name	Description
RXOPT_DOMAIN_EN	Domain Match Enable: If set to 1, the Receive Timestamp unit will require the Domain field to match the value programmed in the PTP_DOMAIN field of the PTP_RXCFG3 register. If set to 1, the Receive Timestamp will ignore the PTP_DOMAIN field.
RXOPT_ALT_MAST_DIS	Alternate Master Timestamp Disable: Disables Timestamp generation if the Alternate_Master flag is set: 1 = Do not generate Timestamp if Alternate_Master = 1 0 = Ignore Alternate_Master flag
RXOPT_USER_IP_SEL	IP Address data select: Selects portion of IP address accessible through the PTP_RXCFG2 register: 0 = Most Significant Octets 1 = Least Significant Octets
RXOPT_USER_IP_EN	Enable User-programmed IP address filter: Enable detection of UDP/IP Event messages using a programmable IP addresses. The IP Address is set using the PTP_RXCFG2 register.
RXOPT_RX_SLAVE	Receive Slave Only: By default, the Receive Timestamp Unit will provide Timestamps for event messages meeting other requirements. Setting this bit to a 1 will prevent Delay_Req messages from being Timestamped by requiring that the Control Field (offset 32 in the PTP message) be set to a value other than 1.

RXOPT_IP1588_EN0 RXOPT_IP1588_EN1 RXOPT_IP1588_EN2	Enable IEEE 1588 defined IP address filters: Enable detection of UDP/IP Event messages using the IANA assigned IP Destination addresses. Zero or more of these options may be OR'ed together to form the desired filter. This field affects operation for both IPv4 and IPv6. A Timestamp is captured for the PTP message if the IP destination address matches the following: RXOPT_IP1588_EN0: Dest IP address = 224.0.1.129 RXOPT_IP1588_EN1: Dest IP address = 224.0.1.130-132 RXOPT_IP1588_EN2: Dest IP address = 224.0.0.107
RXOPT_RX_L2_EN	Layer2 PTP Detection Enable: Enables detection of IEEE 802.3/Ethernet encapsulated PTP event messages.
RXOPT_RX_IPV6_EN	IPv6 PTP Detection Enable: Enables detection of UDP/IPv6 encapsulated PTP event messages.
RXOPT_RX_IPV4_EN	IPv4 PTP Detection Enable: Enables detection of UDP/IPv4 encapsulated PTP event messages.
RXOPT_SRC_ID_HASH_EN	Enables source identification hash matching to determine if a receive timestamp should be recorded for a particular incoming PTP frame. If set, the RX_CFG_ITEMS.srcIdHash field must be set.
RXOPT_RX_TS_EN	Receive Timestamp Enable: Enable Timestamp capture for Receive.
RXOPT_ACC_UDP	Record Timestamp if UDP Checksum Error: By default, Timestamps will be discarded for frames with UDP Checksum errors. If this bit is set, then the Timestamp will be made available in the normal manner.
RXOPT_ACC_CRC	Record Timestamp if CRC Error: By default, Timestamps will be discarded for frames with CRC errors. If this bit is set, then the Timestamp will be made available in the normal manner.
RXOPT_TS_APPEND	Append Timestamp for L2: For Layer 2 encapsulated PTP messages, if this bit is set, always append the Timestamp to end of the PTP message rather than inserted in unused message fields. This bit will be ignored if TS_INSERT is 0.
RXOPT_TS_INSERT	Enable Timestamp Insertion: Enables Timestamp insertion into a frame containing a PTP Event Message. If this bit is set, the Timestamp will not be available through the PTP Receive Timestamp Register.
RXOPT_IPV4_UDP_MOD	Enable IPV4 UDP modification: When timestamp insertion is enabled, this bit controls how UDP checksums are handled for IPV4 PTP event messages. If set to a 0, the device will clear the UDP checksum. If a UDP checksum error is detected the device will force a CRC error. If set to a 1, the device will not clear the UDP checksum. Instead it will generate a 2-byte value to correct the UDP checksum and append this immediately following the PTP message. If an incoming UDP checksum error is detected, the device will cause a UDP checksum error in the modified field. This function should only be used if the incoming frames contain two extra bytes of UDP data following the PTP message. This should not be enabled for systems using version 1 of the IEEE 1588 specification.
RXOPT_TS_SEC_EN	Enable Timestamp Seconds: Setting this bit to a 1 enables inserting a

	seconds field when Timestamp Insertion is enabled. If set to 0, only the nanoseconds (with bits [31:30] containing bits [1:0] of the current seconds timestamp value) portion of the Timestamp will be inserted in the frame. This bit will be ignored if TS_INSERT is 0.

Table 4.4-4 – RX_CFG_OPTIONS

Field Name	Data Type	Description
ptpVersion	NS_UINT	Enable Timestamp capture for a specific version of the IEEE 1588 specification. This value may be set to any value between 1 and 15 and allows support for future versions of the IEEE 1588 specification. A value of 0 will disable version checking (not recommended).
ptpFirstByteMask	NS_UINT	Bit mask to be used for matching Byte0 of the PTP Message. A one in any bit enables matching for the associated data bit. If no matching is required, all bits of the mask should be set to 0.
ptpFirstByteData	NS_UINT	Data to be used for matching Byte0 of the PTP Message. This parameter is ignored if ptpFirstByteMask is 0x00.
ipAddrData	NS_UINT32	Receive IP Address Data: 32-bits of the IP Address field.
tsMinIFG	NS_UINT	Minimum Inter-frame Gap: When a Timestamp is appended to a PTP Message, the length of the frame may get extended. This could reduce the Inter-frame Gap (IFG) between frames by as much as 8 byte times (640ns at 100Mb). This field sets a minimum on the IFG between frames in number of byte times. If the IFG is set larger than the actual IFG, preamble bytes of the subsequent frame will get dropped. This value should be set to the lowest possible value that the attached MAC can support.
srcIdHash	NS_UINT	This value defines the expected source identity hash value for incoming PTP event messages. Only Bits 11 – 0 are significant. This value is ignored if RXOPT_SRC_ID_HASH_EN NOT specified in the rxConfigOptions bit map.
ptpDomain	NS_UINT	PTP Domain: Value of the PTP Message domainNumber field. If PTP_RXCFG0:DOMAIN_EN is set to 1, the Receive Timestamp unit will only capture a Timestamp if the domainNumber in the receive PTP message matches the value in this field. If the DOMAIN_EN bit is set to 0, the domainNumber field will be ignored.
tsSecLen	NS_UINT	Inserted Timestamp Seconds Length: This field indicates the length of the Seconds field to be inserted in the PTP message. This field will be ignored if RXOPT_TS_INSERT is 0 or if RXOPT_TS_SEC_EN is 0. The mapping is as follows: 0x00 : Least Significant Byte only of Seconds field 0x01 : Two Least Significant Bytes of Seconds field 0x02 : Three Least Significant Bytes of Seconds field 0x03 : All four Bytes of Seconds field
rxTsNanoSecOffset	NS_UINT	Receive Timestamp Nanoseconds offset: This field provides an

		offset to the Nanoseconds field when inserting a Timestamp into a received PTP message. If RXOPT_TS_APPEND is set, the offset indicates an offset from the end of the PTP message. If RXOPT_TS_APPEND is NOT set, the offset indicates the byte offset from the beginning of the PTP message. This field will be ignored if RXOPT_TS_INSERT is not specified in the RX_CFG_OPTIONS.
rxTsSecondsOffset	NS_UINT	Receive Timestamp Seconds offset: This field provides an offset to the Seconds field when inserting a Timestamp into a received PTP message. If RXOPT_TS_APPEND is set, the offset indicates an offset from the end of the inserted Nanoseconds field. If RXOPT_TS_APPEND is NOT set, the offset indicates the byte offset from the beginning of the PTP message. This field will be ignored if RXOPT_TS_INSERT is not specified in the RX_CFG_OPTIONS.

Table 4.4-5 – RX_CFG_ITEMS

4.4.1.9 PTPCalcSourceIdHash

Utility function that will calculate a 12-bit CRC-32 (IEEE 802.3, no complement) value used to program the RX_CFG_ITEMS.srcIdHash field.

```
NS_UINT
        PTPCalcSourceIdHash (
                IN NS_UINT8 *tenBytesData );
```

Parameters

tenBytesData

This is a pointer to the desired fixed values found in bytes 20 – 29 of the PTP event message that receive source identification filtering should occur.

Return Value

Value representing the CRC-32 (IEEE 802.3, no complement) generated from the specified 10-byte buffer.

Comments

This function would only be used if RXOPT_SRC_ID_HASH_EN was specified in a call to PTPSetReceiveConfig.

4.4.1.10 PTPSetTempRateDurationConfig

Configures the PTP Temporary Duration.

```
void
        PTPSetTempRateDurationConfig (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT32 duration );
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

duration
PTP Temporary Rate Duration: This sets the duration for the Temporary Rate in number of clock cycles. The actual Time duration is dependent on the value of the configured Temporary Rate. This value has a range of up to 2^26 (26-bits may be defined).

Return Value

None

Comments

This value is remembered by the hardware, therefore the setting can be used for multiple temporary clock adjustments, if the desired duration remains constant.

4.4.1.11 PTPSetClockConfig

Configures the general PTP clock configuration options.

```
void
        PTPSetClockConfig (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT clockConfigOptions,
                IN NS_UINT ptpClockDivideByValue,
                IN NS_UINT ptpClockSource,
                IN NS_UINT ptpClockSourcePeriod );
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

clockConfigOptions
A bitmap of configuration options. Zero or more of the bits defined in the CLOCK_CFG_OPTIONS bit table below OR’ed together.

ptpClockDivideByValue
PTP Clock Divide-by Value: This parameter defines the divide-by value for the output clock. The output clock is divided from an internal 250MHz clock. Valid values range from 2 to 255 (0x02 to 0xFF), giving a nominal output frequency range of 125MHz down to 980.4kHz. Divide-by values of 0 and 1 are not valid and will stop the output clock.

ptpClockSource
PTP Clock Source Select: Selects among three possible sources for the PTP reference clock, one of the following values may be specified:
0x00 - 125MHz from internal PGM (default)
0x01 - Divide-by-N from 125MHz internal PGM
0x02 - External reference clock

ptpClockSourcePeriod
PTP Clock Source Period: Configures the PTP clock source period in nanoseconds. Values less than 8 are invalid. This parameter is used as follows by the different clock source modes:
0x00 – 125MHz - Ignored
0x01 – Divide-by-N - Bits 6:3 are used to divide the 125MHz PGM clock by a value between 1 and 15. Bits 2:0 are ignored.
0x02 – External - Bits 6:0 indicate the nominal period in nanoseconds of the external reference clock.

