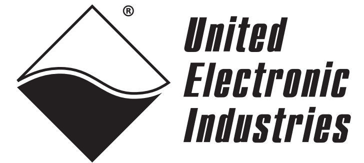


PowerDNx
1PPS / PTP Synchronization
Interface Manual

DNx 1PPS / PTP synchronization interface
for PPCx & -1G Cube and RACK series systems

May 2018

PN Man-DNx-Sync

© Copyright 1998-2018 United Electronic Industries, Inc. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form by any means, electronic, mechanical, by photocopying, recording, or otherwise without prior written permission.

Information furnished in this manual is believed to be accurate and reliable. However, no responsibility is assumed for its use, or for any infringement of patents or other rights of third parties that may result from its use.

All product names listed are trademarks or trade names of their respective companies.

See the UEI website for complete terms and conditions of sale:
http://www.ueidaq.com/cms/terms-and-conditions/

Contacting United Electronic Industries

Mailing Address:

27 Renmar Avenue
Walpole, MA 02081
U.S.A.

For a list of our distributors and partners in the US and around the world, please contact our support team:

Support:

Telephone: (508) 921-4600
Fax: (508) 668-2350

Also see the FAQs and online “Live Help” feature on our web site.

Internet Support:

Support: support@ueidaq.com
Website: www.ueidaq.com
FTP Site: ftp://ftp.ueidaq.com

Product Disclaimer:

WARNING!

DO NOT USE PRODUCTS SOLD BY UNITED ELECTRONIC INDUSTRIES, INC. AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS.

Products sold by United Electronic Industries, Inc. are not authorized for use as critical components in life support devices or systems. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness. Any attempt to purchase any United Electronic Industries, Inc. product for that purpose is null and void and United Electronic Industries Inc. accepts no liability whatsoever in contract, tort, or otherwise whether or not resulting from our or our employees' negligence or failure to detect an improper purchase.

Specifications in this document are subject to change without notice. Check with UEI for current status.

Table of Contents

Chapter 1 Introduction 1
1.1 Organization of this Manual 1
1.2 PPS / PTP Synchronization Overview 3
1.2.1 PTP Synchronization 3
1.2.2 PPS Synchronization 4
1.2.3 Determining Product Versions 5
1.3 Features 6
1.4 External Connections for Synchronization 7
1.4.1 Sync Connector Pinouts 8
1.5 Hardware System Configuration Examples 9
1.5.1 Synchronization Using an External 1PPS Signal 9
1.5.2 Synchronization Using IEEE-1588 PTP Standard 12
1.6 Internal Connections & Resources for Synchronization 15
1.6.1 Internal SYNC Bus 15
1.6.2 Adaptive Digital PLL 17
1.6.3 I/O Board Clock & Trigger Resources 18
1.7 I/O Board Clock & Trigger Configuration 20
Chapter 2 Programming the Synchronization Interface 21
2.1 About the Sync API 21
2.2 Sync Structure for Hardware Configuration 22
2.2.1 Sync Scheme Structure 22
2.2.2 Section A: IOM SYNC Source Configuration 23
2.2.3 Section B: Master Server Configuration 25
2.2.4 Section C: Clock Configuration 26
2.2.5 Section D: Trigger Configuration 28
2.2.6 Section E: SyncOut Configuration 31
2.3 Setting up the Sync Scheme 33
2.4 Setting up PTP Server Parameters 34
2.5 Programming I/O Board Clocks 36
2.6 Setting I/O Board Triggers 37
2.6.1 Arming Triggers 37
2.7 Setting I/O Board Timestamp Reference 38
2.7.1 Setting/Resetting Timestamps 38
2.8 Retrieving Status 39
2.9 Retrieving PTP Status 42
2.10 Retrieving UTC Time 45
2.11 Disabling Sync / Releasing Sync Hardware 46
Chapter 3 System Configuration Tutorials 47
3.1 Configuring Synchronization to an External PPS 48
3.1.1 Connecting Hardware for 1PPS Synchronization 48

© Copyright 2018
United Electronic Industries, Inc.

3.1.2 Configuring a UEI Chassis as 1PPS Master 49
3.1.3 Configuring a UEI Chassis as 1PPS Slave 53
3.1.4 Configuring Synchronized I/O Board Clocks 57
3.1.5 Configuring Synchronized Triggers & Timestamps 62

3.2 Configuring Hardware for PTP Synchronization 67
3.2.1 Configuring PTP Interface Parameters 68
3.2.2 Configuring a PTP Grandmaster 69
3.2.3 Configuring a Boundary Clock (IEEE-1588-capable Switch) 72
3.2.4 Configuring a UEI Chassis for PTP Synchronization 76

Chapter 4 Code Examples 84

4.1 About Sync Code Examples 84

4.2 Supported Data Acquisition Modes for Sync Interface 85

4.3 Example Code for Synchronization in RtVMap Mode. 85
4.3.1 Initialization (RtVMap) 86
4.3.2 Configuration (RtVMap) 89
4.3.3 Verify ADPLL Status (RtVMap) 90
4.3.4 Arm Trigger & Reset Timestamp (RtVMap). 91
4.3.5 Send / Receive Messages (RtVMap). 91
4.3.6 Stop Cleanly (RtVMap) 92

4.4 Example Code for Synchronization in ACB Mode 93
4.4.1 Initialization (ACB) 93
4.4.2 Configuration (ACB). 97
4.4.3 Verify ADPLL Status (ACB) 99
4.4.4 Reset Timestamp and Arm Trigger (ACB). 100
4.4.5 Send / Receive Messages (ACB) 100
4.4.6 Stop Cleanly (ACB) 101

Appendix A 102

A.1 DNA-CBL-SYNC-10/R3 Cable & Schematic. 102

A.2 DNA-CBL-SYNC-RJ-1G/R3 Cable Schematic 104

A.3 DNA-STP-SYNC-1G STP Panel. 106

List of Figures

Chapter 1 Introduction 1
1-1 Connectors on PPCx-1G Cube 7
1-2 Pinouts of Sync Connectors on UEI Chassis...8
1-3 Connection Diagram for 2-cube 1PPS Synchronization 9
1-4 Interconnection Diagram for Multi-chassis External 1PPS Synchronization................10
1-5 Block Diagram of DNA-STP-SYNC-1G 11
1-6 Example Configuration - PTP Master Clock / Boundary Clock / Slaves 12
1-7 Example Configuration - UEI Chassis as PTP Master...13
1-8 Example Configuration - Separate Operation and PTP Network................................14
1-9 Schematic of Sync Connections on Cube I/O Board 15
1-10 Block Diagram of Example SYNC Bus Connections 16
1-11 Diagram of Connecting to the Sync Interface Bus over Individual I/O Boards 20

Chapter 2 Programming the Synchronization Interface 21

Chapter 3 System Configuration Tutorials 47
3-1 Example Hardware Configuration for External 1PPS 48
3-2 Block Diagram of Master Configuration 51
3-3 Block Diagram of Slave Configuration 55
3-4 Block Diagram of Slave Configuration 60
3-5 Diagram of Connecting Clock from SYNC2 to AI-207 61
3-6 Block Diagram of Slave Configuration 63
3-7 Diagram of Connecting Trigger & Timestamp Reference from SYNC to AI-207 Board65
3-8 Example of PTP Hardware Configuration..67
3-9 Rear of the SecureSync™ PTP Grandmaster 69
3-10 Spectracom Grandmaster Dashboard 69
3-11 Spectracom Grandmaster PTP Config Screen..70
3-12 Spectracom PTP Grandmaster PTP Advanced Screen 71
3-13 UEI NIC1 Ports 72
3-14 Boundary Clock Dashboard...73
3-15 Boundary Clock PTP Settings Screen 74
3-16 Boundary Clock PTP Settings Screen 75
3-17 Block Diagram of PTP Configuration on UEI CPU Board..78
3-18 Code Snippet of Synchronization Structure Settings for PTP Sync.............................78

Chapter 4 Code Examples 84
A-1 Photo of DNA-CBL-SYNC-10/R3 Cable 102
A-2 Schematic of DNA-CBL-SYNC-10/R3 Cable..103
A-3 Photo of DNA-CBL-SYNC-RJ-1G Cable 104
A-4 Schematic of DNA-CBL-SYNC-10/R3 Cable..105

Chapter 1 Introduction

This manual provides documentation for synchronizing UEI Cube and RACK systems.

1PPS synchronization is supported on UEI systems that have CPU Logic 02.12.2D (2017) or later. (See Section 1.2.3 for checking logic versions.)

IEEE-1588 synchronization is supported on -02 and -03 versions of UEI Cube and RACK chassis with CPU Logic 02.12.46 (2018) or later.

NOTE: The software and API described in Chapters 2, 3, and 4 of this manual are for programming with the low-level C libraries; For users programming with the DAQLIB framework (C++, C#, LabVIEW, etc.), please refer to the *UeiDaq Framework User Manual*.

Chapter 1 contains the following sections:

- Organization of this Manual (Section 1.1)
- PPS / PTP Synchronization Overview (Section 1.2)
- Features (Section 1.3)
- External Connections for Synchronization (Section 1.4)
  - Sync Connector Pinouts (Section 1.4.1)
- Hardware System Configuration Examples (Section 1.5)
- Internal Connections & Resources for Synchronization (Section 1.6)
- I/O Board Clock & Trigger Configuration (Section 1.7)

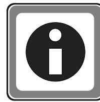
1.1 Organization of this Manual

This *1PPS / PTP Sync Interface User Manual* is organized as follows:

- **Introduction**
  Chapter 1 provides an overview of the synchronization interface features for various chassis and board models, device architecture, connectivity and logic.
- **Programming the SYNC Interface**
  Chapter 2 provides an overview of the low-level API functions that configure the synchronization interfaces for UEI cube and rack systems.
- **Tutorials**
  Chapter 3 provides step by step tutorials for setting up hardware and software for synchronization to an external 1PPS reference and synchronization using the IEEE-1588 / PTP standard.
- **Example Code**
  Chapter 4 provides example code for setting up synchronization in different data acquisition modes.
- **Appendix A - Accessories**
  This appendix provides descriptions of cable accessories and the SYNC STP board available for 1PPS / PTP Sync Interface.
- **Index**
  This is an alphabetical listing of the topics covered in this manual.

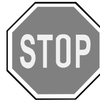
A glossary of terms used with the PowerDNA Cube/RACK and I/O boards can be viewed or downloaded from www.ueidaq.com.

Manual Conventions
To help you get the most out of this manual and our products, please note that we use the following conventions:



Tips are designed to highlight quick ways to get the job done or to reveal good ideas you might not discover on your own.

NOTE: Notes alert you to important information.



CAUTION! *Caution advises you of precautions to take to avoid injury, data loss, and damage to your boards or a system crash.*

Text formatted in **bold** typeface generally represents text that should be entered verbatim. For instance, it can represent a command, as in the following example: “You can instruct users how to run setup using a command such as **setup.exe**.”

Bold typeface will also represent field or button names, as in “Click **Scan Network**.”

Text formatted in `fixed` typeface generally represents source code or other text that should be entered verbatim into the source code, initialization, or other file.

Examples of Manual Conventions

Before plugging any I/O connector into the Cube or RACKtangle, be sure to remove power from all field wiring. Failure to do so may cause severe damage to the equipment.

Usage of Terms

Throughout this manual, the term “Cube” refers to either a PowerDNA Cube product or to a DNR- RACKtangle^{TM} rack mounted system, whichever is applicable. The term DNR- is a specific reference to the RACKtangle, DNA- to the PowerDNA I/O Cube, and DNx to refer to both.

© Copyright 2018
United Electronic Industries, Inc.

1.2 PPS / PTP Synchronization Overview

The UEI synchronization interface provides hardware and software resources to synchronize one or more chassis (Cubes or RACKs) to an external resource.

Systems can synchronize multiple, distributed chassis using either of the following:

- IEEE-1588 Precision Time Protocol (PTP) over IPv4/UDP Ethernet
- External pulse-per-second (PPS) reference signal via the 10-pin sync ports

1.2.1 PTP Synchronization

PTP synchronization is supported on -02 and -03 versions of UEI Cubes and RACKs having CPU Logic 02.12.46 (2018) or later.

The IEEE-1588 PTP standard defines the protocol for establishing a master/ slave relationship between a reference clock source (the PTP grandmaster) and all other devices in the system that will synchronize their clocks to the master clock (slaves).

The master/slave hierarchy is established through an exchange of PTP packets containing clock attributes (clock accuracy, etc.). PTP devices on the network process the clock attributes announced by a potential master using an algorithm of prioritized attributes called the Best Master Clock Algorithm (BMCA). If a grandmaster-capable device determines it has better clock attributes than those already announced, it announces, and this continues until the grandmaster for the system is determined.

The grandmaster is responsible for sending PTP Sync packets containing traffic timestamping data. Slave devices receive the PTP packets, determine network latency, and derive a local synchronized reference signal from timestamp data.

1.2.1.1 PTP Specification

UEI's implementation of the IEEE-1588 standard supports the following:

- PTP v2 (IEEE 1588-2008) supported on -02 and -03 versions of UEI GigE Cubes and RACKS (products with Freescale 8347/8347E CPU-types only)
- PTP over IPv4/UDP (Annex D)
- TBD resolution (packet timestamping)
- TBD accuracy (master to slave)
- Multicast transmission mode (unicast point-to-point is not currently supported)
- Hardware timestamping
- Grandmaster capability:
  - Hosted deployments of UEI systems are capable of being a PTP grandmaster or slave device
  - Standalone deployments (UEIPACs) can only be slave devices
- Clock Class: clockClass attribute is 248 (default clock class)
- Capable of connecting via an end-to-end boundary clock (transparent clocks not currently supported)
- Supported for I/O board synchronization only

Note that the chassis real-time clock cannot be synchronized to PTP time.