Return Value

None

Comments

Cfg Bit Name	Description
CLKOPT_CLK_OUT_EN	PTP Clock Output Enable: If set, enables PTP divide-by-N clock output. If not specified, disables PTP divide-by-N clock output.
CLKOPT_CLK_OUT_SEL	PTP Clock Output PGM Select: If set, selects the PGM as the root clock for generating the divide-by-N output. If not specified, selects the FCO as the root clock for generating the divide-by-N output.
CLKOPT_CLK_OUT_SPEED_SEL	PTP Clock Output I/O Speed Select: If set, enables faster rise/fall time for the divide-by-N clock output pin. If not specified, enables normal rise/fall time for the divide-by-N clock output pin.

Table 4.4-6 – CLOCK_CFG_OPTIONS

4.4.1.12 PTPSetGpioInterruptConfig

Configures the GPIO pin to be used for the PTP Interrupt function.

```
void
        PTPSetGpioInterruptConfig (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT gpioInt);
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

gpioInt
Specifies the number of the GPIO to assign to the PTP Interrupt. This is a value from 1 – 12. Specifying 0 disables interrupt / GPIO assignment.

Return Value

None

Comments

None

4.4.1.13 PTPSetMiscConfig

Sets various miscellaneous PTP configuration options.

```
void
        PTPSetMiscConfig (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT ptpEtherType,
                IN NS_UINT ptpOffset,
                IN NS_UINT txSfdGpio,
                IN NS_UINT rxSfdGpio );
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

ptpEtherType
This parameter defines the Ethernet Type field used to detect
PTP messages transported over Ethernet layer 2. Normally this would be set to 0xF788.

ptpOffset
This parameter defines the offset in bytes to the PTP Message from the preceding header. For Layer2, this is the offset from the Ethernet Type Field. For IP/UDP, it is the offset from the end of the UDP Header. Values are from 0x00 – 0xFF.

txSfdGpio,
Tx Start of Frame GPIO Select: This parameter specifies the GPIO output to which the Tx SFD signal is assigned. Valid values are 0 (disabled) or 1-12.

rxSfdGpio
Rx Start of Frame GPIO Select: This parameter specifies the GPIO output to which the Rx SFD signal is assigned. Valid values are 0 (disabled) or 1-12.

Return Value

None

Comments

None

4.4.2 IEEE 1588 Clock APIs

This section describes the API functions related to the IEEE 1588 clock features in the DP83640, DP83630, and DP83620 devices that are typically used during run-time system operation. Refer to the IEEE 1588 Hardware overview section of this document for details on the various methods of adjustment and setting the clock.

4.4.2.1 PTPClockReadCurrent

Returns a snapshot of the current IEEE 1588 clock value.

```
void
        PTPClockReadCurrent (
                IN PEPL_PORT_HANDLE portHandle,
                IN OUT NS_UINT32 *retNumberOfSeconds,
                IN OUT NS_UINT32 *retNumberOfNanoSeconds );
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

retNumberOfSeconds
Will be set on return to the number of seconds comprising the IEEE 1588 hardware clock.

retNumberOfNanoSeconds
Will be set on return to the number of nanoseconds comprising the IEEE 1588 hardware clock. This value cannot be larger then 1e9 (1 second).

Return Value

None

Comments

None

4.4.2.2 PTPClockStepAdjustment

A step adjustment value is added to the current IEEE 1588 hardware clock value. Note that the adjustment value can be positive or negative.

```
void
        PTPClockStepAdjustment (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT32 numberOfSeconds,
                IN NS_UINT32 numberOfNanoSeconds,
                IN NS_BOOL negativeAdj );
```

Parameters

portHandle
: Handle that represents a port. This is obtained using the EPLEnumPort function.

numberOfSeconds
: The number of seconds to add to the current clock value. This should always be a positive value, use the negativeAdj flag to indicate a negative overall value.

numberOfNanoSeconds
: The number of nanoseconds to add to the current clock value. This should always be a positive value, use the negativeAdj flag to indicate a negative overall value. This value must NOT be larger then 1e9 (1 second).

negativeAdj
: If TRUE, the numberOfSeconds and numberOfNanoSeconds values will be subtracted from the current clock value, otherwise the values will be an added to the clock.

Return Value

None

Comments

This function does not account for the underlying time required within the hardware to make the adjustment (2 clock periods = 16ns). The caller should add 16ns to the signed value of the adjustment prior to determining whether the adjustment is positive or negative.

4.4.2.3 PTPClockSet

Sets the IEEE 1588 hardware clock equal to the specified time value.

```
void
        PTPClockSet (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT32 numberOfSeconds,
                IN NS_UINT32 numberOfNanoSeconds );
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

numberOfSeconds
The number of seconds to set the current clock value too.

numberOfNanoSeconds
The number of nanoseconds to set the current clock value too. This value must NOT be larger then 1e9 (1 second).

Return Value

None

Comments

None

4.4.2.4 PTPClockSetRateAdjustment

The clock can be programmed to operate at an adjusted frequency value by programming a rate adjustment value. The rate adjustment allows for correction on the order of 2-32ns per reference clock cycle. The frequency adjustment will allow the clock to correct the offset over time, avoiding any potential side-effects caused by a step adjustment in the time value. The rate adjustment can be the normal adjustment rate that will always be used, and a temporary rate adjustment can be specified that will only occur for a preprogrammed amount of time.

```
void
        PTPClockSetRateAdjustment (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT32 rateAdjValue,
                IN NS_BOOL tempAdjFlag,
                IN NS_BOOL adjDirectionFlag );
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

rateAdjValue
This 26-bit magnitude is the number of 2^-32 ns units that should be added to or subtracted from the IEEE 1588 hardware clock per reference cycle (8ns).

tempAdjFlag
If set to TRUE the set rate will override any previously set normal adjustment value for the amount of time specified using the PTPSetTempRateDurationConfig() function. If this is FALSE, this function sets the normal clock adjustment time.

adjDirectionFlag
If set to TRUE, this will cause the PTP Clock to operate at a higher frequency than the reference. The rateAdjValue value will be decremented from the clock on every cycle. If set to FALSE, the rateAdjValue will be added to the clock on every cycle thus causing the IEEE 1588 hardware clock to operate at a slower frequency.

Return Value

None

Comments

This function allows setting two different rate adjustment values, a Normal Rate and a Temporary Rate. The Temporary Rate allows the 1588 Clock to operate at a modified rate for a programmed amount of time, as defined with PTPSetTempRateDurationConfig(). The Normal Rate will be selected and used if a Temporary Rate is not currently active.

When setting a rate, the tempAdjFlag parameter indicates the rate is to be temporary. Following completion of the time duration the rate will revert back to the Normal Rate. Note that the Normal Rate may be changed while a Temporary Rate is active. This will have not effect on the Temporary Rate, but the new Normal Rate will be used when the Temporary Rate Duration completes.

To adjust the time value, software uses the 1588 protocol to determine the time correction required. The time correction may be spread over multiple clock cycles by programming a

Temporary Rate value. To determine the rate setting, software should compute the rate difference as the time correction divided by the time duration in number of 8ns clock cycles. This value should be multiplied by 2^32 to convert to the correct units. This rate difference should then be added to the current PTP Rate setting to provide the Temporary Rate.

The Temporary Rate value is a 26-bit value plus a sign bit (adjDirectionFlag), providing a range of -(2^26-1) to +(2^26-1) in units of 2^-32ns/cycle. Since each reference clock cycle is 8ns, this allows for a rate adjustment maximum of approximately +/-1950ppm.

The Temporary Rate duration is a 26-bit value providing a duration of up to 536ms.

Example:

```
Conditions:
Current Rate = +10 ppm, which gives PTP_Rate = 343597 (2^-32ns/cycle)
Time_Error = 20ns, gives Time_Corr = -20ns
Assume the correction will be done over 1ms, gives
Temp_Rate_duration = 1ms/8ns = 125,000

Calculation:
Temp_Rate_delta = (Time_Corr/Temp_Rate_duration) * 2^32
       = (-20/125000) * 2^32 = -687194
Temp_Rate = Current_Rate + Temp_Rate_delta = 343597 + -687194 = -343597
```

4.4.3 IEEE 1588 Check For Events API

This section defines an API function that must be used to determine if any hardware events are waiting for processing.

4.4.3.1 PTPCheckForEvents

Checks to determine if any of the following hardware events are outstanding: Transmit timestamp, receive timestamp, trigger done and event timestamp.

```
NS_UINT
    PTPCheckForEvents (
        IN PEPL_PORT_HANDLE portHandle );
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

Return Value

A bit map of zero or more bits set indicating the types of events that are available from the hardware. The defined bits are:

- PTPEVT_TRANSMIT_TIMESTAMP_BIT
- PTPEVT_RECEIVE_TIMESTAMP_BIT
- PTPEVT_EVENT_TIMESTAMP_BIT
- PTPEVT_TRIGGER_DONE_BIT

Comments

This must be called prior to retrieving individual events from the hardware. The bit map must be used to determine which “Get” functions need to be called. The applicable “Get” functions are:

- PTPGetTransmitTimestamp
- PTPGetReceiveTimestamp
- PTPGetEvent
- PTPHasTriggerExpired

It is NOT necessary to call this function prior to calling PTPHasTriggerExpired, although this function can be useful in a main polling loop to quickly determine if there is an expired trigger as well as the other types of events with a single call.

4.4.4 IEEE 1588 Transmit Frame Timestamp APIs

4.4.4.1 PTPGetTransmitTimestamp

Returns the next available transmit timestamp.

```
void
        PTPGetTransmitTimestamp (
                IN PEPL_PORT_HANDLE portHandle,
                IN OUT NS_UINT32 *retNumberOfSeconds,
                IN OUT NS_UINT32 *retNumberOfNanoSeconds,
                IN OUT NS_UINT *overflowCount );
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

retNumberOfSeconds
Will be set on return to the next transmit’s seconds timestamp value.

retNumberOfNanoSeconds
Will be set on return to the next transmit’s nanoseconds timestamp value. This value will never be larger then 10^9 (1 second).

overflowCount
Will be set on return and indicates if timestamps were dropped due to an overflow of the Transmit Timestamp queue. The overflow counter will stick at a value of three if more then three timestamps were missed. Normally this value should be 0.

Return Value

Nothing

Comments

The caller must have previously called PTPCheckForEvents() and determined that the PTPEVT_TRANSMIT_TIMESTAMP_BIT bit was set prior to invoking this function. This function does NOT check to determine if an outstanding transmit event is available.

The hardware can queue up to four transmit timestamps. If more then four transmits occur without first reading the transmit timeout, then the overflowCount will indicate (up to 3) the number of timestamps that were dropped due to overflow.

This function does not adjust the timestamp to account for the delay to the wire. The caller should make the necessary adjustment(s) as needed. The typical adjustment is to add the appropriate default outbound latency delay value, either DEFAULT_OUTBOUND_LATENCY or DEFAULT_OUTBOUND_LATENCY_10MB which can be found in constants.h, to the timestamp.

4.4.5 IEEE 1588 Receive Frame Timestamp APIs

4.4.5.1 PTPGetReceiveTimestamp

Returns the next available receive timestamp from the device’s receive timestamp queue.

```
void
        PTPGetReceiveTimestamp (
                IN PEPL_PORT_HANDLE portHandle,
                IN OUT NS_UINT32 *retNumberOfSeconds,
                IN OUT NS_UINT32 *retNumberOfNanoSeconds,
                IN OUT NS_UINT *overflowCount,
                IN OUT NS_UINT *sequenceId,
                IN OUT NS_UINT8 *messageType,
                IN OUT NS_UINT *hashValue );
```

Parameters

portHandle
: Handle that represents a port. This is obtained using the EPLEnumPort function.

retNumberOfSeconds
: Will be set on return to the next receive’s seconds timestamp value.

retNumberOfNanoSeconds
: Will be set on return to the next receive’s nanoseconds timestamp value. This value will never be larger then 1e9 (1 second).

overflowCount
: Will be set on return and indicates if timestamps were dropped due to an overflow of the Transmit Timestamp queue. The overflow counter will stick at a value of three if more then three timestamps were missed. Normally this value should be 0.

sequenceId
: Will be set on return to the16-bit SequenceId field from the incoming PTP frame.

messageType
: Will be set on return to the messageType field from the incoming PTP frame. For version 1 of the IEEE 1588 specification the Timestamp unit will return the lower four bits of the control field (octet 32 in the message). Otherwise, the Timestamp unit will record the version 2 messageType field which is the least significant bits of the first octet in the PTP message.

hashValue
: Will be set on return to a 12-bit hash value on octets 20-29 of the PTP event message. For version 1 of the IEEE 1588 specification, this corresponds to the messageType, sourceCommunicationTechnology, sourceUuid, and sourcePortId fields. For version 2 of the IEEE 1588 specification, this corresponds to the 10-octet sourcePortIdentity field. The combination of hash value and sequenceId, allows software to correctly match a Timestamp with the correct receive event message.

  The hash algorithm used is the CRC function as defined in section 3.2.8 the IEEE 802.3 specification. The Timestamp unit returns the 12 most significant bits as the CRC computation (the resultant bits are not complemented as done in the 802.3 CRC generation).

To minimize unnecessary timestamp capture, the device may be configured to filter based on the source identification hash value. This value may be programmed using the PTPSetMiscConfig() call with relevance to the ptpRxHash parameter. If the source hash value for the incoming PTP event message does not match the programmed hash value, then the message will not be timestamped.

Return Value

Nothing

Comments

The caller must have previously called PTPCheckForEvents() and determined that the PTPEVT_RECEIVE_TIMESTAMP_BIT bit was set prior to invoking this function. This function does NOT check to determine if an outstanding receive event is available.

The hardware can queue up to four receive timestamps.

This function does not adjust the timestamp to account for the delay from the wire. The caller should make the necessary adjustment(s) as needed. The typical adjustment is to subtract the appropriate default inbound latency delay value, either DEFAULT_INBOUND_LATENCY or DEFAULT_INBOUND_LATENCY_10MB which can be found in constants.h, from the timestamp.

4.4.5.2 PTPGetTimestampFromFrame

Extracts the embedded receive timestamp from a received PTP frame.

```
void
        PTPGetTimestampFromFrame (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT8 *receiveFrameData,
                IN OUT NS_UINT32 *retNumberOfSeconds,
                IN OUT NS_UINT32 *retNumberOfNanoSeconds );
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.
receiveFrameData
Points to the start of the PTP header.
retNumberOfSeconds
Will be set on return to the next receive’s seconds timestamp value. The number of significant bits of magnitude returned is determined by the RXOPT_TS_SEC_EN option bit used during the call to PSPSetReceiveConfig().
retNumberOfNanoSeconds
Will be set on return to the next receive’s nanoseconds timestamp value. This value will never be larger then 10^9 (1 second).

Return Value

Nothing

Comments

This function can be used when receive timestamp insertion has been enabled using the PSPSetReceiveConfig() function with the RXOPT_TS_INSERT option set. It extracts the embedded timestamp from the frame, either from the end of the frame if RXOPT_TS_APPEND is enabled, or from the configured locations within the PTP frame itself.

If RXOPT_TS_APPEND is disabled, the relevant reserved fields are set to a value of 0.

This function does not adjust the timestamp to account for the delay from the wire. The caller should make the necessary adjustment(s) as needed. The typical adjustment is to subtract the appropriate default inbound latency delay value, either DEFAULT_INBOUND_LATENCY or DEFAULT_INBOUND_LATENCY_10MB which can be found in constants.h, from the timestamp

4.4.6 IEEE 1588 Trigger APIs

The basic behavioral configuration of an individual trigger is set using the PTPSetTriggerConfig() function and may be changed at a later time if desired. The following functions arm (initiate) a trigger, cancel a trigger and check for a completed trigger event.

4.4.6.1 PTPArmTrigger

Establishes a trigger's expiration time and trigger pulse width behavior.

```
void
        PTPArmTrigger (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT trigger,
                IN NS_UINT32 expireTimeSeconds,
                IN NS_UINT32 expireTimeNanoSeconds,
                IN NS_BOOL initialStateFlag,
                IN NS_BOOL waitForRolloverFlag,
                IN NS_UINT32 pulseWidth,
                IN NS_UINT32 pulseWidth2 );
```

Parameters

portHandle
: Handle that represents a port. This is obtained using the EPLEnumPort function.

trigger
: The trigger to arm, 0 – 7.

expireTimeSeconds
: The seconds portion of the desired expiration time relative to the IEEE 1588 hardware clock.

expireTimeNanoSeconds
: The nanoseconds portion of the desired expiration time relative to the IEEE 1588 hardware clock. This value may be not be larger then 2^30 (1 second).

initialStateFlag
: Indicates initial state of signal to be set when trigger is armed (FALSE will cause a signal rise at trigger time, TRUE will cause a signal fall at trigger time). This parameter is ignored in toggle mode.

waitForRolloverFlag
: If set to TRUE the device should not arm the trigger until after the seconds field of the clock time has rolled over from 0xFFFF_FFFF to 0.

pulseWidth
: Sets the 50% duty cycle time for triggers 2 – 7. Its format is bits [31:30] = seconds, [29:0] = nanoseconds. For triggers 0 and 1 sets the width of the first part of the pulse, the width of the second part of the pulse is set by the pulseWidth2 parameter.

pulseWidth2
: Ignored for triggers 2 – 7. For Triggers 0 and 1, in a single or periodic pulse type signal, a second pulse width value controls the 2nd pulse width (period is pulseWidth + pulseWidth2). Its format is bits [31:30] = seconds, [29:0] = nanoseconds.

: For Edge type signals, pulseWidth2 is interpreted as a 16-bit seconds field and pulseWidth is a 30-bit nanoseconds field.

Return Value

None

Comments

Once this function has been called, the trigger will be armed.

4.4.6.2 PTPHasTriggerExpired

Determines if a particular trigger has occurred or if an error has occurred. An interrupt mechanism may also be used to detect trigger expiration.

```
NS_STATUS
        PTPHasTriggerExpired (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT trigger );
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

trigger
The trigger to check, 0 – 7.

Return Value

NS_STATUS_SUCCESS
The specified trigger has expired successfully.

NS_STATUS_FAILURE
The specified trigger has not yet expired.

NS_STATUS_INVALID_PARM
The specified trigger was armed late and expired. For a periodic signal, if the Trigger-if-late control is set, this function will return NS_STATUS_SUCCESS. If the Trigger-if-late is not set, this status return code will be returned.

Comments

It is NOT necessary to call PTPCheckForEvents() prior to invocation.

4.4.6.3 PTPCancelTrigger

Cancels a previously schedule trigger event (if active).

```
void
        PTPCancelTrigger (
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_UINT trigger );
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.
trigger
The trigger to cancel, 0 – 7.

Return Value

None

Comments

It is possible that a trigger will expire before this function can cancel it, depending on how close the expiration time is to the current IEEE 1588 time.

4.4.7 IEEE 1588 Event Timestamp APIs

Provides a function to obtain information about an outstanding timestamped event. Events are configured using the PTPSetEventConfig() function.