1.2.2 PPS Synchronization

Synchronization to an external pulse per second (PPS) reference is supported on UEI Cubes and RACKs having CPU Logic 02.12.2D (2017) or later.

UEI chassis can synchronize to an external pulse per second reference signal provided by a master source. Typically this is a one pulse per second (1PPS) signal. The 1PPS is routed to each chassis and input over the 10-pin sync port at the front of the Cube or RACK chassis.

A UEI chassis can also act as a master 1PPS source and generate and route its 1PPS reference signal for distribution to all slave chassis in the system

Once cubes and/or RACKs lock to the common 1PPS pulse, they can be programmed to generate internal clocks and triggers for all configured I/O boards in a system, allowing time alignment among each of the I/O boards on all of the synchronized chassis, as well as allowing the synchronization of timestamps.

1.2.2.1 PPS Specification

1PPS Input/Output:

- pulse-per-second (configurable)
- TTL signal levels
- TBD accuracy

NOTE: All Cubes and RACKs synchronizing to the same 1PPS pulse will lock with approximately a 100 ns accuracy range to each other.

Inter-connect Delays

Connection delays associated with PPS routing through cabling and screw terminal panels (STPs) are provided in Section 1.5.1.3 on page 11.

© Copyright 2018
United Electronic Industries, Inc.

1.2.3 Determining Product Versions

You can check the logic version installed on the CPU of your chassis to verify which synchronization methods your system supports.

You can use PowerDNA Explorer, a GUI application for communicating with a Cube or RACK chassis, or you can use the serial port.

Getting a hardware report using PowerDNA Explorer is the preferred method.

1.2.3.1 Checking CPU Logic using PowerDNA Explorer

To determine the logic version using PowerDNA Explorer, do the following:

1. Open PowerDNA Explorer:
   On Windows systems, you can access PowerDNA Explorer from the Windows Start menu:
   Start > *All Programs* > *UEI* > *PowerDNA* > *PowerDNA Explorer*

   On Linux systems, you can access PowerDNA Explorer under the UEI installation directory (`<PowerDNA-x.y.z>/explorer`) and type:
   `java -jar PowerDNAExplorer.jar`
2. In the Menu bar of PowerDNA Explorer, click
   View > *Show Hardware Report*.
3. Note the **Logic** version under the **Layer: CPU** section.

1.2.3.2 Checking CPU Logic using Serial Connection

To determine the logic version using a serial connection, do the following:

1. Install a serial cable between your host PC and the UEI chassis.
2. Open a serial communication application, (e.g., PuTTY, MTTTY, minicom) on your host PC, and configure settings to 57600 bits/s, 8 data bits, 1 stop bit and no parity, and then connect.
3. Once connected, type `devtbl logic <Return>` at the command prompt. The CPU board is designated as DevN “14”, and the logic version is listed under the **Logic** column.
4. To check the CPU version, type `devtbl <Return>`. The CPU board is listed as “cpu” under the **Phy/Virt** column, and the version number is listed under the **Option** column.

NOTE: For a UEIPAC-based chassis, you can also access the command prompt using `telnet` or `ssh` by connecting to the chassis CPU over the Ethernet port. Once connected, use the commands listed in steps 3 and 4 to check versioning.

1.3 Features

The following is an overview of the synchronization interface features.

- Synchronization mode options:
  - 1PPS synchronization
  - IEEE 1588 PTP synchronization
  - Support for using DNx-IRIG-650 board as a source for 1PPS & GPS synchronization
  - NTP synchronization in later versions of the hardware
- Synchronization signal routing:
  - Systems synchronizing to an external 1PPS signal receive and transmit the 1PPS signal via a 10-pin sync connector at the front of the chassis
  - Systems synchronizing using the IEEE-1588 PTP standard can receive packets via the NIC1 or NIC2 Ethernet port
- Master / Slave configuration:
  - Chassis can be configured as PPS master (generate and route the 1PPS) or slave (receive an external 1PPS)
  - Hosted deployments synchronizing to the IEEE PTP standard can be a PTP slave or grandmaster
  - Standalone deployments (UEIPACs) synchronizing to the IEEE PTP standard can only be a PTP slave
- Clock options:
  - Clocks can be synchronized to the external reference and distributed to I/O boards
  - If an application requires it, I/O boards can be run from other clock sources that are not synchronized to the IEEE PTP standard or to an external 1PPS reference
- Trigger options:
  - Synchronization requires all boards to start together
  - The start trigger can be an external hardware trigger, a software trigger, or generated relative to a PPS edge
  - The stop trigger can be programmed to stop after a time or number of scans

1.4 External Connections for Synchronization

For PTP synchronization, PTP hardware timestamping is supported on either Ethernet port. Using NIC1 (Ethernet port 0) or NIC2 (Ethernet port 1) is user-configurable.

For synchronization to an external 1PPS signal, the 1PPS routes into or out of a UEI chassis through a 10-pin sync connector at the front of the chassis. Note that for systems synchronizing more than 2 chassis to an external 1PPS signal, UEI offers a screw terminal panel (STP), the DNA-STP-SYNC-1G.

Refer to **Figure 1-1** for locations of the sync port and Ethernet ports.

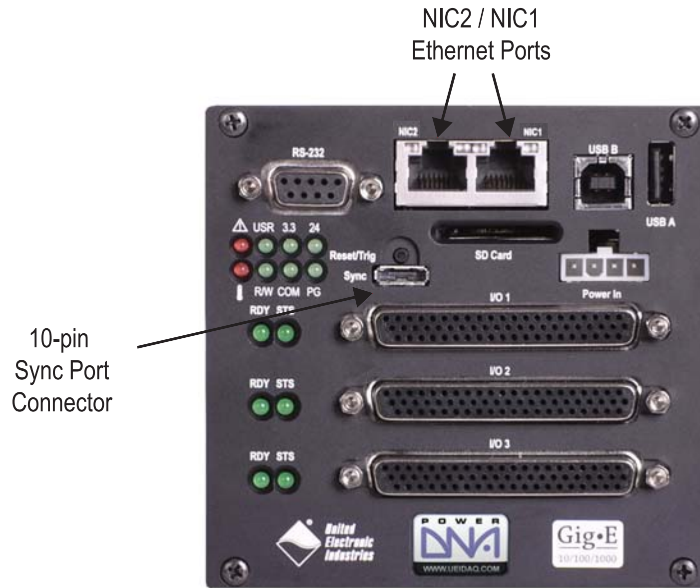


Figure 1-1 Connectors on PPCx-1G Cube

On a PowerDNA PPCx Cube (5200 CPU: i.e., DNA-PPC5/8/9 or UEIPAC-300/600), the 10-pin sync connector provides one sync input (Sync In 0) and one sync output (Sync Out 0) for routing an external 1PPS signal. PTP synchronization is not supported on these product versions.

On a PowerDNA PPCx-1G Cube and DNR/F RACK chassis (8347 CPUs), the 10-pin sync connector provides two sync inputs and two sync outputs. Users have a choice of which sync pin on the connector to route an external 1PPS signal. This is configurable in software. (For PTP synchronization, the sync port is not used and all synchronization occurs over the Ethernet ports).

1.4.1 Sync Connector Pinouts

Pinouts of the sync connector for the PPCx-1G Cube / RACK and the PPCx Cube (i.e., DNA-PPC5/8/9 or UEIPAC-300/600) are shown below in **Figure 1-2**.

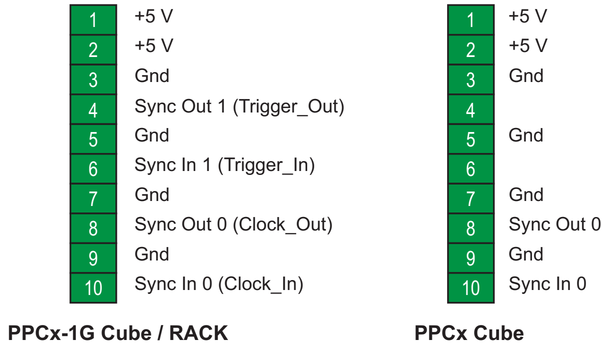


Figure 1-2 Pinouts of Sync Connectors on UEI Chassis

NOTE: For information regarding wiring delays associated with cat5e or better cables and screw terminal panels (STPs) for synchronization, please see "Connection Delays when Using Multiple Chassis" on page 11.

1.5 Hardware System Configuration Examples

The following sections provide examples of connecting hardware for synchronizing with an external 1PPS source or using PTP.

Example tutorials of configuring UEI chassis and other components to support these configurations are provided in **Chapter 3**.

1.5.1 Synchronization Using an External 1PPS Signal

To synchronize a multi-chassis system, one cube (or RACK) can be designated as the 1PPS master and the others as slaves. Alternatively, all UEI chassis can be configured as slaves to non-UEI 1PPS-generating hardware.

1.5.1.1 Example of Syncing to an External 1PPS in a 2-Chassis System

An example of synchronizing a 2-chassis system using one chassis as the 1PPS master and the second as its slave is shown in **Figure 1-3**.

Figure 1-3 Connection Diagram for 2-cube 1PPS Synchronization

For more information about UEI synchronization cables, refer to Appendix A.

NOTE: The absolute maximum signal frequency supported over the sync connector is 5 MHz.

© Copyright 2018
United Electronic Industries, Inc.

1.5.1.2 Example of Syncing to an External 1PPS in a Multi-chassis System

For synchronizing multiple chassis to an external 1PPS master, UEI offers a screw terminal panel (STP) that can serve up to 6 slave ports on the STP.

As an example configuration, **Figure 1-4** shows a UEI chassis acting as the 1PPS master; however, note that any 1PPS source could be used.

Additional slaves can be added to the system by daisy-chaining STP boards together. Refer to the UEI DNA-STP-SYNC-1G Synchronization Interconnection Panel documentation for more information.

NOTE: Connection delays through cabling and STP are described in Section 1.5.1.3 on page 11.

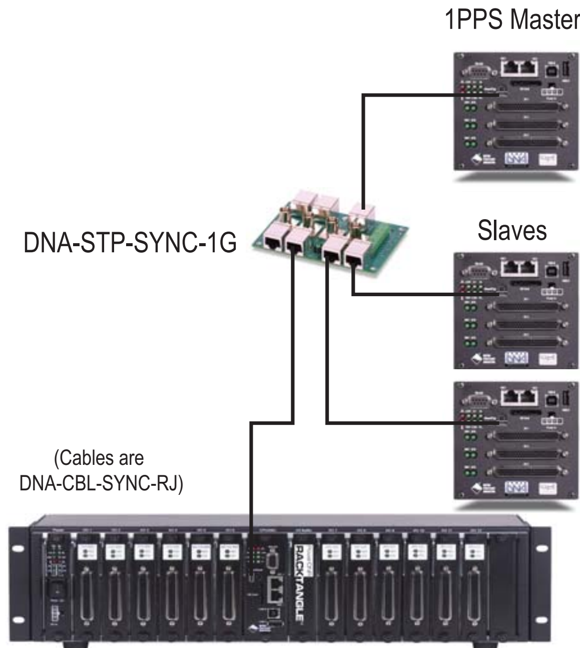


Figure 1-4 Interconnection Diagram for Multi-chassis External 1PPS Synchronization

Figure 1-5 provides a block diagram of the DNA-STP-SYNC-1G panel. For more information about UEI synchronization cables, refer to Appendix A.

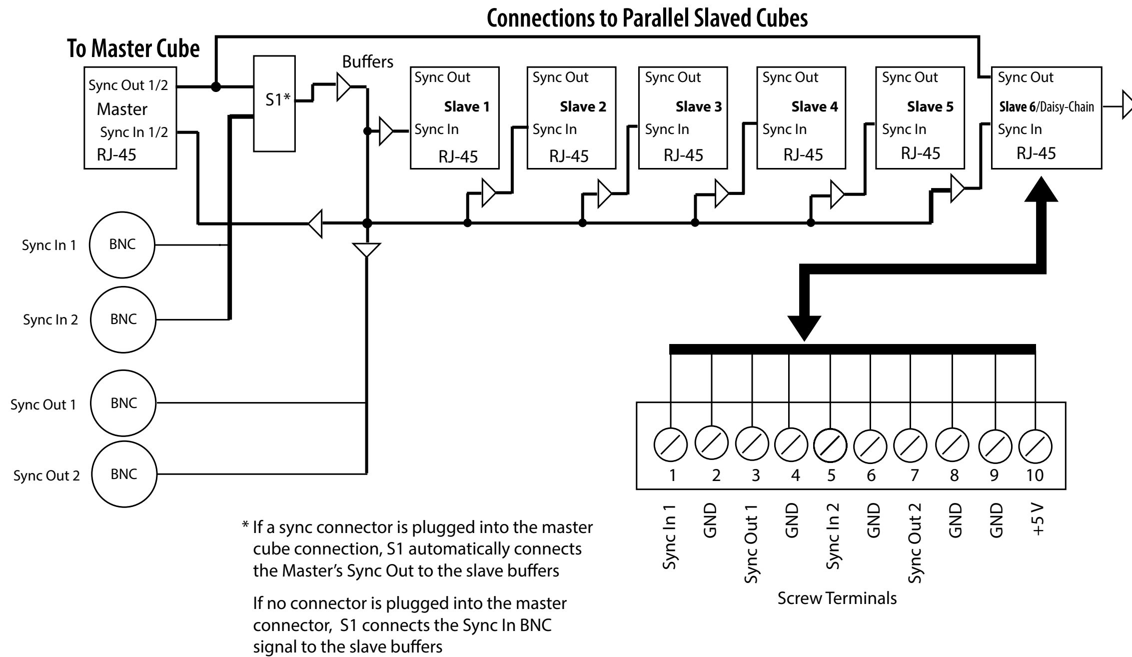


Figure 1-5 Block Diagram of DNA-STP-SYNC-1G

1.5.1.3 Connection Delays when Using Multiple Chassis

Connecting multiple chassis using cat5e or better cables are subject to the following delays:

- Delay through 1/2 foot of cat5e cable is approximately 1 ns
- Delay through DNA-STP-SYNC-1G (**Figure 1-5**) is approximately 100 ns

NOTE: Cable and STP panel delays were tested at UEI using up to 800 foot cables (1.6 µs).