4.4.7.1 PTPGetEvent

Returns the next available asynchronous event timestamp.

```
NS_BOOL
        PTPGetEvent (
                IN PEPL_PORT_HANDLE portHandle,
                IN OUT NS_UINT *eventNum,
                IN OUT NS_BOOL *riseFlag,
                IN OUT NS_UINT32 *eventTimeSeconds,
                IN OUT NS_UINT32 *eventTimeNanoSeconds,
                IN OUT NS_UINT *eventsMissed );
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

eventNum
Set on return to the number of the event that occurred if an event was present, 0 – 7.

riseFlag
Set on return to TRUE if the event occurred on the rising edge, FALSE if it occurred on the falling edge.

eventTimeSeconds
The seconds portion of the IEEE 1588 clock timestamp when this event occurred.

eventTimeNanoSeconds
The nanosecond portion of the IEEE 1588 clock timestamp when this even occurred. This value will not be larger then 1e9 (1 second).

eventsMissed
Set on return to indicate the number of events that have been missed prior to this event due to internal event queue overflow. The maximum value is 7.

Return Value

TRUE if an event was returned, FALSE otherwise.

Comments

The caller must have previously called PTPCheckForEvents() and determined that the PTPEVT_EVET_TIMESTAMP_BIT bit was set prior to invoking this function. This function does NOT check to determine if an outstanding event is available.

This function properly tracks and handles events that occur at the same exact time. It also adjusts the timestamp values to compensate for input path and synchronization delays.

This function subtracts time from the event timestamp to account for the delay between the input pin and edge detection. This value is defined by PIN_INPUT_DELAY found in epl_1588.h

4.4.8 IEEE 1588 Interrupts

EPL does not provide higher level functions for PHY device interrupt handling. Because interrupt handling is highly environment specific it is necessary to interact directly with interrupt related registers from the host's PHY device driver. Refer to the device's data sheet for information on how to use the Phy's numerous interrupt capabilities.

4.4.9 Miscellaneous APIs

4.4.9.1 MonitorGpioSignals

Provides a function to read the current status of the device's GPIO signals.

```
NS_UINT
        MonitorGpioSignals (
                IN PEPL_PORT_HANDLE portHandle );
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

Return Value

Returns bits [11:0] reflecting the current values on the 12 GPIOs. GPIO 12 is bit 11, etc.

Comments

None

4.4.10 PHY Status Frame Processing APIs

If enabled, the PHY can present various event and status information in the form of specially formatted IEEE 1588 frames to the MAC. These are formatted so that they are ultimately passed to the host's PTP stack. This section provides a set of functions that can be used to determine if a frame is a PHY Status Frame (PSF) and to provide de-multiplexing of the message's information. This eliminates the need for host software to parse incoming frames directly to detect a PSF and simplifies obtaining underlying PHY message information.

4.4.10.1 IsPhyStatusFrame

Determines if the specified receive frame is a specially formatted IEEE 1588 PHY Status Frame (PSF).

```
EXPORT NS_UINT8 *
          IsPhyStatusFrame (
            IN PEPL_PORT_HANDLE portHandle,
            IN NS_UINT8 *frameBuffer,
            IN NS_UINT16 frameLength)
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

frameBuffer
Pointer to a byte array containing a pointer to the incoming IEEE 1588 frame. This should point to the start of the Ethernet frame.

frameLength
The number of bytes pointed to by frameBuffer.

Return Value

If a valid message is found a pointer to the Phy message that should be processed will be returned. Otherwise NULL will be returned.

Comments

This function used the previously configured values for the IEEE 1588 header transportSpecific, messageType and versionPTP to determine if the frame is a PHY Status Frame. Refer to PTPSetPhyStatusFrameConfig().

If the frame is a PSF message, the caller should use the GetNextPhyMessage() to obtain the message's information.

4.4.10.2 GetNextPhyMessage

Returns information about the next PHY message contained in a list of one or more PHY message structures.

```
EXPORT NS_UINT8 *
        GetNextPhyMessage (
          IN PEPL_PORT_HANDLE portHandle,
          IN OUT NS_UINT8 *msgLocation,
          IN OUT PHYMSG_MESSAGE_TYPE_ENUM *messageType,
          IN OUT PHYMSG_MESSAGE *phyMessageOut)
```

Parameters

portHandle
Handle that represents a port. This is obtained using the EPLEnumPort function.

msgLocation
Pointer to the PHY message to check/process. This is initially determined by calling IsPhyStatusFrame(). The caller should treat this field opaquely.

messageType
Set on return to one of the PHYMSG_MESSAGE_TYPE_ENUM values indicating the type of message structure returned in the message union buffer.

phyMessageOut
Caller allocated union/structure that will be filled out on return to contain the relevant message information fields.

Return Value

Returns a pointer to the PHY message location to be checked/processed. Returns NULL if there are no further messages to process.

Comments

Message Type	**Description**
PHYMSG_STATUS_TX	Transmit Timestamp Status Message
PHYMSG_STATUS_RX	Receive Timestamp Status Message
PHYMSG_STATUS_TRIGGER	Trigger Status Message
PHYMSG_STATUS_EVENT	Event Timestamp Status Message
PHYMSG_STATUS_ERROR	Error Status Message
PHYMSG_STATUS_REG_READ	Register Read Results Message

Table 4.4-7 – PHYMSG_MESSAGE_TYPE_ENUM

The PHYMSG_MESSAGE structure consists of a union of the following structures. It is defined as follows:

```
typedef union PHYMSG_MESSAGE {
    struct {
        NS_UINT32 txTimestampSecs;
        NS_UINT32 txTimestampNanoSecs;
        NS_UINT8  txOverflowCount;          // 2-bit value in 8-bit variable
    } TxStatus;

    struct {
        NS_UINT32 rxTimestampSecs;
        NS_UINT32 rxTimestampNanoSecs;
        NS_UINT8  rxOverflowCount;          // 2-bits data in 8-bit variable
        NS_UINT16 sequenceId;               // 16-bits
        NS_UINT8  messageType;              // 4-bits data in 8 bit variable
        NS_UINT16 sourceHash;               // 12-bit data in 16-bit variable
    } RxStatus;

    struct {
        NS_UINT16 triggerStatus;           // Bits [11:0] indicating what
    } TriggerStatus;                       // triggers occurred (12 - 1
respectively).

    struct {
        NS_UINT16 ptpEstsRegBits;           // 12-bits - See PTP_ESTS register defs
        NS_BOOL   extendedEventStatusFlag;  // 8-bits - TRUE if ext. info available
        NS_UINT16 extendedEventInfo;        // See register definition for more info
        NS_UINT32 evtTimestampSecs;
        NS_UINT32 evtTimestampNanoSecs;
    } EventStatus;

    struct {
        NS_BOOL frameBufOverflowFlag;       // 8-bits
        NS_BOOL frameCounterOverflowFlag;   // 8-bits
    } ErrorStatus;

    struct {
        NS_UINT8 regIndex;                  // 5-bits data in 8-bit variable
        NS_UINT8 regPage;                   // 3-bits data in 8-bit variable
        NS_UINT16 readRegisterValue;        // 16-bits
    } RegReadStatus;
} PHYMSG_MESSAGE;
```

4.5 OS Abstraction Interface (OAI)

This section defines the OAI functions that must be implemented for each target platform. The prototypes and definitions for these functions are provided in the **oai.h** file. The OAI must provide the following functions.

- OAIInitialize
- OAIAlloc
- OAIFree
- OAICreateMutex
- OAIReleaseMutex
- OAIBeginCriticalSection
- OAIEndCriticalSection
- OAIBeginRegCriticalSection
- OAIEndRegCriticalSection
- OAIBeginMultiCriticalSection
- OAIEndMultiCriticalSection
- OAIManagementError (DP83640, DP83630, and DP83620 devices only)

4.5.1 OAIInitialize

Called by EPL to initialize the OAI module.

```
void
        OAIInitialize(
                IN OAI_DEV_HANDLE oaiDevHandle);
```

Parameters

oaiDevHandle
Handle that represents the MDIO bus that the write operation should occur on. The definition of this is completely up to higher layer software.

Return Value

Nothing

Comments

None

4.5.2 OAIAlloc

Provides basic memory allocation.

```
void *
        OAIAlloc(
                IN sizeInBytes);
```

Parameters

sizeInBytes
Size of the memory block to allocate in bytes.

Return Value

Returns a pointer to the allocated memory block. If a failure occurred allocating the memory, NULL is returned.

Comments

Memory allocated internally by the library will be freed automatically but it is up to the application to free any memory that it allocates using this function.

4.5.3 OAIFree

Frees a memory block allocated using the OAIAlloc function.

```
void
        OAIFree(
                IN void *memPtr);
```

Parameters

memPtr
Pointer to the memory block to free.

Return Value

Nothing

Comments

None

4.5.4 OAICreateMutex

Creates a handle for a mutex object that can be used to provide mutual exclusion.

```
HANDLE
        OAICreateMutex(
                void);
```

Parameters

None

Return Value

Returns a handle to a mutex object

Comments

This can be called to create additional mutex objects if needed.

4.5.5 OAIReleaseMutex

Provides basic memory allocation.

```
void *
        OAIReleaseMutex(
                IN HANDLE hMutex );
```

Parameters

hMutex
: HANDLE to mutex object previously created.

Return Value

Nothing

Comments

None

4.5.6 OAIBeginRegCriticalSection

Provides mutual exclusion for a multi-step device access operation.

```
void
        OAIBeginRegCriticalSection(
                IN OAI_DEV_HANDLE oaiDevHandle);
```

Parameters

oaiDevHandle

Handle that represents the MDIO bus that the write operation should occur on. The definition of this is completely up to higher layer software.

Return Value

Nothing

Comments

This function uses the **regularMutex** member of the **oaiDevHandle** defined in **platform.h**

In a multi-threaded environment where multiple threads may access the EPL functions simultaneously, this function must provide some type of lock that prevents multi-step register access EPL APIs from being reentered.

4.5.7 OAIEndRegCriticalSection

Releases a previously held lock that was obtained by a call to OAIBeginRegCriticalSection.

```
void
        OAIEndRegCriticalSection(
                IN OAI_DEV_HANDLE oaiDevHandle);
```

Parameters

oaiDevHandle
Handle that represents the MDIO bus that the write operation should occur on. The definition of this is completely up to higher layer software.

Return Value

Nothing

Comments

This function uses the **regularMutex** member of the **oaiDevHandle** defined in **platform.h**

4.5.8 OAIBeginMultiCriticalSection

Provides mutual exclusion for a multi-step device access operation.

```
void
        OAIBeginMultiCriticalSection(
                IN OAI_DEV_HANDLE oaiDevHandle);
```

Parameters

oaiDevHandle

Handle that represents the MDIO bus that the write operation should occur on. The definition of this is completely up to higher layer software.

Return Value

Nothing

Comments

This function uses the **multiOpMutex** member of the **oaiDevHandle** defined in **platform.h**

In a multi-threaded environment where multiple threads may access the EPL functions simultaneously, this function must provide some type of lock that prevents multi-step register access EPL APIs from being reentered.

4.5.9 OAIEndMultiCriticalSection

Releases a previously held lock that was obtained by a call to OAIBeginMultiCriticalSection.

```
void
        OAIEndMultiCriticalSection(
                IN OAI_DEV_HANDLE oaiDevHandle);
```

Parameters

oaiDevHandle

Handle that represents the MDIO bus that the write operation should occur on. The definition of this is completely up to higher layer software.

Return Value

Nothing

Comments

This function uses the **multiOpMutex** member of the **oaiDevHandle** defined in **platform.h**

4.5.10 OAIManagementError

EPL calls this function if a data integrity error is detected over the management information interface.

```
void
        OAIManagementError (
                IN OAI_DEV_HANDLE oaiDevHandle );
```

Parameters

oaiDevHandle

Handle that represents the MDIO bus that the error occurred on. The definition of this is completely up to higher layer software.

Return Value

Nothing

Comments

If a checksum mismatch between host software’s checksum tally and the hardware’s checksum tally for register reads and writes, this function is called. It indicates that one or more bits read from or written to the PHY were corrupted. This would normally indicate a serious system error and should be dealt with by the host environment as necessary.

4.6 Interface Function Reference

4.6.1 ALP 100/Opal Kelly Interface Related Functions

This provides the necessary functions to communicate with the ALP 100 board using the Opal Kelly Front Panel interface. The **okMAC.c/.h** files provide the wrapper for EPL to communicate through the **okFrontPanel.dll** interface.

4.6.1.1 MACInitialize

Called to initialize the interface and prepare it for operation.

```
NS_STATUS
        MACInitialize(
                IN OAI_DEV_HANDLE oaiDevHandle);
```

Parameters

oaiDevHandle
Handle that represents the MDIO bus that the write operation should occur on. The definition of this is completely up to higher layer software.

Return Value

NS_STATUS_SUCCESS
Function was successfully executed.
NS_STATUS_FAILURE
Function was not able to initialize the interface. This usually means that a device was not found.

Comments

This call is not usually called directly. It is typically called from the EPLEnumDevice() function if it hasn’t already been initialized.
The input to this call is an OAI_DEV_HANDLE. The connector member of that structure is used to define which connector on the ALP board to look for a PHY device. NS_STATUS_FAILURE will be returned if a device isn’t found.

4.6.1.2 MACDeInitialize

This is called to shutdown ALP 100 / Opal Kelly Interface

```
void
        MACDeInitialize(
                IN OAI_DEV_HANDLE oaiDevHandle);
```

Parameters

oaiDevHandle
Handle that represents the MDIO bus that the write operation should occur on. The definition of this is completely up to higher layer software.

Return Value

Nothing

Comments

This call is not usually called directly. It is typically called from the EPLDeInitialize() function as part of the overall EPL shutdown process.

4.6.1.3 MACSendPacket

This is called to send a packet of data through the ALP 100 / Opal Kelly Interface.

```
void
        MACSendPacket(
                IN PEPL_PORT_HANDLE eplPortHandle,
                IN NS_UINT8 *txBuf,
                IN NS_UINT  length);
```

Parameters

eplPortHandle
Handle to the port to send the data.
txBuf
Pointer to the packet of data to send.
length
Length of the data packet to be sent.

Return Value

Nothing

Comments

This is called to send a packet of data through the ALP 100 FPGA’s MAC. It calculates and sets the IP & UDP checksum fields.

4.6.1.4 MACSendPacketNoUpdChecksum

This is called to send a packet of data through the ALP 100 / Opal Kelly Interface without updating the UDP checksum field.

```
void
        MACSendPacketNoUdpChecksum(
                IN PEPL_PORT_HANDLE eplPortHandle,
                IN NS_UINT8 *txBuf,
                IN NS_UINT  length);
```

Parameters

eplPortHandle
Handle to the port to send the data.
txBuf
Pointer to the packet of data to send.
length
Length of the data packet to be sent.

Return Value

Nothing

Comments

This is called to send a packet of data through the ALP 100 FPGA's MAC. This is the same as MACSendPacket() except it only calculates and sets the IP checksum field.

4.6.1.5 MACFlushReceiveFifos

This is called to send flush out all the receive data.

```
void
        MACFlushReceiveFifos(
                IN PEPL_PORT_HANDLE eplPortHandle);
```

Parameters

eplPortHandle
Handle to the port to send the data.

Return Value

Nothing

Comments

This procedure effectively flushes out all receive data by calling MACReceivePacket() until there is no data left.

4.6.1.6 MACReceivePacket

This is called to check for any data on the interface

```
NS_UINT
        MACReceivePacket(
                IN PEPL_PORT_HANDLE eplPortHandle
                IN NS_UINT8 *rxBuf,
                IN NS_UINT  length);
```

Parameters

eplPortHandle
Handle to the port to send the data.

rxBuf
Pointer to a buffer that will be filled in if any data is available

length
Length of the data packet to be sent.

Return Value

TRUE if data is available
FALSE if no data is available

Comments

None

4.6.1.7 MACReadReg

This is called read a PHY register using the FPGA MDIO interface.

```
NS_UINT
        MACReadReg(
                IN NS_UINT mdioAddr,
                IN OAI_DEV_HANDLE oaiDevHandle,
                IN NS_UINT  regIndex);
```

Parameters

mdioAddr
: MDIO address to use for the transaction.

oaiDevHandle
: Handle that represents the MDIO bus that the write operation should occur on. The definition of this is completely up to higher layer software.

regIndex
: Index of the register to read. Bits 7:5 select the register page (000-pg0, 001-pg1, 010-pg3, 011-pg4, etc.).

Return Value

TRUE if data is available
FALSE if no data is available

Comments

Issues a direct register read operation. Host software must implement this function as a synchronous operation.

4.6.1.8 MACWriteReg

This is called write a PHY register using the FPGA MDIO interface.

```
void
        MACWriteReg(
                IN NS_UINT mdioAddr,
                IN OAI_DEV_HANDLE oaiDevHandle,
                IN NS_UINT  regIndex,
                IN NS_UINT  value);
```

Parameters

mdioAddr
MDIO address to use for the transaction.

oaiDevHandle
Handle that represents the MDIO bus that the write operation should occur on. The definition of this is completely up to higher layer software.

regIndex
Index of the register to read. Bits 7:5 select the register page (000-pg0, 001-pg1, 010-pg3, 011-pg4, etc.).

value
The value to write.

Return Value

Nothing

Comments

None

4.6.1.9 MacMIIReadReg

Issues a read request PHY control operation through the environment's MAC interface.

```
NS_UINT
        MacMIIReadReg (
                IN NS_UINT mdioAddr,
                IN OAI_DEV_HANDLE oaiDevHandle,
                IN NS_UINT8 *readRegRequestPacket,
                IN NS_UINT length );
```

Parameters

mdioAddr
MDIO address to use for the transaction.

oaiDevHandle
Handle that represents the MDIO bus that the write operation should occur on. The definition of this is completely up to higher layer software.

readRegRequestPacket
Fully formatted packet buffer containing a register read PHY Control Frame (PCF).

length
Specifies the length of the readReqRequestPacket buffer in bytes.

Return Value

Returns the value read from the register.

Comments

Host software must implement this function as a synchronous operation. This function is only required if the system is supposed to implement PHY Control Frames.

The hardware only supports a single outstanding read request so there should only be 1 PCF in the request and length should be 4.

NOTE: The PCF request packets need to be formatted in network order. For example, if the 32-bit value is 0x42001122 the packet would be created as follows:

```
*request + 0 = 0x42
*request + 1 = 0x00
*request + 2 = 0x11
*request + 3 = 0x22
```

4.6.1.10 MacMIIWriteReg

Issues a write operation PHY control operation through the environment's MAC interface.

```
void
        MacMIIWriteReg (
                IN NS_UINT mdioAddr,
                IN OAI_DEV_HANDLE oaiDevHandle,
                IN NS_UINT8 *writeRegRequestPacket,
                IN NS_UINT length );
```

Parameters

mdioAddr
MDIO address to use for the transaction.

oaiDevHandle
Handle that represents the MDIO bus that the write operation should occur on. The definition of this is completely up to higher layer software.

writeRegRequestPacket
Fully formatted packet buffer containing a register write PHY Control Frame (PCF). This frame may contain more then one register write request.

length
Specifies the length of the writeReqRequestPacket buffer in bytes.

Return Value

Nothing

Comments

Host software must implement this function as a synchronous operation. This function is only required if the system is supposed to implement PHY Control Frames.

The hardware supports multiple write requests so any number of PCFs can be concatentated together and the length should be a multiple of 4 (i.e. 4 times the number of PCFs in the request.)

NOTE: The PCF request packets need to be formatted in network order. For example, if the 32-bit value is 0x42001122 the packet would be created as follows:

```
*request + 0 = 0x42
*request + 1 = 0x00
*request + 2 = 0x11
*request + 3 = 0x22
```

4.6.1.11 SetDuplex

This procedure is called to set the FPGA’s MAC interface duplex mode.

```
void
        SetDuplex(
                IN PEPL_PORT_HANDLE portHandle,
                IN NS_BOOL halfDuplex);
```

Parameters

eplPortHandle
Handle to the port to configure
halfDuplex
Set to TRUE to enable half duplex or FALSE to enable full duplex.

Return Value

Nothing

Comments

None

4.6.2 Cypress USB Related Functions

This provides the necessary functions to communicate with the ALP Nano board using the Cypress USB interface. Since the hardware only supports the MDIO interface the only functionality provided through this interface is what is available using PHY register reads and writes. The **ifCyUSB.cpp/.h** files provide the wrapper for EPL to communicate through the **CyAPI.lib** interface.

4.6.2.1 ifCyUSB_Init

Called to initialize the interface and prepare it for operation.