1.5.2 Synchronization Using IEEE-1588 PTP Standard

For optimal synchronization among UEI slave chassis, the Grandmaster clock and IEEE-1588 capable Ethernet switch (configured as an end-to-end boundary clock) must support hardware timestamping.

Examples in this section use the following network hardware for synchronizing a system using the IEEE-1588 PTP standard:

UEI slave chassis	-02 and -03 versions of UEI's GigE RACKtangle and Cube
IEEE 1588 PTP grandmaster (master clock source)	Example PTP masters include: Spectracom's SecureSync™ PTP grandmaster or -02 and -03 versions of UEI's GigE RACKtangle and Cube
IEEE 1588-capable Industrial switch	Examples include managed Ethernet switches with PTP support **Note that UEI's current IEEE-1588 implementation requires configuring your IEEE 1588-capable switch as an end-to-end boundary clock**
Host PC	Used to configure a PTP grandmaster and boundary clock and to run the user application in PowerDNA (hosted) deployments (non UEIPAC deployments)

A **boundary clock** is an Ethernet switch that additionally manages IEEE-1588 packets. It redistributes PTP packets from the grandmaster to an isolated subnet of slaves. A boundary clock acts as a slave to the grandmaster and then becomes the master clock to any slave devices on the subnet. All clocks ultimately derive their time from the grandmaster clock.

1.5.2.1 Example Configuration Using Same NIC for Ethernet and PTP Packets

Figure 1-6 shows the following example configuration:

- UEI Cubes & RACKs using PTP synchronization in hosted deployment
- Host PC running user application
- IEEE 1588 PTP master clock source providing synchronization packets
- (1) IEEE 1588 boundary clock routing PTP packets between the PTP master and UEI slave chassis (NIC1) and application & data packets between host PC & UEI chassis (NIC1)

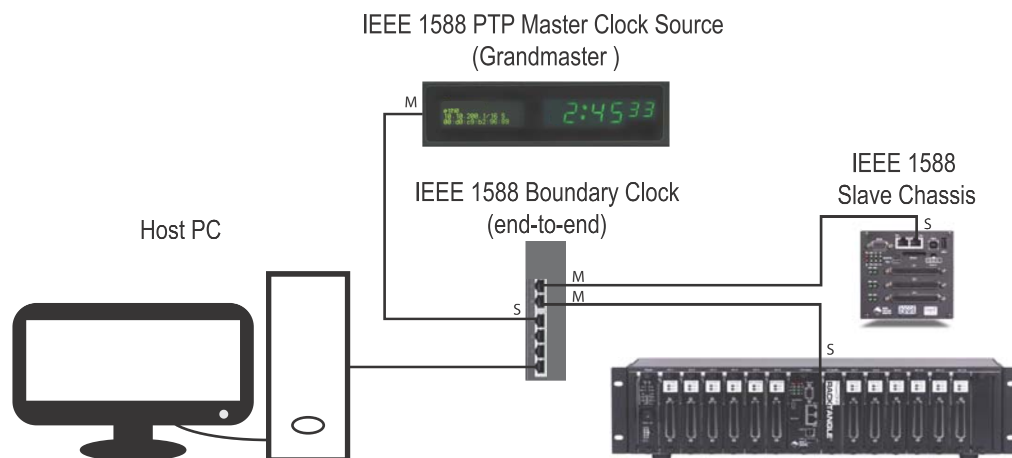


Figure 1-6 Example Configuration - PTP Master Clock / Boundary Clock / Slaves

1.5.2.2 Example Configuration Using UEI Chassis as PTP Master

A UEI Cube or RACK is capable of serving as the PTP master of your system.

Figure 1-7 shows the following configuration:

- UEI Cubes & RACKs using PTP synchronization in hosted deployment
- Host PC running user application
- UEI Cube acting as 1588 PTP master clock source providing synchronization packets
- (1) IEEE 1588 boundary clock routing PTP packets and application & data packets between UEI chassis (NIC1) & host PC

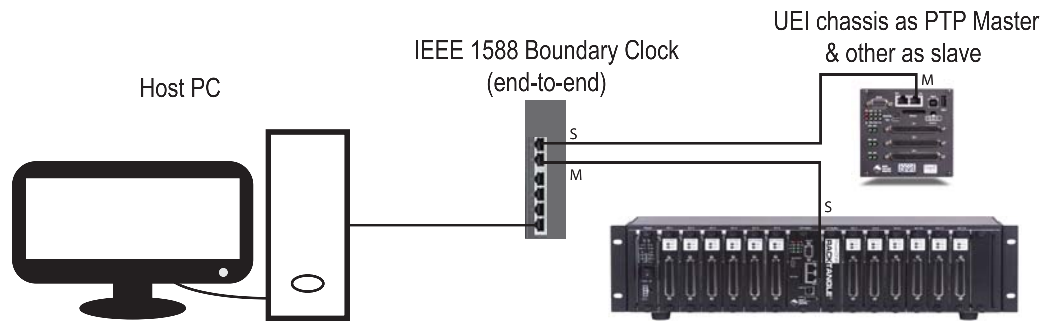


Figure 1-7 Example Configuration - UEI Chassis as PTP Master

Which Cube or RACK is PTP master is determined by the best master clock algorithm, (BMCA). UEI also supports a debug mode, where the selection of the PTP master can be forced using a setting in the configuration structure.

1.5.2.3 Example Configuration Using Separate NICs for Operation and PTP Packets

As an alternative, you can configure two separate networks: one for operation and one for PTP synchronization.

Figure 1-8 shows the this configuration:

- UEI Cubes & RACKs using PTP synchronization in hosted deployment
- Host PC running user application
- IEEE 1588 PTP Grandmaster clock source providing synchronization
- (1) IEEE 1588 boundary clock routing PTP packets between PTP master and UEI slave chassis (NIC2)
- (1) 1G network switch routing application & data packets between host PC & UEI chassis (NIC1)

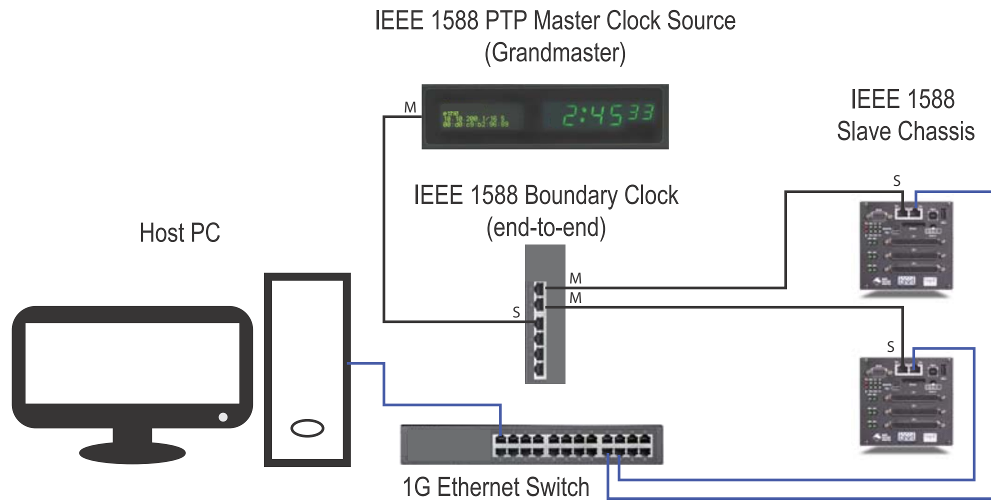


Figure 1-8 Example Configuration - Separate Operation and PTP Network

1.6 Internal Connections & Resources for Synchronization

UEI chassis use an internal interboard bus that routes synchronization signals between the CPU board and I/O boards via four internal SYNC lines. SYNC lines are designated SYNC0 through SYNC3. The schematic below shows how the SYNC lines route through the main connector of a DNA Cube I/O board.

- On a Cube chassis, interboard signals are bused through stacked connectors. **Figure 1-9** shows SYNC lines routed to connector labeled JMAIN1.
- On a RACK chassis, interboard bus signals route across the chassis backplane and connect through backplane connectors to each board installed in the RACK.

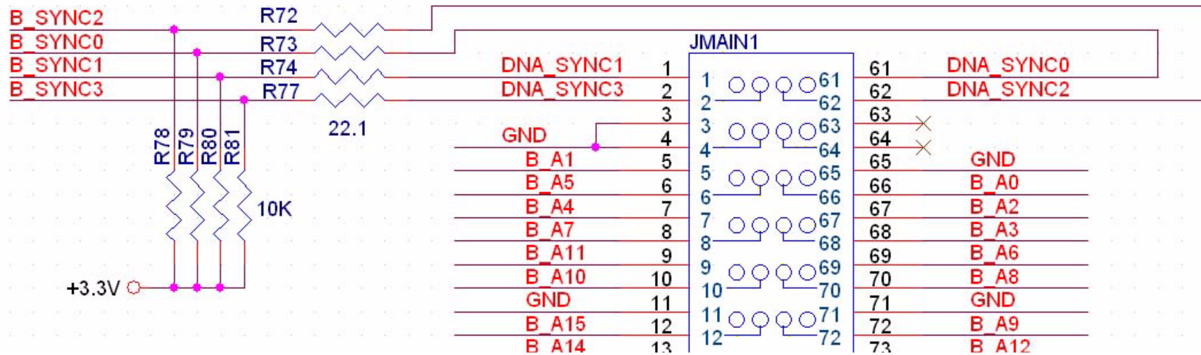


Figure 1-9 Schematic of Sync Connections on Cube I/O Board

As shown above, each SYNC line is pulled up with a 10 kΩ resistor to prevent synchronization lines from bouncing and also ensure that proper drive is available from every board. In the PPC8 Cube, for example, the total resistance is 1430 Ω with a termination current of 2.3 mA.

1.6.1 Internal SYNC Bus

The SYNC lines can be accessed from the chassis CPU board and are accessible to each I/O board in the chassis. The driving source for each of the four SYNC lines is user programmable.

The following are sync sources that can be routed to or from the CPU board over the sync bus:

- External sources routed through the 10-pin sync connector
- A phase-locked loop (PLL) clock generator local to the CPU board
- An adaptive digital phase-locked loop (ADPLL) that validates and follows an external, raw 1PPS or PTP-derived 1PPS reference
- An Event Module that synthesizes and/or divides a CPU generated clock based on the locked 1PPS reference from the ADPLL
- Trigger circuitry

SYNC lines are assigned to hardware resources as defined by configuration settings in a global DQ_SYNC_SCHEME structure. See **Chapter 2** for more information about programming the sync interface.

Hardware Checks

UEI firmware checks that SYNC line routing is only set once. For example, if the user programs a clock on SYNC line 1 and later programs a trigger on SYNC line 1, the firmware will produce a warning and only connect the last assignment to prevent hardware damage.

© Copyright 2018
United Electronic Industries, Inc.

1.6.1.1 Internal SYNC Bus Routing

Each I/O board in a chassis can be programmed to use a clock or trigger resource from the CPU board routed via any of the SYNC lines. Determining which resource is connected to which internal sync line is user configurable.

UEI's synchronization hardware and software allow for highly flexible and customizable system designs. For standardization, we use the following sync line mapping, which you will find in most of our example code:

- SYNC0: raw 1PPS pulse routed to ADPLL for internal use
- SYNC1: Clock source (CPU-generated clock or divided clock that is routed to I/O boards)
- SYNC2: Clock source (usually the Event Module clock locked to the 1PPS pulse and routed to I/O boards)
- SYNC3: CPU generated trigger synchronized to the 1PPS pulse and routed to I/O boards

NOTE: When synchronizing to an external 1PPS reference, one SYNC line is needed to route the raw 1PPS reference to the adaptive digital phase-locked loop (ADPLL) on the CPU board. The other three SYNC lines can be connected to clock and trigger resources as your application requires. Refer to **Figure 1-10**.

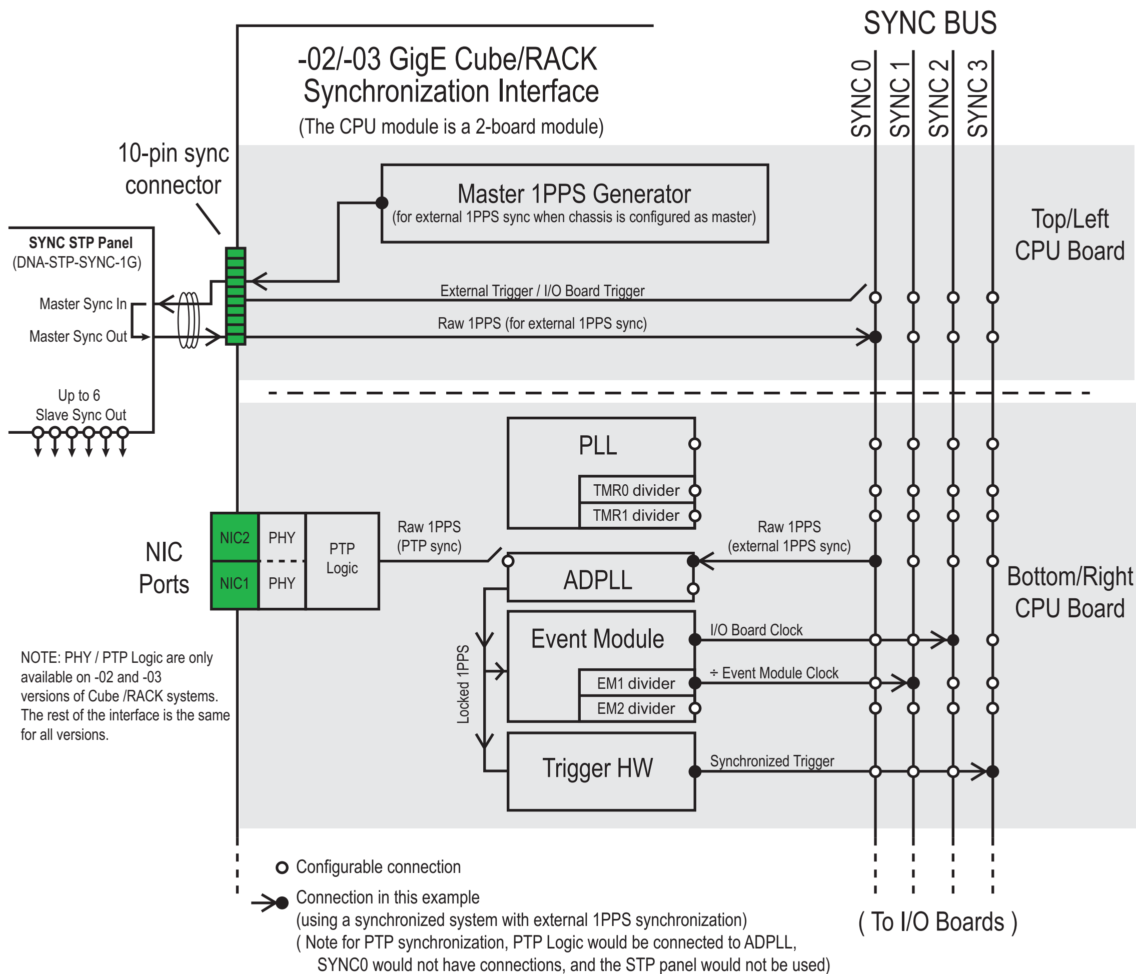


Figure 1-10 Block Diagram of Example SYNC Bus Connections

1.6.2 Adaptive Digital PLL

The adaptive digital phased-lock-loop (ADPLL) synchronizes to a raw PPS reference signal:

- When synchronizing to an external 1PPS input, a raw 1PPS signal is delivered to the ADPLL via the 10-pin sync connector
- When synchronizing using the PTP standard, a raw 1PPS signal is derived from the PTP timestamps.

The ADPLL produces an internal reference from the raw 1PPS for generating clocks, triggers, and timestamps for all I/O boards installed in the chassis.

The ADPLL is responsible for the following functions:

- validates the raw 1PPS by comparing the incoming signal to a user defined tolerance value
- follows an external/raw 1PPS reference signal and once synchronized continues to produce a 1PPS pulse for internal use in absence of a valid 1PPS raw signal
- provides minimum, maximum, and averaged period values to the software, along with other status data that can be read via an API function call for system monitoring and troubleshooting.

Refer to Section 2.8 on page 39 for more information about API for retrieving status when programming with the low-level C libraries. API for retrieving status when programming with the DAQLIB framework (C++, C#, LabVIEW, etc.) are found in the *UeiDaq Framework User Manual*.

1.6.3 I/O Board Clock & Trigger Resources

Users can program which clock or trigger source each I/O board installed in a cube or RACK will use.

To synchronize I/O board clocks in a multi-chassis system, use the Event Module and/or the Event Module dividers as the clock source for your I/O boards that you want synchronized. The Event Module is synchronized to the PPS/PTP reference source.

- Internal Clock Sources are described below in **Table 1-1**.
- Trigger sources are described on page 19.

Refer to **Chapter 2** for more information about how to program these options.

1.6.3.1 Clock Sources

Table 1-1 provides descriptions of available clock sources:

Table 1-1 Clock Sources for PowerDNx Boards

Clock Source	Description
Event Module (1PPS/PTP synchronized)	The Event Module is located on the CPU board and is used to provide synchronized clock sources to I/O boards at user-programmed frequencies. The Event Module requires CPU board logic greater than 12.2D (refer to Section 1.2.3 to learn how to determine CPU board logic). The Event Module receives the locked 1PPS reference from the ADPLL and performs additional stabilization by tracking the number of clocks per PPS and adjusting the frequency to stay within ±1 pulse/clock cycle. This produces a highly-stablized, synchronized clock for use by I/O boards at the rate programmed by the user. Synchronized clock rates are generated to the accuracy of 1 Hz to the programmed rate. The maximum clock rate supported is 1 MHz. **Note**: Each I/O board can further divide the clock routed via the SYNC line. EM0 should be programmed with a rate divisible by the clock rates required by each of the I/O boards.
Event Module Dividers: EM1 and EM2 (1PPS/PTP synchronized)	The main clock produced by the Event Module (EM0) can be routed to either of two 8-bit Event Module clock dividers, EM1 or EM2. Using EM1 or EM2, you can divide the EM0 clock by a maximum of 255. This means that three different clocks can be generated by the Event Module: EM0, EM1, and/or EM2. To use the divider clocks, the Event Module is programmed and routed to a SYNC line, and then either or both EMx dividers can be programmed and their divided clocks routed to additional SYNC lines.
PLL	A phase-locked loop (PLL) onboard the CPU board can be routed throughout the chassis and used as an alternate clock source. The PLL is not synchronized to the PTP/1PPS reference.

© Copyright 2018
United Electronic Industries, Inc.

Table 1-1 Clock Sources for PowerDNx Boards (Continued)

Clock Source	Description
PLL Dividers: TMR0 and TMR1	The PLL can be divided down by either of two 32-bit timers (TMR0 and TMR1) and routed to SYNC lines for use by I/O boards. To use the dividers, the PLL is programmed and routed to a SYNC line, and either or both dividers can also be programmed and routed. Clocks generated by dividing the PLL are not synchronized to the PTP/1PPS reference.
External Clocks	A clock generated external to the chassis can be routed in through the 10-pin sync connector and made accessible to chassis I/O boards via one of the SYNC lines. An externally generated clock will not be locked to the internal ADPLL or to the PTP/1PPS reference.
None of the above	By default, the clock source for I/O boards is generated on the board, divided down from the 66 MHz system clock. Additionally, several boards provide their own onboard PLL: the AI-211, AI-217, AI-218, and AI-228 boards all have this capability. Neither of these options will be synchronized to the PTP/1PPS reference.

1.6.3.2 Trigger Sources

UEI systems can trigger from the ADPLL 1PPS signal, an external signal, or software.

The start trigger aligns all boards and starts acquisition. For triggers referenced to the ADPLL 1PPS, the stop trigger can be programmed using a user-defined number of clock cycles of a user-specified clock reference routed to a SYNC line or a user-defined time in milliseconds.

1.7 I/O Board Clock & Trigger Configuration

PowerDNx I/O boards are configured by defining a clock source, timestamp source, and trigger source. Each I/O board has the capability of dividing down an input clock that is routed from a SYNC bus line.

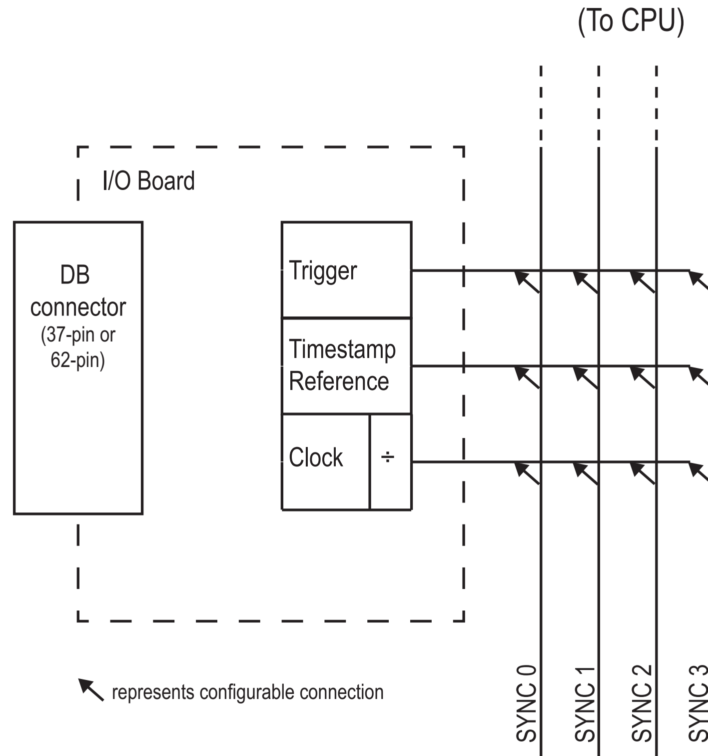


Figure 1-11 Diagram of Connecting to the Sync Interface Bus over Individual I/O Boards

Refer to Section 2.3 on page 33 for API to setup I/O board synchronization options and Section 3.1.4 and 3.1.5 starting on page 57 for a tutorials.

Additionally, IRIG-650 boards can provide a variety of clock and trigger options, which can get routed to the SYNC bus. Please refer to the DNx IRIG-650 data sheet and user's manual for information, and the *PowerDNA API Reference Manual* for API descriptions.

© Copyright 2018
United Electronic Industries, Inc.

Chapter 2 Programming the Synchronization Interface

This chapter outlines API and software structures for programming 1PPS / PTP synchronization when programming with the low-level C libraries.

NOTE: Refer to the *UeiDaq Framework User Manual* for information about synchronizing systems using the DAQLIB framework (C++, C#, LabVIEW, etc.).

The following sections are included in this chapter:

- About the Sync API (Section 2.1)
- Sync Structure for Hardware Configuration (Section 2.2)
- Setting up the Sync Scheme (Section 2.3)
- Setting up PTP Server Parameters (Section 2.4)
- Programming I/O Board Clocks (Section 2.5)
- Setting I/O Board Triggers (Section 2.6)
- Setting I/O Board Timestamp Reference (Section 2.7)
- Retrieving Status (Section 2.8)
- Retrieving PTP Status (Section 2.9)
- Retrieving UTC Time (Section 2.10)
- Disabling Sync / Releasing Sync Hardware (Section 2.11)

1PPS synchronization is supported after CPU Logic 02.12.2D (2017). PTP synchronization is supported in -02 and -03 versions of the GigE Cube and RACK chassis only.

Refer to Section 1.2.3 for instructions on how to check logic versions.

2.1 About the Sync API

The first step in configuring your UEI system for synchronization is to define your system and hardware configuration.

Configure the following for each UEI chassis requiring synchronization:

- For PTP synchronization, you first configure PTP server parameters using the `DqSyncDefinePTP()` API.
- Hardware configuration for CPU board and chassis is defined in a software structure of type `DQ_SYNC_SCHEME`, which is set in hardware using the `DqSyncDefineSyncScheme()` API.
- After that, I/O board clocks, triggers, timestamps and other synchronization options can be set up.

NOTE: The PowerDNA API Reference Manual provides detailed descriptions of the `DQ_SYNC_SCHEME` structure and synchronization API, and **Chapter 3** provides tutorials.

2.2 Sync Structure for Hardware Configuration

Define signal routing and other synchronization options for each chassis in your system using the sync structure (`DQ_SYNC_SCHEME`).

The full structure is shown below. Element options are described in detail in the following sections, Section 2.2.2 thru Section 2.2.6.

2.2.1 Sync Scheme Structure

The Sync Scheme structure is set up in hardware using the `DqSyncDefineSyncScheme()` API.

```
typedef struct {
    // ==== section A =====================================================
    // IOM Sync Source Configuration
    uint32 sync_device;       // IOM CPU type (5200,8347,or 8347 with PTP capability)
    uint32 sync_source;       // where to get nPPS clock to synchronize system
    uint32 sync_line;         // which SYNC line to route external 1PPS clock
    uint32 sync_mode;         // mode of synchronization
    uint32 nPPS;              // N - number of pulses per second for input nPPS clock
    uint32 nPPS_us;           // Expected accuracy of the nPPS clock in µs, clocks
                              //   outside of the range will be ignored, 0=default

    // ==== section B =====================================================
    // synchronization output: tell IOM to become 1PPS master
    uint32 sync_server;       // Identify chassis as 1PPS master
    uint32 srv_param;         // which external sync connector pin routes 1PPS out
                              //  to the 1PPS slave chassis in the system
    uint32 trig_server;       // <Reserved>

    // ==== section C =====================================================
    // Clock configuration: select clock source for each SYNC line (0 thru 3)
    uint32 clock_src[DQL_SYNC_LINES];     // clock source for each SYNC line
    uint32 clock_tmr[DQL_SYNC_LINES];     // PLL and external clock can be
                                          //   divided on TMR0 or TMR1
    uint32 clock_frq[DQL_SYNC_LINES];     // clock frequency (for PLL/EM)
    uint32 clock_div[DQL_SYNC_LINES];     // clock divider for EMx
                                          //   (0 == divide by 1, 2, 3 etc.)

    // ==== section D =====================================================
    // Trigger Configuration: tell IOM where to get (or generate)/route trigger signal
    uint32 trig_source;        // where to take trigger to start acquisition
    uint32 trig_line;          // <reserved>
    uint32 trig_start;         // start trigger mode selection
    uint32 trig_delay;         // offset of the trigger pulse from nPPS clock\
                               //    (microseconds)
    uint32 trig_period_ms;     // period in ms to issue start trigger
    uint32 trig_stop;          // source for the stop trigger
    uint32 trig_stop_src;      // stop source for stop trigger upon N-count
    uint32 trig_duration;      // milliseconds before issuing stop trigger or N-count

    // ==== section E =====================================================
    // destination to route signals: from SYNC lines or to the outside SyncOut0/1
    uint32 clclk_dest[DQL_SYNC_LINES];  // where to feed CL clock
    uint32 pps_dest;                    // where to feed 1PPS clock to
    uint32 trig_dest;                   // where to feed start/stop trigger

} DQ_SYNC_SCHEME, *pDQ_SYNC_SCHEME;
```

2.2.2 Section A: IOM SYNC Source Configuration

Section A parameters describe synchronization source set up information.