```
NS_STATUS
    ifCyUSB_Init(
        IN OAI_DEV_HANDLE oaiDevHandle);
```

Parameters

oaiDevHandle
Handle that represents the MDIO bus that the write operation should occur on. The definition of this is completely up to higher layer software.

Return Value

NS_STATUS_SUCCESS
Function was successfully executed.
NS_STATUS_FAILURE
Function was not able to initialize the interface. This usually means that a device was not found.

Comments

This call is not usually called directly. It is typically called from the EPLEnumDevice() function if it hasn't already been initialized.

4.6.2.2 ifCyUSB_DeInit

This is called to shutdown interface.

```
void
        ifCyUSB_DeInit(
                IN OAI_DEV_HANDLE oaiDevHandle);
```

Parameters

oaiDevHandle
Handle that represents the MDIO bus that the write operation should occur on. The definition of this is completely up to higher layer software.

Return Value

Nothing

Comments

This call is not usually called directly. It is typically called from the EPLDeInitialize() function as part of the overall EPL shutdown process.

4.6.2.3 ifCyUSB_ReadMDIO

This is called read a PHY register using MDIO off of the Cypress USB device.

```
NS_UINT
        ifCyUSB_ReadMDIO(
                IN PEPL_PORT_HANDLE portHandle
                IN NS_UINT  regIndex);
```

Parameters

portHandle
Handle to the port to send the data.
regIndex
Index of the register to read. Bits 7:5 select the register page (000-pg0, 001-pg1, 010-pg3, 011-pg4, etc.).

Return Value

TRUE if data is available
FALSE if no data is available

Comments

Issues a register read operation through the Cypress USB interface. Host software must implement this function as a synchronous operation.

4.6.2.4 ifCyUSB_WriteMDIO

This is called write a PHY register using MDIO off of the Cypress USB device.

```
void
        ifCyUSB_WriteMDIO(
                IN PEPL_PORT_HANDLE portHandle
                IN NS_UINT  regIndex
                IN NS_UINT  value);
```

Parameters

portHandle
Handle to the port to send the data.
regIndex
Index of the register to read. Bits 7:5 select the register page (000-pg0, 001-pg1, 010-pg3, 011-pg4, etc.).
value
The value to write.

Return Value

Nothing

Comments

None

4.6.3 LPT Related Functions

This provides the necessary functions for implementing an MDIO bit banging interface on an LPT port. Depending on the OS environment the use of this code requires additional driver support in order to be able to access the LPT port directly. **The use of this interface is discouraged and provided only as an example of what could be done if other options are not available.**

4.6.3.1 ifLPTReadReg

This is called read a PHY register using MDIO off of the LPT interface

```
NS_UINT
        ifLPTReadReg(
                IN PEPL_PORT_HANDLE portHandle
                IN NS_UINT  regIndex);
```

Parameters

portHandle
Handle to the port to send the data.
regIndex
Index of the register to read. Bits 7:5 select the register page (000-pg0, 001-pg1, 010-pg3, 011-pg4, etc.).

Return Value

TRUE if data is available
FALSE if no data is available

Comments

Issues a register read operation through the LPT interface. Host software must implement this function as a synchronous operation.

4.6.3.2 ifLPTWriteReg

This is called write a PHY register using MDIO off of the LPT interface

```
void
        ifLPTWriteReg(
                IN PEPL_PORT_HANDLE portHandle
                IN NS_UINT  regIndex
                IN NS_UINT  value);
```

Parameters

portHandle
Handle to the port to send the data.

regIndex
Index of the register to read. Bits 7:5 select the register page (000-pg0, 001-pg1, 010-pg3, 011-pg4, etc.).

value
The value to write.

Return Value

Nothing

Comments

None

4.6.3.3 ifLPTMdioReadBit

Reads one bit from the MDIO bus.

```
NS_BOOL
        ifLPTMdioReadBit(
                IN OAI_DEV_HANDLE oaiDevHandle);
```

Parameters

oaiDevHandle
Handle that represents the MDIO bus that the read operation should occur on.

Return Value

TRUE if bit was asserted(1), FALSE otherwise (0).

Comments

This function must assert MDC, read the MDIO state, then de-assert MDC. In general there is a maximum MDIO clocking frequency of 25 MHz (40ns clock). If the environment could possibly allow MDIO clocking faster then this, this function should include a delay of at least 40ns so that the max. clocking frequency is not violated.

Normally this function isn't accessed directly. The higher level ifLPTReadReg() procedure would be used to initiate a complete read sequence using this function.

4.6.3.4 ifLPTMdioWriteBit

Writes one bit on the MDIO bus.

```
void
        ifLPTMdioWriteBit(
                IN OAI_DEV_HANDLE oaiDevHandle,
                IN NS_BOOL bit);
```

Parameters

oaiDevHandle
Handle that represents the MDIO bus that the write operation should occur on. The definition of this is completely up to higher layer software.

bit
TRUE if a 1 should be written on the MDIO line, FALSE if a 0 should be written.

Return Value

Nothing

Comments

This function must assert MDC, drive the MDIO input line as specified, then de-assert MDC. In general there is a maximum MDIO clocking frequency of 25 MHz (40ns clock). If the environment could possibly allow MDIO clocking faster then this, this function should include a delay of at least 40ns so that the max. clocking frequency is not violated.

Normally this function isn’t accessed directly. The higher level ifLPTWriteReg() procedure would be used to initiate a complete write sequence using this function.

5 Test, Example, and Demonstration Code

The library contains several examples applications and scripts that provide some hints and direction for how the library is intended to be used. This section provides an overview of those examples.

NOTE: The applications are intended to provide complete working examples of the setup and operation of the library and stack functionality. While an attempt has been made to make reasonable choices, the values or sequences used are not necessarily what would be used in a production system for optimal performance.

5.1 EPLTestApp

5.1.1 Overview

EPLTestApp is a simple application that is designed to demonstrate the usage of the EPL core functionality. All of the major functions are executed to show how to setup for the calls and how to parse the results. As noted above a normal production system would probably not use the calls in this order or in this way.

The application provides an example of you to scan for PHY devices using the library. Once a device is found the **RunTests()** procedure is called to run all of the tests on the device.

5.1.2 Operational Details

This section provides highlights of how the application works.

5.1.2.1 EPL Initialize/Deinitialize

The library much be initialized before any other operations and deinitialized before leaving the application. The EPLInitialize() call provides the opportunity to get the library setup for operations. The EPLDeinitialize() function performs necessary cleanup operations like freeing memory that was allocated for library structures. It does NOT free memory allocated by the application using the OAIAlloc() function.

5.1.2.2 Scanning For Devices

This application shows an example of how to scan for devices. The library and example are built to support the ALP/Opal Kelly board so the process is specific to that environment. To support that interface the library is setup to support 3 levels (board, connector, and MDIO Address) of scanning. For other hardware solutions the library can be expanded by adding another interface type and either using the existing levels or changing the OAI_DEV_HANDLE_STRUCT to include members that are relevant for that interface.

The basic scan process is as follows:

```
#define MAX_BOARD     2
#define MAX_CONNECTOR 4
#define MAX_MDIO_ADDR 31
for( nBrd=nBoard; nBrd < MAX_BOARD; nBrd++ ) {
    for( nConn=nConnector; nConn < MAX_CONNECTOR; nConn++ ) {
        for( nMDIO=nMDIOAddr; nMDIO < MAX_MDIO_ADDR; nMDIO++ ) {
            // Initialize structure for search
            memset( oaiDH, 0x00, sizeof( OAI_DEV_HANDLE_STRUCT ) );
            oaiDH->pcfDefault = bPCF;
            oaiDH->board = nBrd;
            oaiDH->connector = nConn;
            // Look for devices using the current enumeration type
            devHandle = EPLEnumDevice( oaiDH, nMDIO, iType );
            if ( devHandle ){   // Did we get something back?
                // Got a device, find out more
                devInfo = EPLGetDeviceInfo( devHandle );
                if( devInfo ){  // Did we get valid info back?
                    // Got info parse it out and run the tests
                    if( devInfo->numOfPorts == 0 ) {
                        printf( "ERROR: No ports on device b:%d c:%d a:%d\n",
                                 nBrd+1, nConn+1, nMDIO );
                        continue;
                    }  // if( devInfo->numOfPorts == NULL )
                    RunTests( devHandle );
                    // Move onto next device
                    nMDIO += devInfo->numOfPorts;
                }  // if( devInfo )
            }  // if( defHandle )
        } // for( nMDIO... )
    } // for( nConn... )
} // for( nBrd )
```

In the example above oaiDH is a pointer to a OAI_DEV_HANDLE_STRUCT and is used to tell hold information used by the library in communicating with the device. As noted above in a real world solution the structure will probably be modified as needed by a newly created interface module.

oaiDH->pcfDefault is used by the application to signal to the library if PHY Control Frames (PCFs) and PHY Status Frames (PSFs) should be used exclusively to access PHY registers. This is necessary where the device is not connected to an MDIO level interface. The device must be properly strapped (GPIO2 pulled high) in order for it to recognize PCFs for write operations. Once writes are successful, PSFs can be configured to allow read response to be generated by the part. **NOTE: Care must be taken when using certain EPL calls such as EPLResetDevice(). Once the device is reset it will no longer support register reads using the PCFs/PSFs until it is reconfigured.**

5.1.2.3 Running Tests

Once a valid device is found the **RunTests()** function is called to make all of the calls into the library. It is pretty straight forward and provides examples of how to setup for the calls and how to parse the results.

See the actual code for details. The code calls all the major functions of the library and can be expanded or stripped down to perform any specific test or task.

5.1.3 Command Line Parameters

EPLTestApp is setup to take some command line parameters. The actual parameters can be obtained by running the application with a -?:

```
-----------------------------------------------------------------------------
*** EPL Test Application v1.90  Build: 1
-----------------------------------------------------------------------------