All IOMs that are PPS / PTP synchronized, whether they are the master 1PPS server or a slave chassis, need to initialize parameters described below in **Table 2-1**.

Element Name	Description & Options
`sync_device`	Identifies what type of IOM this structure is used to program. Set `sync_device` to any of the following: • `DQ_SYNC_8347`: standard 8347 CPU (PPCx-1G Cube or RACK) • `DQ_SYNC_5200`: standard 5200 CPU (PPCx Cube) • `DQ_SYNC_8347S`: 8347 CPU with 1588 support hardware (-02 or -03 Cube or RACK) • `DQ_SYNC_5200S`: <Reserved>
`sync_source`	Sets external input pin on the 10-pin connector that receives nPPS pulse. Set `sync_source` to any of the following: • `0` for PTP synchronization • `DQ_SYNCCLK_SYNCIN0`: Uses SyncIn0 (5200 or 8347) (pin 10) • `DQ_SYNCCLK_SYNCIN1`: Uses SyncIn1 (8347 only) (pin 6) See Figure 1-2 on page 8 for pin descriptions.
`sync_line`	Sets internal bus SYNC line that routes the nPPS signal to the ADPLL. Set `sync_line` to any of the following: • `0` for PTP synchronization • `DQ_SYNCCLK_SYNC0`: Sync0 line delivers nPPS clock • `DQ_SYNCCLK_SYNC1`: Sync1 line delivers nPPS clock • `DQ_SYNCCLK_SYNC2`: Sync2 line delivers nPPS clock • `DQ_SYNCCLK_SYNC3`: Sync3 line delivers nPPS clock
`sync_mode`	Defines how (or which protocol is used) to synchronize the chassis. • `DQ_SYNCCLK_SYNC:` synchronized with external PPS pulse • `DQ_SYNCCLK_NTP:` <Reserved> • `DQ_SYNCCLK_1588:` synchronized with IEEE-1588 PTP protocol. Logically OR with `DQ_SYNCCLK_ETH0` or `DQ_SYNCCLK_ETH1` to configure NIC port to use (e.g., `DQ_SYNCCLK_1588 | DQ_SYNCCLK_ETH0`). Eth0 (NIC1) is default. 1588 (PTP) can only be used when `sync_device` is `DQ_SYNC_8347S` and with -02 or -03 Cube or RACK chassis
`nPPS`	Determines the number of pulses per second in the nPPS reference. • Set `nPPS` to the number of pulses / second. (1 is typical – it will define a 1PPS signal)

Table 2-1 Descriptions for Section A: IOM SYNC Source Configuration

Element Name	Description & Options
nPPS_us	Sets the expected accuracy of the nPPS source. The ADPLL uses this to validate the incoming nPPS pulse and ignores pulses outside this range. The ADPLL will maintain an internal nPPS signal while the reference nPPS input is out of the nPPS_us range. Set nPPS_us to the accuracy of the device generating the sync pulse. For example, an nPPS pulse produced by a chassis acting as the 1PPS master can stay within 100 µs accuracy; therefore, nPPS_us should be programmed with a value of 100. As a general rule, program this number to the jitter value of your clock source + 100 µs. 0 will use the default value for the mode of operation.

Table 2-1 Descriptions for Section A: IOM SYNC Source Configuration

2.2.3 Section B: Master Server Configuration

Section B parameters provide options to set the chassis as the 1PPS server (Master server). When the following options are set, the chassis is responsible for generating a 1PPS synchronization signal internally and routing it externally to synchronize all chassis configured in the system.

Master server parameters are described in **Table 2-2**.

Element Name	Description & Options
`sync_server`	Configures the chassis to generate a synchronization signal (1PPS) which can be routed externally to synchronize all chassis in the system. A chassis configured as a sync server will generate the 1PPS pulse. Other protocols are currently <reserved>. Set `sync_server` to either of the following: • `DQ_SYNCSRV_1PPS` to configure the chassis as the 1PPS master • 0 to leave chassis as 1PPS slave or for PTP synchronization **NOTE** that the `srv_param` element below contains additional information about how to route the signal.
`srv_param`	Controls where the internally generated 1PPS signal will be routed when `sync_server` is set to `DQ_SYNCSRV_1PPS`. Set `srv_param` to any of the following: • `DQ_SYNCSRV_SYNCOUT0`: Routes to SyncOut0 (5200 or 8347) (pin 8) • `DQ_SYNCSRV_SYNCOUT1`: Routes to SyncOut1 (8347 only) (pin 4) • `0`: set to 0 if chassis is a 1PPS slave or for PTP synchronization Additionally, `srv_param` can be logically ORed with any of the following to route the 1PPS onto internal SYNC bus: • `DQ_USE_SYNC0`: Raw 1PPS to SYNC0 • `DQ_USE_SYNC1`: Raw 1PPS to SYNC1 • `DQ_USE_SYNC2`: Raw 1PPS to SYNC2 • `DQ_USE_SYNC3`: Raw 1PPS to SYNC3 For example, the following routes the 1PPS externally to SyncOut0 on the 10-pin sync connector and internally to SYNC line 0. `DQ_SYNCSRV_SYNCOUT0 | DQ_USE_SYNC0`
`trig_server`	Reserved, set to 0.

Table 2-2 Descriptions for Section B: Master Configuration

2.2.4 Section C: Clock Configuration

Section C parameters provide options for configuring clocks, (e.g., clock routing over SYNC lines and setting clock frequency).

Each of the parameters in this section is an array of four with one element mapped to each SYNC line. Array element 0 corresponds to SYNC line 0 while array element 3 corresponds to SYNC line 3.

Clock parameters are described below in **Table 2-3**.

NOTE: Typically the SYNC lines are also used to carry an nPPS signal and a trigger. The array elements corresponding to these lines should be left as ‘0’.

Element Name	Description & Options
`clock_src[4]`	Routes a particular clock source to the corresponding SYNC line. The SYNC line is determined by the element position of the array {<SyncLine0>, <SyncLine1>, <SyncLine2>, <SyncLine3>} or alternatively set by bitwise ORing a `DQ_USE_SYNCx` value with the value set. The following represent clock sources: • `DQ_CLOCKSRC_UNUSED`: Not connected to any of the following clock sources or connected elsewhere (or left unused) • <`DQ_CLOCKSRC_ADPLL`: locked 1PPS from ADPLL, for debug only> • `DQ_CLOCKSRC_EM0`: Clock synthesized by Event Module • `DQ_CLOCKSRC_EM1`: Event Module divider 1 clock (use `clock_div` as divider) • `DQ_CLOCKSRC_EM2`: Event Module divider 2 clock (use `clock_div` as divider) • `DQ_CLOCKSRC_SYNCIN0`: Clock input from SYNCIN0 pin (5200 or 8347 CPU) • `DQ_CLOCKSRC_SYNCIN1`: Clock input from SYNCIN1 pin (8347 CPU board) • `DQ_CLOCKSRC_PLL0`: Phase-lock loop (PLL0) on CPU board • `DQ_CLOCKSRC_PLL0TMR0`: PLL0 on CPU board divider 1 (reg0) • `DQ_CLOCKSRC_PLL0TMR1`: PLL0 on CPU board divider 2 (reg1) NOTE: Only the ADPLL and Event Module are synchronized to the PPS / PTP source (SYNCIN0/1, PLL0, and PLL0TMR0/1 are not locked with the PPS signal). Lines that have previously been set or will be set to something else should be written with 0, (i.e., `DQ_CLOCKSRC_UNUSED`). As an example, following programs the Event Module to generate a clock routed to the SYNC2 internal bus line, and all the other SYNC lines are not used for clocks: * Set `clock_src` to {0, 0, `DQ_CLOCKSRC_EM0`, 0} Or equivalently, * Set `clock_src` to (`DQ_CLOCKSRC_EM0` | `DQ_USE_SYNC2`)

Table 2-3 Descriptions for Section C: Clock Configuration

Element Name	Description & Options
clock_tmr[4]	<Reserved>
clock_freq[4]	Sets the frequency of the clock specified for each SYNC line. For example, if the Event Module needs to produce an 8 kHz clock that is routed to SYNC line 2, and all other SYNC lines do not map to a clock, clock_src would be set to {0, 0, DQ_CLOCKSRC_EM0, 0} and clock_freq would be set to {0, 0, 8000, 0}
clock_div[4]	• Used as an 8-bit Event Module clock divider when clock_src is set as DQ_CLOCKSRC_EMx. • Used as a 32-bit PLL clock divider when clock_src is set as DQ_CLOCKSRC_PLL0TMRx. Set clock_div to 0 or 1 to divide by 1 (i.e., leave clock_freq as programmed), set to 2 to divide by 2, set to 3 to divide by 3, etc. For example, clock_div set to {0, 0, 0, 0} will not divide any of the programmed rates and will not use divider hardware.

Table 2-3 Descriptions for Section C: Clock Configuration

2.2.5 Section D: Trigger Configuration

Section D parameters provide options for configuring the start and stop trigger. Trigger parameters are described below in **Table 2-4**.

Element Name	Description & Options
`trig_source`	Routes an externally generated or internally generated trigger to the corresponding SYNC line for distribution to I/O boards in chassis. Set `trig_source` to any of the following: • `DQ_USE_SYNC0`: Feed trigger from SYNC0 • `DQ_USE_SYNC1`: Feed trigger from SYNC1 • `DQ_USE_SYNC2`: Feed trigger from SYNC2 • `DQ_USE_SYNC3`: Feed trigger from SYNC3 If using an externally generated trigger, (i.e., `trig_start` = `DQ_TRIGSTART_SYNC`), `trig_source` must take an additional parameter to identify which external pin the trigger is routed in on. If using an externally generated trigger, either of the following parameters must be bitwise ORed with one of the `DQ_USE_SYNCx` parameters: • `DQ_TRIGSTART_SYNCIN0`: Delivers trigger via SyncIn0 (5200 or 8347) (pin 10) • `DQ_TRIGSTART_SYNCIN1`: Delivers trigger via SyncIn1 (8347 only) (pin 6) See Figure 1-2 on page 8 for pin descriptions.
`trig_line`	Reserved, set to 0.

Table 2-4 Descriptions for Section D: Trigger Configuration

Element Name	Description & Options
trig_start	Describes the mode of operation of the start trigger. The trigger can be programmed as an external signal routed in through the sync connector or an internally generated synchronized trigger. Set trig_start to either of the following: • DQ_TRIGSTART_SYNC: Derive start trigger from the SYNC line • DQ_TRIGSTART_NPPS: Issue start trigger on the next PPS (plus <trig_delay>) NOTE: • When using **DQ_TRIGSTART_SYNC**, the chassis routes an external trigger as described by trig_source. This means the I/O boards will start on the rising edge of the input signal and stop on the falling edge. This trigger will not be locked locally with the 1PPS reference. • When using **DQ_TRIGSTART_NPPS,** the user must issue a command to “arm” the trigger on all the boards. To arm all chassis at once, the broadcast API, DqSyncTrigOnNextPPSBrCast(), can be used; to arm a single chassis, DqSyncTrigOnNextPPS() can be used. Once armed, the chassis will trigger on the next nPPS signal received. The stop (trig_stop) can be programmed as a time span (trig_duration) or as a number of clock cycles (trig_stop_source and trig_duration).
trig_delay	Used to issue a trigger after a time delay from the nPPS signal (in µs) when trig_start is configured as DQ_TRIGSTART_NPPS. The minimum value of 0 causes the start trigger to be issued as soon as the nPPS signal is received. The maximum value is 1048575 (0xFFFFF), or approximately 1 second of delay.
trig_period_ms	Reserved.

Table 2-4 Descriptions for Section D: Trigger Configuration

Element Name	Description & Options
trig_stop	Defines the stop trigger for I/O boards to stop acquiring data. • DQ_TRIGSTOP_SYNC: Derive stop trigger from the SYNC line • DQ_TRIGSTOP_DURATION: Issue stop trigger after a user-programmed delay (in ms) • DQ_TRIGSTOP_NCLOCKS: Issue stop trigger after a user-programmed number of clock cycles of a user-specified clock source on a SYNC line • 0: no stop trigger NOTE: • DQ_TRIGSTOP_SYNC is used when the start trigger is external • DQ_TRIGSTOP_DURATION causes the chassis to issue a stop trigger after a programmed number of milliseconds (see trig_duration description below) • DQ_TRIGSTOP_NCLOCKS causes the chassis to issue the stop trigger once a certain number of user-defined clock cycles have passed (see trig_stop_src and trig_duration parameters below)
trig_stop_src	Used when trig_stop=DQ_TRIGSTOP_NCLOCKS. This parameter specifies which SYNC line provides the clock source used to determine the number of clocks (clock sources are defined and routed in Section C: Clock Configuration). Set trig_stop_source to any of the following: • DQ_USE_SYNC0: Feed clock from SYNC0 • DQ_USE_SYNC1: Feed clock from SYNC1 • DQ_USE_SYNC2: Feed clock from SYNC2 • DQ_USE_SYNC3: Feed clock from SYNC3
trig_duration	Sets when the stop trigger will be asserted. Used when trig_stop is DQ_TRIGSTOP_DURATION or DQ_TRIGSTOP_NCLOCKS. • When trig_stop is set as DQ_TRIGSTOP_DURATION, trig_duration is the time in milliseconds the stop trigger will be issued after the start trigger is detected. The maximum value is 1048575 (0xFFFFF), or approximately 17 minutes of acquisition. • When trig_stop is set as DQ_TRIGSTOP_NCLOCKS, trig_duration defines a number of clock cycles that will pass until the stop trigger is issued. The maximum value is 1048575 (0xFFFFF), or DQ_TRIGSTOP_NCLOCKS * (1/frequency) of the trig_stop_source.

Table 2-4 Descriptions for Section D: Trigger Configuration

2.2.6 Section E: SyncOut Configuration

Section E parameters describe where to route signals to the SYNC lines or to the external SyncOut pin(s).

SyncOut parameters are described below in **Table 2-5**.

Element Name	Description & Options
`clclk_dest[4]`	Routes clock source from a particular SYNC line out through sync connector. Each element takes a SYNC line bitwise ORed with the desired pin of the sync connector: Set `clclk_dest` to any of the following sync lines: • `DQ_USE_SYNC0`: Feed clock from SYNC0 • `DQ_USE_SYNC1`: Feed clock from SYNC1 • `DQ_USE_SYNC2`: Feed clock from SYNC2 • `DQ_USE_SYNC3`: Feed clock from SYNC3 and bitwise OR `DQ_USE_SYNCx` with either of the following to route clock out externally through the sync connector: • `DQ_CLKDEST_SYNCOUT0`: Delivers clock via SyncOut0 (5200 or 8347) (pin 8) • `DQ_CLKDEST_SYNCOUT1`: Delivers clock via SyncOut1 (8347 only) (pin 4) See Figure 1-2 on page 8 for pin descriptions. To route SYNC line 2 out ClkOut pin, set `clclk_dest` to: • `{0,0, DQ_USE_SYNC2 | DQ_CLKDEST_SYNCOUT0, 0}` NOTE: ORing `DQ_USE_SYNCx` with the `SYNCOUT` variable in the array is a different format than the other arrayed elements in the `DQ_SYNC_SCHEME` structure.
`pps_dest`	Route the PPS signal out sync connector. The SYNC line routing the PPS signal is bitwise ORed with the desired pin of the sync connector. Set `pps_dest` to any of the following sync lines: • `DQ_USE_SYNC0`: Feed clock from SYNC0 • `DQ_USE_SYNC1`: Feed clock from SYNC1 • `DQ_USE_SYNC2`: Feed clock from SYNC2 • `DQ_USE_SYNC3`: Feed clock from SYNC3 and bitwise OR `DQ_USE_SYNCx` with either of the following to route the nPPS out externally through the sync connector: • `DQ_nPPSDEST_SYNCOUT0`: Delivers nPPS via SyncOut0 (5200 or 8347) (pin 8) • `DQ_nPPSDEST_SYNCOUT1`: Delivers nPPS via SyncOut1 (8347 only) (pin 4) See Figure 1-2 on page 8 for pin descriptions.

Table 2-5 Descriptions for Section E: Sync Out Configuration

Element Name	Description & Options
trig_dest	Route the trigger out sync connector. The SYNC line routing the PPS signal is bitwise ORed with the desired pin of the sync connector. Set pps_dest to any of the following sync lines: • DQ_USE_SYNC0: Feed clock from SYNC0 • DQ_USE_SYNC1: Feed clock from SYNC1 • DQ_USE_SYNC2: Feed clock from SYNC2 • DQ_USE_SYNC3: Feed clock from SYNC3 Bitwise OR DQ_USE_SYNCx with either of the following: • DQ_TRGDEST_SYNCOUT0: Delivers trigger via SyncOut0 (5200 or 8347) (pin 8) • DQ_TRGDEST_SYNCOUT1: Delivers trigger via SyncOut1 (8347 only) (pin 4) See Figure 1-2 on page 8 for pin descriptions.

Table 2-5 Descriptions for Section E: Sync Out Configuration

2.3 Setting up the Sync Scheme

The sync scheme structure is set with the DqSyncDefineSyncScheme() API. The function sets up the sync scheme options in hardware.

Table 2-6 API for Hardware Settings for Sync Interface

API Name	Description
DqSyncDefineSyncScheme	Sets up hardware settings for the sync interface. Parameters: • int handle: Handle to the IOM • pDQ_SYNC_SCHEME scheme: See Section 2.2 • uint32* status: Returned status. See NOTE below

NOTE: A returned status value of 0 means the sync setup passed. A non-zero value points to the element in the DQ_SYNC_SCHEME structure that caused the failure.

DqSyncDefineSyncScheme() should be called after the following setup:

- after chassis I/O boards are initialized with board-specific configuration APIs (e.g., channel list selection, gain selection, etc.)
- after boards are put into the Operation State, (e.g., after DqeEnable() is called when acquiring data in ACB mode or when DqRtVMapStart*() is called when acquiring data in VMAP mode.)
- after PTP server parameters are configured if synchronizing using the IEEE-1588 standard (after DqSyncDefinePTPServer is called).

Once DqSyncDefineSyncScheme() is called, I/O board clocks, triggers, and timestamps can be configured, and acquisition can start, triggered by a broadcast message or an external trigger. API for configuring I/O board clocks, triggers, and timestamps are described below.

2.4 Setting up PTP Server Parameters

If using PTP synchronization, call `DqSyncDefinePTPServer()` API to set up the PTP parameters. If `DqSyncDefinePTPServer()` is not called, default parameters stored in FLASH will be used.

NOTE: PTP synchronization is supported on CPU Logic version 12.46 and later and on -02 and -03 GigE Cube and RACK product versions.

Table 2-7 API for Setting PTP Server Parameters

API Name	Description
`DqSyncDefinePTPServer`	Defines settings for PTP protocol handling. Parameters: • `int handle`: Handle to the IOM • `int mode`: <Reserved, set to 0> • `pDQ_SYNC_DEFPTP pPTP`: Structure of PTP parameters, see description below

```
typedef struct {
    uint8 subdomain;         // PTP subdomain (Domain) number
    uint8 priority1;         // Priority 1 of the device
    uint8 priority2;         // Priority 2 of the device
    int8  logSyncInterval;   // log2(period of sync messages)
    int8  logMinDelayRequestInterval; // log2(minimum space
                                      //   between delay requests)
    int8  logAnnouceInterval; // log2(period of announce msgs)
    uint8 annouceTimeout;     // number of announce messages
                              //   before timing out
    // nonstandard extensions
   uint32 cfg;                  // For debug; mask to bypass BMCA
   uint32 static_master_ip;     // assigned master IP address when
                                //     bypassing BMCA
} DQ_SYNC_DEFPTP, *pDQ_SYNC_DEFPTP;
```

PTP Parameter	Description
`uint8 subdomain`	Subdomain field that specifies the set of clocks in a multiple clock distribution system that are capable of synchronizing with each other. Default is 0.
`uint8 priority1`	User-assigned, preemptive priority to the best master clock algorithm (Smaller numbers indicate higher priority. This is automatically set to 255 when a chassis is configured in slave only mode; 128 otherwise)
`uint8 priority2`	User-assigned priority2 (Smaller numbers indicate higher priority. This is automatically set to 255 when a chassis is configured in slave only mode; 128 otherwise)

Table 2-8 PTP Parameters

PTP Parameter	Description
`int8 logAnnouceInterval`	Log2(period of announce messages) How often the PTP master clock sends Announce messages. For example, when `logAnnouceInterval=4,` the time between log messages will be 2^4 or 16 seconds.
`uint8 annouceTimeout`	Number of announce intervals allowed to transpire without the slave receiving an Announce message from the master. After this delay, the system will timeout. "ANNOUNCE_RECEIPT_TIMEOUT_EXPIRES" will occur: at which point, a device capable of being a master will try to become master, and a slave-only device returns to the state of listening for a new master.
`int8 logSyncInterval`	Log2(period of sync messages) How often the PTP master clock sends Sync messages in multicast mode. For example, when `logSyncInterval=0,` the time between log messages will be 2^0 or 1 second.
`int8 logMinDelayRequestInterval`	log2(minimum space between delay requests) Minimum interval allowed between PTP delay-request messages.Slave devices extract this from sync packets. (If chassis is master clock, this value must be set). For example, when `logMinDelayRequestInterval=1,` the time between log messages will be 2^1 or 2 seconds
`int32 cfg`	Bitmask for PTP configuration: • Set to 0 for normal operation • Set to `DQ_PTP_USE_STATIC_MASTER` to bypass BMCA and select the PTP master directly with `static_master_ip`. Default is 0.
`uint32 static_master_ip`	IP address of master IEEE-1588 device (will be set and reset automatically using the best-master clock algorithm) Can be configured for debug.

Table 2-8 PTP Parameters

`DqSyncDefinePTPServer()` should be called according to the following:

- call after chassis I/O boards are initialized with board-specific configuration APIs (e.g., channel list selection, gain selection, etc.)
- call after boards are put into the Operation State, (e.g., after `DqeEnable()` is called when acquiring data in ACB mode or when `DqRtVMapStart*()` is called when acquiring data in VMAP mode.)
- call before the API to set up synchronization hardware is called (call `DqSyncDefinePTPServer` before `DqSyncDefineSyncScheme`).

2.5 Programming I/O Board Clocks

Program I/O board clocks using the `DqSyncDefineLayerClock()` API. This API must be called for each I/O board that requires synchronization.

Note the clock source supplied to an I/O board via the SYNC line can be further divided down locally on each I/O board.

Table 2-9 API for Programming Clock Source

API Name	Description
`DqSyncDefineLayerClock`	Sets clock source for I/O boards. This function must be called after `DqSyncDefineSyncScheme()`. Parameters: • `int handle`: Handle to the IOM • `int devn`: Board position in IOM (zero-based index) • `pDQ_SYNC_DEF_CLOCKS clocks`: See Note below

NOTE: The `DQ_SYNC_DEF_CLOCKS` defines board-specific clock parameters, which sets up the SYNC line providing the clock source, a local divider for boards that require a divided down clock rate, group delay and mode parameters:

```
typedef struct {
    int clk_line;  // SYNC line to use for clock source
    int divider;   // 0,1 = original clock,
                   // 2 thru n = divide clock by (n)
    int grp_delay; // group delay in samples,
                   //       -1 = auto define
    uint32 flags;  // flags to change mode of operation
} DQ_SYNC_DEF_CLOCKS, *pDQ_SYNC_DEF_CLOCKS;
```

The `grp_delay` represents the group delay of the finite impulse response (FIR) filters available on several Analog Input boards: AI-205, AI-211, AI-217, AI-218, and AI-228.

Setting this parameter to -1 allows the software to program the default group delay associated with the particular I/O board you are programming. Allowing the software to compensate for the group delay will guarantee timestamp alignment. The software holds off incrementing the timestamp until the first filtered input sample is ready for output. Default group delays for each I/O board are provided in the *PowerDNA API Reference Manual*.

© Copyright 2018
United Electronic Industries, Inc.

2.6 Setting I/O Board Triggers

Program I/O board triggers using the `DqSyncDefineLayerTrigger()` API. This API must be called for each I/O board that requires synchronized triggers.

Table 2-10 API for Programming Trigger Source

API Name	Description
`DqSyncDefineLayerTrigger`	Sets trigger source for I/O boards. This function must be called after `DqSyncDefineSyncScheme()`. Parameters: • `int handle`: Handle to the IOM • `int devn`: Board position in IOM (zero-based index) • `int trig_line`: SYNC line or resource providing the trigger source • `int mode`: Reserved, set to 0

2.6.1 Arming Triggers

When a UEI system is configured to generate a trigger on the next PPS transition (see “Section D: Trigger Configuration” on page 28), arm I/O board triggers using either of the following API.

Table 2-11 API for Arming Triggers

API Name	Description
`DqSyncTrigOnNextPPSBrCast`	Sends a broadcast UDP command to trigger on the next PPS that is received by IOMs identified in the `handle_arr` array. Parameters: • `int handle`: Handle to the IOM • `int nIOM`: Number of IOMs to broadcast to • `int reserved`: Reserved, set to 0 • `int* handle_arr`: List of handles of IOMs to broadcast to
`DqSyncTrigOnNextPPS`	Triggers a single chassis* on the next PPS. Parameters: • `int handle`: Handle to the IOM • `uint32 reserved`: Reserved, set to 0 (*If the trigger generated from this API is routed externally through the 10-pin sync connector to all slave chassis, this API can externally synchronize all slave chassis in the system. Note that using this method to route the trigger externally will introduce delays associated with cable lengths, which will not be case when using the multiple chassis broadcast API, `DqSyncTrigOnNextPPSBrCast()`).

2.7 Setting I/O Board Timestamp Reference

Program the clock reference for generating I/O board timestamps with the `DqSyncDefineLayerTimestamp()` API.

The timestamp reference clock is used to index the timestamp every reference clock transition. This is usually the same reference as the I/O board clock.

Table 2-12 API for Programming Timestamp Reference Source

API Name	Description
`DqSyncDefineLayerTimestamp`	Sets the timestamp clock source for I/O boards. This function must be called after `DqSyncDefineSyncScheme()`. Parameters: • `int handle`: Handle to the IOM • `int devn`: Board position in IOM (zero-based index) • `int trig_line`: SYNC line or resource providing timestamp source

2.7.1 Setting/ Resetting Timestamps

I/O board timestamps are synchronized, set, and/or reset simultaneously with the following API.

Table 2-13 API for Setting or Resetting Timestamps

API Name	Description
`DqCmdResetTimestampBrCast`	Sends a broadcast UDP command to set/reset timestamp to timestamp value. Received by all IOMs. Parameters: • `int handle`: Handle to the IOM • `uint32 timestamp`: Value to all IOM timestamps to

2.8 Retrieving Status

Retrieve synchronization status with the `DqSyncGetSyncStatus()` API.