EPLTestApp [  -H | -? | -V | -S | -D bb cc aa ]
Commands:
  -H or -h or -?   Display this help
  -V or -v         Verbose Mode where
  -S or -s         Scan only (don't run tests)
  -D bb cc aa      Device where:
                   bb is the board number
                   cc is the connector number
                   aa is the address number
```

-V – Verbose mode adds calls to the TDR Oscope functions.
-S – Scan only mode will scan through board, connector, and MDIO Address to identify what devices are available but will not actually call **RunTests()** for each device.
-D – Device option that allows you to specify a specific board, connector, and MDIO Address to try and test.

If –D or –S options aren't specified the program will iterate through all board, connector, and MDIO Addresses from 0 to MAX_BOARD, MAX_CONNECTOR, and MAX_MDIO_ADDR as defined in the code. It will then call **RunTests()** for each device found.

5.2 EPLTest.py

5.2.1 Overview

EPLTest.py is a simple test script that is designed to demonstrate the usage of the EPL core functionality from Python. It roughly follows the sequence in EPLTestApp. The main purpose is to show how the library can be used from Python as some of the calls are slightly different. All of the major functions are executed to show how to setup for the calls and how to parse the results. As noted above a normal production system would probably not use the calls in this order or in this way.

5.2.2 Operational Details

The script is designed to be called from a command prompt that has a "PATH" that includes the Python executables. A sample command file (**testit.cmd**) is provided that contains the following to run the script:

```
@setlocal
@set PATH=c:\Program Files\python24
@set PYTHONPATH=.\
@python EPLTest.py %1 %2 %3 %4 %5
@endlocal
```

While the command file allows for passing parameters into the script the script currently isn't setup to do anything with parameters.

5.3 PTPTestApp

5.3.1 Overview

PTPTestApp is a simple application that is designed to demonstrate the usage of the PTP functionality provided by the library. The main goal of the program is to provide an example of how to get the PTP stack configured and operational. Since most of the PTP functions are called internally by the init and PTP stack they are not called by directly by PTPTestApp. The functionality provided by PTPTestApp is designed to be similar to what is available in the ALP Framework Demo application. As noted above a normal production system would probably not use the calls in this order or in this way.

The application is kept as simple as possible to show the operation of the system. In order to keep it simple it doesn't create a graphical interface and doesn't use threads. Due to the simplistic nature there are some limitations and restrictions on how it can be run. For example, because it doesn't use formal threading techniques the master and slave operations must be run in separate command line sessions. This provides a good balance of simple design and demonstration of functionality. See below for more details on running the application.

5.3.2 Running The Application

The application is a simple Windows console application. In order to run 2 sessions for a master and a slave it is necessary to run each in a separate command window. Using command line parameters you can specify which device (board, connector, and MDIO address) to use and whether it should be a master or a slave. See the Command Line Parameters section below for details.

Each application connects to the EPL.DLL and receives a separate data space. So even though there is only one version of the DLL loaded to each application it appears as a separate version. There is code in the DLL that allow it to share data between all the applications connected to it but it isn't used at the moment. The main reason it isn't used is because there are issues with the Opal Kelly (OK) DLL. The OK DLL doesn't allow multiple instances (HANDLES) to communicate with the same board. While shared memory allows EPL to use the same OK DLL handle it still doesn't work because the second instance of the application is a separate Windows process and Windows enforce protect that prevents the second application for accessing data owned by the first. This could all be overcome if the application was setup with to start separate threads for master and slave that would be owned by the master process similar to the way the ALP Framework does. However as explained earlier that would add an extra layer of complexity to the application and wouldn't enhance its ability to demonstrate how to configure and start the stack. See below for the hardware configuration that is necessary to run the application.

Once the hardware requirements are met the application can operate as a master or a slave and can interact with other sessions of the application or with the ALP Framework Application.

5.3.3 Hardware Requirements

In order to run the application as both a master and a slave it is necessary to have 2 separate ALP boards and DP836x0 modules. This allows each application to have access to the separate device using separate instances/handles to the OK DLL. Each board should be configured to allow MDIO access from the FPGA “MAC“ (both jumpers on J22 set for PMDIO_MII) and GPIO2 should be pulled high for PCF/PSF operations.

5.3.4 Command Line Parameters

The application is setup to take some command line parameters. The actual parameters can be obtained by running the application with a -?:

```
-----------------------------------------------------------------------------
*** PTP Test Application v1.90  Build: 1
-----------------------------------------------------------------------------

PTPTestApp [  -H | -? | -V | -F | -MM | -MS | -D bb cc aa | -T[nnnnn] ]
Commands:
  -H or -h or -?   Display this help
  -V or -v         Verbose Mode where
  -F or -f         Find all devices
  -MM or -mm       Mode Master (Default)
  -MS or -ms       Mode Slave
  -D bb cc aa      Device where:
                   bb is the board number
                   cc is the connector number
                   aa is the address number
  -T[nnnnn]        Update Time where nnnnn is the interval in ms. 1000 is default
```

-V – Enables verbose mode and displays all messages coming into the stdioCallback function.
-F – Causes the program to scan for devices without actually initializing them.
-MM – Sets operation for MASTER MODE
-MS – Sets operation for SLAVE MODE
-D bb cc aa – Allows you to specify a specific device by board, connector, and address
-T[nnnn] – Tells the program to periodically make calls to PTPReadCurrentTime() and display the local time from the device. NOTE: the time update operation is very simple and relies on messages coming through the stdioCallback interface to work. Therefore, if no messages are being sent from the PTP Stack the time update won't happen.

Typical parameters for setting up a MASTER on the 1st board connector 2 and MDIO address 1 are as follows:

PTPTestApp –d 1 2 1 –v –mm –T1000

Typical parameters for setting up a SLAVE on the 2nd board connector 4 and MDIO address 1 are as follows:

PTPTestApp –d 2 4 1 –v –ms –T1000

5.3.5 Run Time Keystrokes

The application as a very simple keystroke handler built in that can control the application while it is running. NOTE: the time update operation is very simple and relies on messages coming through the stdioCallback interface to work. Therefore, if no messages are being sent from the PTP Stack the time update won't happen. The following keys are supported:

<ESC> - This will cause the application to call PTPKillThread() which will signal to the PTP Stack that it should shutdown so the earlier call to PTPThreadC() will return and the application will exit.

<R> - This will cause the application to call PTPSetClock(0, 0) which will set the local time to 0.

<Shift><R> - This will call PTPKillThread() which will return from PTPThreadC() but instead of exiting it will reconfigure and restart the stack by calling configurePTP() and startPTP(). This is essentially the same as the Reset button in the ALP Framework GUI.

<T> - This will toggle on or off the update/display of the local time periodically.

<V> - This will toggle on or off the display of messages in the stdioCallback. This can be used to turn on/off the stream of message coming from the PTP Stack.

<P> - This will toggle on or off the display of PSF messages in the statusUpdateCallback function.

<S> - This will toggle on or off display of offset statistics in the statusUpdateCallback function.

5.3.6 Scanning For Devices

The process of scanning for devices is essentially the same as EPLTestApp. For additional details refer to the section above and the actual PTPTestApp source code.

5.3.7 PTP Initialization and Startup

After scanning for the device and initializing the communication path to the device the application calls configurePTP() to initializes the data structures necessary to properly configure the PTP Stack.

The main structure that is used by the stack is the RunTimeOpts structure. See the RunTimeOpts section below for a summary of the various structure members.

Once the structure is filled a call to startPTP() is made. Currently startPTP()simply calls PTPThreadC() in the library to start up the stack. startPTP() could be expanded to create and initialize a separate thread for the stack.

PTPThreadC() calls PTPInitHardware() to configure the stack using many of the PTP calls documented above to finish the device configuration. Once the device is configured it calls protocol() which is the PTP stacks main loop. It remains in this loop until PTPKillThread() is called to signal a shutdown.

5.3.8 RunTimeOpts

The RunTimeOpts structure is the main structure that is used to configure and control the operation of the PTP stack. Some of the members are used for configuration while others are used to in the internal operation. This section documents what they are intended for. Use the source to determine exact usage and implementation:

- **Boolean revA1SiliconFlag;** - This is used internally to apply bug fixes for A1 silicon. It is automatically detected and doesn't need to be initialized by the application.
- **Integer16 syncInterval; -** This defines the interval between sync messages. This should be set the same for master and slave.
- **Octet subdomainName[PTP_SUBDOMAIN_NAME_LENGTH]; -** This is used to define the PTP subdomain that the device lives on. Default value is "_DFLT" with alternates of "_ALT1", "_ALT2", "_ALT3" as defined in **constants.h**
- **Octet clockIdentifier[PTP_CODE_STRING_LENGTH]; -**This defines the PTP clock identifier. Default value is defined by IDENTIFIER_DFLT with others defined in **constant.h**.
- **UInteger32 clockVariance; -** This is the PTP clock variance. Default is defined by DEFAULT_CLOCK_VARIANCE in **constants.h**
- **UInteger8 clockStratum;** - This is the PTP clock stratum and defaults to DEFAULT_CLOCK_STRATUM defined in **constant.h**
- **Boolean clockPreferred;** - True or false value to specify whether or not this is the preferred clock. This value is used in the BMC algorithm.
- **Integer16 currentUtcOffset; -** This is used to track the UTC Offset. Default is 0 as defined by DEFAULT_UTC_OFFSET in **constants.h**
- **UInteger16 epochNumber; -** Used by the stack internally to track number times the clock as been reset.
- **Octet ifaceName[IFACE_NAME_LENGTH];** - Optional field used to define the interface.
- **Boolean noResetClock;** - This field is not really used in the stack. Default is FALSE as defined by DEFAULT_NO_RESET_CLOCK in **constants.h**
- **Boolean noAdjust;** - This is used by initClock() to determine if the clock frequency needs to be adjusted. This is not used in the Windows implementation of the stack. Set to FALSE.
- **Boolean displayStats; -** Use by protocol to determine if a debug message should be generated to show stack statistics. Called every time a state change is made.
- **Boolean csvStats; -** Used to tweak operation of displayStats procedure.
- **Octet directAddress[NET_ADDRESS_LENGTH];** - This is used by the Linux variant to set the socket address. This is not used in the Windows implementation.
- **Integer16 ap; -** This is used internally in the operation of the stack. Default value is defined by DEFAULT_AP in **constants.h**
- **Integer16 ai; -** This is used internally in the operation of the stack. Default value is defined by DEFAULT_AI in **constants.h**
- **Integer16 s; -** This is used internally in the operation of the stack. Default value is defined by DEFAULT_DELAY_S in **constants.h**
- **TimeInternal inboundLatency; -** This is the inbound latency. The default is defined by DEFAULT_INBOUND_LATENCY in **constants.h**

- **TimeInternal outboundLatency; -** This is the outbound latency. The default is defined by DEFAULT_OUTBOUND_LATENCY in **constants.h**
- **Integer16 max_foreign_records; -** This defines the number of foreign masters. Default is defined by DEFUALT_MAX_FOREIGN_RECORDS in **constants.h**
- **Boolean slaveOnly; -** This is used to force the stack into slave only mode.
- **Boolean probe;** - This is used by the stack during startup to determine if it should probe for clocks. This is not utilized in the Windows implementation.
- **UInteger8 probe_management_key;** - Used in the probe process. Not used in the Windows implementation.
- **UInteger16 probe_record_key;** - Used in the probe process. Not used in the Windows implementation.
- **Boolean halfEpoch; -** Used internally, not initialized in the Windows implementation.
- **Octet destMACAddress[NET_ADDRESS_LENGTH];** - This is the default destination MAC address that is used to send PTP messages. Default is: 10:00:5E:00:01:81
- **Octet localMACAddress[NET_ADDRESS_LENGTH];** - This is the default source MAC address that is used to send PTP messages. Default is 08:00:17:00:00:01 All device need to have unique addresses.
- **Boolean udpChksumEnable;** - This is a flag that determines if the send routine will generate a checksum on the UDP portion of the packet.
- **Octet srcIPAddress[4];** - This is the IP address used to send messages. Each device must have a unique number.
- **void *eplPortHandle;** - This is the handle to the EPL port object that is used to by the stack to use the library to operate on the port.
- **OAI_DEV_HANDLE_STRUCT *oaiHandle;** - This is a pointer to the OAI_DEV_HANDLE_STRUCT for the port/device. It is usd to communicate with the library.
- **Boolean forceBMCFlag; -** This is a flag that can be used to avoid the BMC operation and force a clock to become the master.
- **Boolean useOneStepFlag; -** This is a flag that is used by the stack to configure and operate in one-step mode. Current the stack only supports a master operating in one step mode. Slaves should be configured to not be in one step mode.
- **Boolean useTempRateFlag; -** Flag used to enable/disable the use of temporary rate adjustments as part of the slave tuning algorithm. Temporary rate adjustment offers the most accurate time correction results. This must be enabled when the synchronized CLKOUT feature is enabled. Default is TRUE
- **NS_UINT tempRateLength;** - This is the amount of time in microseconds to apply a temporary rate adjustment to the slave hardware clock. The default is 10000. The max value is 536870.
- **Boolean limiterEnable;** - This enables the offset limiter. Useful with variable delay links. Default is TRUE
- **NS_UINT limiterThresh;** - Error offset threshold where limiting will be applied. Default is 150
- **NS_UINT limiterThreshMax; -** Error offset threshold where result will be discarded. Default is 250.
- **NS_UINT limiterGoodThresh;** - Offsets less than this will be considered good tuning results and will be counted. Default is 100.

- **NS_UINT limiterLimitMultiplier;** Percent of over limit offset to use in offset correction. Default is 25.
- **TimeInternal syncAdjustValue;** - This is the value added to all sync transmit timestamps. Contains seconds and nanoseconds. Can be positive or negative. Default is 0.
- **TimeInternal delayReqAdjustValue; -** This is the value added to all delay request receive timestamps. Contains seconds and nanoseconds. Can be positive or negative. Default is 0.
- int **numRateSamples; -** Number of offset rate correction measurements that will be averaged together to perform an actual rate adjustment to the PHY device. This is also the number of syncs between hardware clock rate adjustments. Minimum is 2 and maximum is 64. Default is 4.
- int **numRateAvgs;** - This is the number of average rate correction that will be averaged together from the next hardware clock rate adjustment. Minimum is 2 and maximum is 64. Default is 2.
- int **numOneWayAvgSamples;** - This is the number of instantaneous one way delay measurements that will be averaged together to form the average oen way delay value. Minimum is 2 and maximum is 64. Default is 8.
- **Boolean syncEthMode;** - This enables the use of synchronous ethernet mode. This mode provides the most accurate time synchronization. This mode of operation is only functional when the link speed is 100Mb.
- **Boolean phaseAlignClkoutFlag;** - This enabled phase aligning the CLKOUT signal with the master's CLKOUT signal. The slave must be configured to use temporary rate adjustments.
- **Boolean clkOutEnableFlag;** - This enables the PHY device's CLKOUT signal.
- **NS_UINT clkOutDivide;** - This sets the PTP Clock divide-by value. This is the divide-by value for the output clock. The output clock is divided from the internal 250Mhz clock. Valid range is from 2 to 255. giving a nominal output frequency range from 125Mhz down to 980.4kHz. Divide-by values of 0 and 1 are not valid and will stop the output clock.
- **Boolean clkOutSpeed;** - This enables faster PTP clock output I/O rise/fall time for the divide by N clock output pin.
- **Boolean clkOutSource;** - This defines the CLKOUT source. Defined as boolean in stack but is really an int typedef so multiple values are used. 0 is FCO – Link loss loses phase alignment, 1 is FCO at 100Mbps only, and 2 is PGM. The 250Mhz clock source may be selected from either the internal FCO or PGM. The FCO offers reduced jitter in the output clock, while the PGM offers a wider range of frequency correction in the output clock.
- **Boolean ppsEnableFlag;** - This enables the pulse per second (PPS) output on the configured GPIO.
- **NS_UINT ppsStartTime;** - This specifics the PPS start time in seconds.
- **Boolean ppsRiseOrFallFlag;** - This specifies whether the PPS signal is rising or falling edge.
- **NS_UINT ppsGpio;** -This specifies the GPIO that will be used for the PPS signal. Valid values are 1-12. Default is 1.
- **NS_UINT clkOutPeriod; -** This is calculated by PTPInitHardware to be clkOutDivide * 4.
- **Boolean haveLoopbackedSend;** - This is an internal flag used to process messages.
- **NS_UINT lastSendLength;** - This is an internal flag used to process messages.
- **Octet txBuff[2048];** - This is an internal flag used to process messages.
- **Octet rxBuff[2048];** - This is an internal flag used to process messages.

5.3.9 Callbacks

PTPThreadC() is designed to accept pointers to 2 functions. These pointers are designed to provide the application with visibility into the operation of the stack for debug and monitoring of state. The following callbacks are defined:

5.3.9.1 stdioCallback

This is a pointer to a function in the application that is accessible from the thread that calls this function. It should be compatible with the following prototype.