Table 2-14 API for Retrieving Status

API Name	Description
`DqSyncGetSyncStatus`	Retrieves status registers associated with the synchronization hardware. Parameters: • `int handle`: Handle to the IOM • `int mode`: Reserved, set to 0 • `pDQ_SYNC_STATUS status`: See Note below

NOTE: The DQ_SYNC_STATUS structure consists of the following elements:

```
typedef struct {
    DQ_SYNC_ADPLL_STAT adpll_sts;// ADPLL status structure
                                 //  see Table 2-16
    uint32 pll_stat[0];   // <reserved>
    uint32 pps_status;    // <reserved>
    uint32 gps_irig;      // <reserved>
    uint32 evm_stat;      // Event module status register
    uint32 sync_snap;     // snapshot of the SYNC lines
    uint32 sync_conn;     // snapshot of external sync
                          //   connector lines
    uint32 time_since_pps;   // how long since last PPS
} DQ_SYNC_STATUS, *pDQ_SYNC_STATUS;
```

The DQ_SYNC_ADPLL_STAT structure (first element in `DQ_SYNC_STATUS`) provides the status settings specific to the ADPLL:

```
typedef struct {
    uint32 status; // ADPLL status register
    uint32 min_per; // Minimum period length for input clock
    uint32 avg_per; // Averaged detected length of VALIDATED
                    //    input period
    uint32 max_per;// Maximum period length for input clock
    uint32 lst_per;// Measured length of last input period
    uint32 acc_err;// Accumulated pulse position error in
                   //    system clocks
} DQ_SYNC_ADPLL_STAT, *p DQ_SYNC_ADPLL_STAT;
```

NOTE: **Table 2-15** and **Table 2-16** provide bit descriptions of the status registers.

Sync Status Register Name	Description
`status`	**Table 2-16** provides descriptions of the status information returned with the `DQ_SYNC_ADPLL_STAT` structure.
`evm_stat`	Status of Event Module: 28 EVTMOD_STS_DPLL: Reads (1) if the Event Module generated the correct number of clocks 23:0 Event counter. Specific count values are for reserved use; however, note that a non-incrementing counter can indicate an error in your 1PPS source or sync structure configuration.
`sync_snap`	Snapshot of SYNC lines on the internal SYNC bus: 31:24 <Reserved> 23 Current state of internal SYNC3 line 22 Current state of internal SYNC2 line 21 Current state of internal SYNC1 line 20 Current state of internal SYNC0 line 19:0 <Reserved>
`sync_conn`	Snapshot of the external SyncIn and SyncOut lines on the external 10-pin connector: 31:0
`time_since_pps`	Number of 66 MHz clocks since last 1PPS 31:0 Number of cycles

Table 2-15 Bit Mapping of Sync Status Registers

© Copyright 2018
United Electronic Industries, Inc.

.

ADPLL Status Register Name	Description
status	ADPLL status register: 31:17 <Reserved> 16 RESYNC: Reads (1) when ADPLL and external clock are re-synchronized. This bit will be 1 after initial synchronization of the ADPLL and clock source. If it is set during normal ADPLL operation, a 1 indicates that the synchronization with the source clock was lost (and an error grew to > ¼ of the external clock period). Reading a 1 on RESYNC may indicate a drifting or unstable clock source. RESYNC is a sticky bit, auto-cleared. Reset state is 0. 15:3 <Reserved> 2 AV: Reads (1) when the moving average passes validation. Invalidated input clocks will not affect the moving average. Reset state is 0. 1 CV: Reads (1) when the last input clock period passed validation. Reset state is 0. 0 CE: Reads (1) when the last input clock period is too long. Reset state is 0. This bit will be a 1 if the 1PPS pulse is lost.
min_per	Minimum period length for the ADPLL input clock: 31:27 <Reserved> 26:0 Minimum period
avg_per	Measured average period of the ADPLL detected and validated input clock: 31:27 <Reserved> 26:0 Measure average period
max_per	Maximum period length for the ADPLL input clock: 31:27 <Reserved> 26:0 Maximum period
lst_per	Measured length of the last ADPLL input period: 31:27 <Reserved> 26:0 Measured last period
acc_err	Accumulated pulse position error between the ADPLL output clock and the input clock: 31:27 <Reserved> 26:0 Accumulation error *Sign differences between the last measured error and the accumulated position error result in the accumulated error zeroing out.

Table 2-16 Bit Mapping of ADPLL Status Registers

2.9 Retrieving PTP Status

PTP-specific status information can be retrieved with the `DqSyncGetPTPStatus()` API.

Table 2-17 API for Retrieving PTP Status

API Name	Description
`DqSyncGetPTPStatus`	Provides status for the PTP server if PTP is enabled on the IOM. Parameters: • `int handle`: Handle to the IOM • `int mode`: Reserved, set to 0 • `pDQ_SYNC_PTP_STAT pPTPstat`: See notes below

The `DQ_SYNC_PTP_STAT` structure consists of the following elements:

```
typedef struct {
    uint32 state;
    uint64 grandMasterClockID;
    uint64 masterClockID;

    uint32 stepsFromGrandMaster;
    uint32 grandMasterClockClass;

    int32 meanPathDelay;
    int32 lastMeasuredOffset;
    int32 maxMeasuredOffset;
    int32 minMeasuredOffset;
    int32 avgMeasuredOffset;

    // packet statistics
    uint32 totalPkts;
    uint32 annouceRcvd;
    uint32 annouceSnt;
    uint32 syncRcvd;
    uint32 syncSnt;
    uint32 followUpRcvd;
    uint32 followUpSnt;
    uint32 delyReqRcvd;
    uint32 delyReqSnt;
    uint32 delyRspRcvd;
    uint32 delyRspSnt;
    uint32 signalingRcvd;
    uint32 signalingSnt;
} DQ_SYNC_PTP_STAT, *pDQ_SYNC_PTP_STAT;
```

PTP Status Register	Description
uint32 state	Current PTP state: • 1: DQ_PTP_PORT_STATE_INIT • 2: DQ_PTP_PORT_STATE_FAULTY • 3: DQ_PTP_PORT_STATE_DISABLED • 4: DQ_PTP_PORT_STATE_LISTENING • 5: DQ_PTP_PORT_STATE_PRE_MASTER • 6: DQ_PTP_PORT_STATE_MASTER • 7: DQ_PTP_PORT_STATE_PASSIVE • 8: DQ_PTP_PORT_STATE_UNCALIBRATED • 9: DQ_PTP_PORT_STATE_SLAVE
uint64 grandMasterClockID	PTP clock ID of the grand master of the system: When a UEI Cube or RACK is the PTP grandmaster, an EUI-64 format address is generated from our 48-bit MAC address: *1st 3 octets of Grandmaster MAC address + FFFE + last 3 octets*
uint64 masterClockID	PTP clock ID of the current master: When a UEI Cube or RACK is the PTP grandmaster, an EUI-64 format address is generated from our 48-bit MAC address: *1st 3 octets of current master MAC address + FFFE + last 3 octets*
uint32 stepsFromGrandMaster	Value of the PTP data set steps
uint32 grandMasterClockClass	PTP grand master clock class taken from the master's Announce messages *Clock class is 248 for UEI chassis*
int32 meanPathDelay	Current calculated mean path delay
int32 lastMeasuredOffset	Last calculated time offset from PTP master
int32 maxMeasuredOffset	Maximum calculated time offset from PTP master
int32 minMeasuredOffset	Minimum calculated time offset from PTP master
int32 avgMeasuredOffset	Average calculated time offset from PTP master
uint32 totalPkts	Total PTP packets received on the domain
uint32 annouceRcvd	Counter of Announce packets accepted *Note the counter will not increment if a message is rejected because the IOM is in the wrong state*
uint32 annouceSnt	Counter of Announce packets sent *Note the counter will not increment if a message is rejected because the IOM is in the wrong state*
uint32 syncRcvd	Counter of Sync packets accepted *Note the counter will not increment if a message is rejected because the IOM is in the wrong state*

Table 2-18 PTP Status Registers

PTP Status Register	Description
`uint32 syncSnt`	Counter of Sync packets sent *Note the counter will not increment if a message is rejected because the IOM is in the wrong state*
`uint32 followUpRcvd`	Counter of Follow Up packets accepted *Note the counter will not increment if a message is rejected because the IOM is in the wrong state*
`uint32 followUpSnt`	Counter of Follow Up packets sent *Note the counter will not increment if a message is rejected because the IOM is in the wrong state*
`uint32 delyReqRcvd`	Counter of Delay Request packets accepted *Note the counter will not increment if a message is rejected because the IOM is in the wrong state*
`uint32 delyReqSnt`	Counter of Delay Request packets sent *Note the counter will not increment if a message is rejected because the IOM is in the wrong state*
`uint32 delyRspRcvd`	Counter of Delay Response packets accepted *Note the counter will not increment if a message is rejected because the IOM is in the wrong state*
`uint32 delyRspSnt`	Counter of Delay Response packets sent *Note the counter will not increment if a message is rejected because the IOM is in the wrong state*
`uint32 signalingRcvd`	Counter of Signaling packets accepted *Note the counter will not increment if a message is rejected because the IOM is in the wrong state*
`uint32 signalingSnt`	Counter of Signaling packets accepted *Note the counter will not increment if a message is rejected because the IOM is in the wrong state*

Table 2-18 PTP Status Registers

2.10 Retrieving UTC Time

If using PTP synchronization, you can retrieve the Coordinated Universal Time (UTC) from the PTP packets using the `DqSyncGetUTCTimeFromPTP()` API.

Table 2-19 API for Retrieving Status

API Name	Description
`DqSyncGetUTCTimeFromPTP`	Retrieves time. If slave chassis, will retrieve time from PTP packets (relative to time standard of PTP master) Parameters: • `int handle`: Handle to the IOM • `int mode`: Reserved, set to 0 • `pDQ_SYNC_UTC_TIME ptpUTC`: See notes below

The `DQ_SYNC_UTC_TIME ptpUTC` structure consists of the following elements:

```
typedef struct {
   uint32 reserved;  // reserved
   uint32 sec;       // PTP time in seconds
   uint32 nsec;      // PTP time nanoseconds
   uint32 timestamp; // timestamp from the CPU layer
                     //   to the time above
   uint32 flags;     // status flags
} DQ_SYNC_UTC_TIME, *pDQ_SYNC_UTC_TIME;
```

`ptpUTC->flags` will have the bit `DQ_SYNC_UTCTM_TIMEVALID` set if the returned time is valid, (i.e., UTC time is set).

Additional `flags` bits alert the user that the time may not actually be UTC time:

- `DQ_SYNC_UTCTM_ARB_PTPTSCALE`: The PTP master is using an Arbitrary Time Scale.
- `DQ_SYNC_UTCTM_OFFSINVLAID`: UTC offset from the master isn't valid.
- `DQ_SYNC_UTCTM_TIMEVALID`: Time is known. Either we are master or PTP slave.

© Copyright 2018
United Electronic Industries, Inc.

2.11 Disabling Sync / Releasing Sync Hardware

Sync scheme is disabled with the `DqSyncDisableSyncScheme()` API.

The API does the following:

- Disconnects SYNC lines
- Disconnects signals from sync connector
- Stops clocks in the Event Module/ADPLL
- Stops PTP handling I/O

Table 2-20 API for Retrieving Status

API Name	Description
`DqSyncDisableSyncScheme`	Disables synchronization scheme on Cube or RACK system Parameters: • `int handle`: Handle to the IOM • `uint32 status`: Reserved

© Copyright 2018
United Electronic Industries, Inc.

Chapter 3 System Configuration Tutorials

This chapter provides tutorials for configuring UEI chassis and supporting hardware for synchronization.

The following sections are included in this chapter:

- Configuring Synchronization to an External PPS (Section 3.1)
    - Connecting Hardware for 1PPS Synchronization (Section 3.1.1)
    - Configuring a UEI Chassis as 1PPS Master (Section 3.1.2)
    - Configuring a UEI Chassis as 1PPS Slave (Section 3.1.3)
    - Configuring Synchronized I/O Board Clocks (Section 3.1.4)
    - Configuring Synchronized Triggers & Timestamps (Section 3.1.5)

- Configuring Hardware for PTP Synchronization (Section 3.2)
    - Configuring PTP Interface Parameters (Section 3.2.1)
    - Configuring a PTP Grandmaster (Section 3.2.2)
    - Configuring a Boundary Clock (IEEE-1588-capable Switch) (Section 3.2.3)
    - Configuring a UEI Chassis for PTP Synchronization (Section 3.2.4)

NOTE: Example configurations in this chapter can be used as a reference when configuring your application. Many alternative configurations and hardware components exist that are not specified in this chapter.

This chapter provides step-by-step tutorials showing how to configure supporting hardware, as well as how to modify existing sample code to configure a UEI system.

This chapter focuses on configuring synchronization-specific parameters (not configuring data acquisition modes or I/O channel options).

Refer to **Chapter 4** for example code snippets for setting up a single chassis and single board for synchronization using various data acquisition modes.

3.1 Configuring Synchronization to an External PPS

This section provides instructions for configuring two UEI DNA-PPC5-1G chassis to synchronize using a common 1PPS signal.

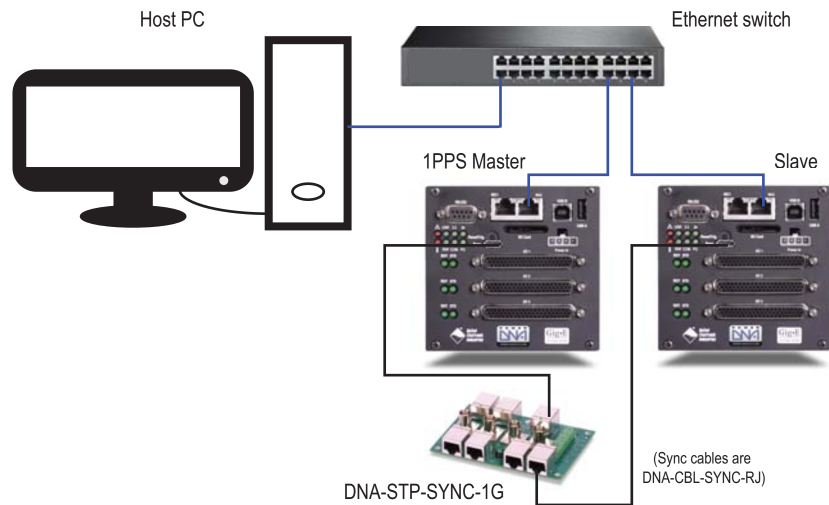


Figure 3-1 Example Hardware Configuration for External 1PPS

3.1.1 Connecting Hardware for 1PPS Synchronization

In this example, one chassis is configured as the 1PPS master and the other is a slave. They are connected using UEI's DNA-STP-SYNC-1G synchronization panel.