```
void printMsg( int  msgType, char *msgString )
// Display message from stack
// Input: msgType - Type of message provided
//          0 - Debug Message (verbose)
//          1 - Debug Message (normal)
//          2 - Nofication Message
//          3 - Error Message
//        msgString – The message string
```

The callback **msgType** parameter can be used filter messages as needed. Most messages currently in the library are type 1 for normal debug.

NOTE: this function should not perform too much work. If possible it should simply place the messages in a queue to be processed by a different thread otherwise significant delays can be introduced into the stack.

5.3.9.2 statusUpdateCallback

This is a pointer to a function that performs status update for the library. The library will call this function to provide operational information to the application.. It should be compatible with the following prototype.

```
void ptpStatusUpdate( NS_UINT8 stsType, void *stsData )
// Display message from stack
// Input: stsType - Type of status provided
//          STS_PSF_DATA (1) –PHY Status Frame Data
//          STS_OFFSET_DATA (2) – Offset data
//         stsData – pointer to actual data
```

The basic operation of this callback should be as follows:

```
switch( stsType ) {
case STS_PSF_DATA:
    {
    PHYMSG_MESSAGE_TYPE_ENUM msgType;
    PHYMSG_MESSAGE phyMsg;
    NS_UINT8 *nxtMsg;
        // Normally we'd call IsPhyStatusFrame() using a raw packet but
        // since we got here we already know that the data is the 1st
        // of potentially several PSFs so we just process it.
```

```
            nxtMsg = (NS_UINT8 *)stsData;
            while( nxtMsg ) {
                nxtMsg = GetNextPhyMessage( devPort, nxtMsg, &msgType, &phyMsg );
                if( !nxtMsg || !bPSF ) {
                    continue;
                }
                switch( msgType ) {
                case PHYMSG_STATUS_TX:
                case PHYMSG_STATUS_RX:
                case PHYMSG_STATUS_TRIGGER:
                case PHYMSG_STATUS_EVENT:
                case PHYMSG_STATUS_ERROR:
                case PHYMSG_STATUS_REG_READ:
                default:
                    // do something with the PSFs
                    break;
                }
            }  // while( nxtMsg )
        }
        break;
    case STS_OFFSET_DATA:
        {
        STS_OFFSET_DATA_STRUCT *stsODS = (STS_OFFSET_DATA_STRUCT *)stsData;
            if( bSTSUpdate ) {
                printf( "ptpStatusUpdate %d.%d  %d.%d  %d.%d  %d.%d\n",
                        stsODS->offset_from_master.seconds,
                        stsODS->offset_from_master.nanoseconds,
                        stsODS->master_to_slave_delay.seconds,
                        stsODS->master_to_slave_delay.nanoseconds,
                        stsODS->slave_to_master_delay.seconds,
                        stsODS->slave_to_master_delay.nanoseconds,
                        stsODS->oneWayAvg.seconds,
                        stsODS->oneWayAvg.nanoseconds );
            }
        }
        break;
    default:
        break;
    } // switch( stsType )
```

The operation of this function is designed to support functionality similar to what is available in the ALP Framework demo. If this functionality is used at all it is expected that both the library and the operation of the procedure will be modified to meet the desired needs of the system.

NOTE: this function should not perform too much work. If possible it should simply place the messages/data in a queue to be processed by the application in an application thread.

5.3.10 PTP Stack Operation

As noted above the library contains an implementation of a PTP stack. It was derived from an open source project found at: http://sourceforge.net/projects/ptpd/ The code has been tweaked to operate as part of the EPL binary.

The application provides initialization of the stack and can display messages/updates from the stack but doesn't have any significant runtime control over the operation of the stack. Once the stack is running it pretty much runs automatically. Tweaking and tuning of the stack is outside the scope of this document.

NOTE: The inclusion of this stack is for demonstration and evaluation purposes only. There is no guarantee that the functionality provided is complete or accurate. No claims are made as to the level of precision provided by this particular implementation.