STEP 1: Connect the NIC1 ports of your UEI chassis and the NIC port of your host PC to a network switch.

STEP 2: Connect the 10-pin sync connector at the front of your master chassis to the MASTER port of the DNA-STP-SYNC-1G board using a DNA-CBL-SYNC-RJ cable.

STEP 3: Connect the 10-pin sync connector at the front of your slave chassis to any of the slave ports on the DNA-STP-SYNC-1G board.

STEP 4: Connect power, and power up your chassis.

© Copyright 2018
United Electronic Industries, Inc.

3.1.2 Configuring a UEI Chassis as 1PPS Master

This section explains how to program a UEI chassis to generate a 1PPS reference signal (master).

NOTE: You will need to program the IP address, device number, clocks, triggers, I/O channel configuration, and data acquisition modes to what your application requires.
This section specifically shows how to program your chassis to act as 1PPS master.

STEP 1: Open your application and/or a UEI synchronization code example.

We highly recommend you start with existing UEI synchronization sample code and update parameters instead of starting from scratch.

Refer to page 84 for location of sync sample code and naming conventions.

The following tutorial assumes you are starting from sample code.

STEP 2: Locate where `DQ_SYNC_SCHEME` structures are initialized in your code.

The parameters that control whether the chassis is a 1PPS master and specify 1PPS signal routing are highlighted in red:

```
typedef struct {
    // ==== section A ====================================================
    // IOM Sync Source Configuration
    uint32 sync_device;       // IOM CPU type (5200,8347,or 8347S with PTP capability)
    uint32 sync_source;       // where to get nPPS clock to synchronize system
    uint32 sync_line;         // which SYNC line to route external 1PPS clock
    uint32 sync_mode;         // mode of synchronization
    uint32 nPPS;              // N - number of pulses per second for input nPPS clock
    uint32 nPPS_us;           // Expected accuracy of the nPPS clock in us, clocks
                              //   outside of the range will be ignored, 0=default

    // ==== section B ====================================================
    // synchronization output: tell IOM to become 1PPS master
    uint32 sync_server;       // Identify chassis as 1PPS master
    uint32 srv_param;         // which sync connector pin routes 1PPS out to
                              //  the 1PPS slave chassis in the system
    uint32 trig_server;       // <Reserved>

    // ==== section C ====================================================
    // clocks: select clock source for each SYNC line (0 thru 3)
    uint32 clock_src[DQL_SYNC_LINES];     // clock source for each SYNC line
    uint32 clock_tmr[DQL_SYNC_LINES];     // PLL and external clock can be
                                          //   divided on TMR0 or TMR1
    uint32 clock_frq[DQL_SYNC_LINES];     // clock frequency (for PLL/EM)
    uint32 clock_div[DQL_SYNC_LINES];     // clock divider for EMx
                                          //   (0 == divide by 1, 2, 3 etc.)

    // ==== section D ====================================================
    // trigger: tell IOM where to get (or generate) and route trigger signal
    uint32 trig_source;         // where to take trigger to start acquisition
    uint32 trig_line;           // <reserved>
    uint32 trig_start;          // start trigger mode selection
    uint32 trig_delay;          // offset of the trigger pulse from nPPS clock\
                                //    (microseconds)
    uint32 trig_period_ms;      // period in ms to issue start trigger
    uint32 trig_stop;           // source for the stop trigger
    uint32 trig_stop_src;       // stop source for stop trigger upon N-count
    uint32 trig_duration;       // milliseconds before issuing stop trigger or N-count
```

```
    // ==== section E ====================================================
    // destination to route signals: from SYNC lines or to the outside SyncOut0/1
    uint32 clclk_dest[DQL_SYNC_LINES];  // where to feed CL clock
    uint32 pps_dest;                    // where to feed 1PPS clock to
    uint32 trig_dest;                   // where to feed start/stop trigger

} DQ_SYNC_SCHEME, *pDQ_SYNC_SCHEME;
```

STEP 3: Copy and paste an existing DQ_SYNC_SCHEME structure, and rename it to identify it as the master configuration (*_master) if one does not already exist.

STEP 4: In the DQ_SYNC_SCHEME *_master structure, modify the following:

Section A IOM Sync Source Configuration:

a. Set sync_device to DQ_SYNC_8347.
-- identifies the chassis type as a standard 1G cube or RACK.

b. Set sync_source to DQ_SYNCCLK_SYNCIN0.
-- configures which input pin (SyncIn0) will route in 1PPS.

c. Set sync_line to DQ_SYNCCLK_SYNC0.
-- routes the sync_source 1PPS to line 0 of the internal SYNC bus.

d. Set sync_mode to DQ_SYNCCLK_SYNC.
-- programs CPU to synchronize with external 1PPS.

e. Set nPPS to 1.
-- configures one pulse per second signal.

f. Set nPPS_us to 100.
-- sets expected accuracy of 1PPS (outside ranges are ignored).

Section B Synchronization Output Configuration:

g. Set sync_server to DQ_SYNCSRV_1PPS.
-- configures this chassis as a master.

h. Set srv_param to DQ_SYNCSRV_SYNCOUT0.
-- configures which output pin (SyncOut0) the generated 1PPS drives out of.

i. Set trig_server to 0.
-- reserved.

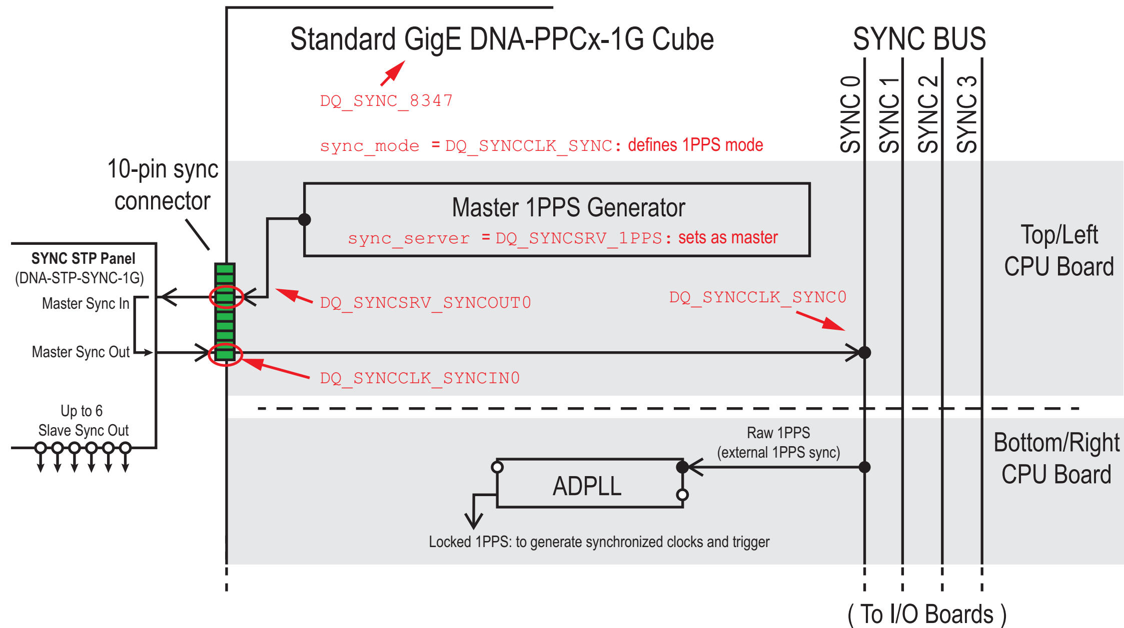


Figure 3-2 Block Diagram of Master Configuration

The following shows updated code for 1PPS-master configuration (the above block diagram highlights how hardware is configured):

```
DQ_SYNC_SCHEME sync_scheme_1PPS_master = {
// ==== section A =================================================================
// IOM synchronization
DQ_SYNC_8347,      // sync_device: 8347 standard chassis; 8347S -02/-03 rev chassis
DQ_SYNCCLK_SYNCIN0, // sync_source: raw 1PPS is routed into chassis via this pin
DQ_SYNCCLK_SYNC0,   // sync_line: raw 1PPS is routed to this SYNC line on backplane
DQ_SYNCCLK_SYNC,    // sync_mode: DQ_SYNCCLK_SYNC is external 1PPS mode
1,                  // nPPS: N - number of pulses per second for input nPPS clock
100,                // nPPS_us: Expected accuracy of the nPPS clock in us,
                    //          clocks outside of range will be ignored, 0=default

// ==== section B =================================================================
// Synchronization output (Master configuration)
DQ_SYNCSRV_1PPS,    // sync_server: configures chassis as 1PPS master
DQ_SYNCSRV_SYNCOUT0,// srv_param: generated 1PPS routes out chassis via this pin
0,                  // trig_server : <reserved>
// ==== section C, D, E not shown here
};
```

STEP 5: Locate the API that starts data acquisition (e.g., `DqRtVmapStartTr()`).

NOTE: Refer to section Section 4.2 on page 85 for more information about data acquisition modes.

STEP 6: Locate the `DqSyncDefineSyncScheme` API.

Verify that `DqSyncDefineSyncScheme` APIs are called *after* the API that starts data acquisition (located in previous step).

STEP 7: Add/update call to `DqSyncDefineSyncScheme,` and pass the `*_master` sync structure definitions.

```
if (master){
        DqSyncDefineSyncScheme(handle, &sync_scheme_1PPS_master, &status);}
```

STEP 8: Configure slave chassis, synchronized clocks, timestamps, and triggers as required by your application. Refer to the following sections for instructions:

- "Configuring a UEI Chassis as 1PPS Slave" on page 53
- "Configuring Synchronized I/O Board Clocks" on page 57
- "Configuring Synchronized Triggers & Timestamps" on page 62

3.1.3 Configuring a UEI Chassis as 1PPS Slave

This section explains how to program a UEI chassis to act as a slave to an external 1PPS source.

STEP 1: Open your application and/or a UEI synchronization code example.

We highly recommend you start with existing UEI synchronization sample code and update parameters instead of starting from scratch.

Refer to page 84 for location of sync sample code and naming conventions.

The following tutorial assumes you are starting from sample code.

STEP 2: Locate where `DQ_SYNC_SCHEME` structures are initialized in your code.

The parameters that control whether the chassis is a 1PPS slave and specify 1PPS signal routing are highlighted in red:

```
typedef struct {
    // ==== section A ====================================================
    // IOM Sync Source Configuration
    uint32 sync_device;      // IOM CPU type (5200,8347,or 8347S with PTP capability)
    uint32 sync_source;       // where to get nPPS clock to synchronize system
    uint32 sync_line;         // which SYNC line to route external 1PPS clock
    uint32 sync_mode;         // mode of synchronization
    uint32 nPPS;             // N - number of pulses per second for input nPPS clock
    uint32 nPPS_us;           // Expected accuracy of the nPPS clock in us, clocks
                              //   outside of the range will be ignored, 0=default

    // ==== section B ====================================================
    // synchronization output: tell IOM to become 1PPS master
    uint32 sync_server;       // Identify chassis as 1PPS master
    uint32 srv_param;         // which sync connector pin routes 1PPS out to
                              //  the 1PPS slave chassis in the system
    uint32 trig_server;       // <Reserved>

    // ==== section C ====================================================
    // clocks: select clock source for each SYNC line (0 thru 3)
    uint32 clock_src[DQL_SYNC_LINES];     // clock source for each SYNC line
    uint32 clock_tmr[DQL_SYNC_LINES];     // PLL and external clock can be
                                          //   divided on TMR0 or TMR1
    uint32 clock_frq[DQL_SYNC_LINES];     // clock frequency (for PLL/EM)
    uint32 clock_div[DQL_SYNC_LINES];     // clock divider for EMx
                                          //   (0 == divide by 1, 2, 3 etc.)

    // ==== section D ====================================================
    // trigger: tell IOM where to get (or generate) and route trigger signal
    uint32 trig_source;        // where to take trigger to start acquisition
    uint32 trig_line;          // <reserved>
    uint32 trig_start;         // start trigger mode selection
    uint32 trig_delay;         // offset of the trigger pulse from nPPS clock\
                               //    (microseconds)
    uint32 trig_period_ms;     // period in ms to issue start trigger
    uint32 trig_stop;          // source for the stop trigger
    uint32 trig_stop_src;      // stop source for stop trigger upon N-count
    uint32 trig_duration;      // milliseconds before issuing stop trigger or N-count
```

```
    // ==== section E ====================================================
    // destination to route signals: from SYNC lines or to the outside SyncOut0/1
    uint32 clclk_dest[DQL_SYNC_LINES];  // where to feed CL clock
    uint32 pps_dest;                    // where to feed 1PPS clock to
    uint32 trig_dest;                   // where to feed start/stop trigger

} DQ_SYNC_SCHEME, *pDQ_SYNC_SCHEME;
```

STEP 3: Copy and paste the DQ_SYNC_SCHEME structure, and rename it to identify it as the slave configuration (*_slave) if one does not already exist.

STEP 4: In the DQ_SYNC_SCHEME *_slave structure, modify the following:

Section A IOM Sync Source Configuration:

a. Set sync_device to DQ_SYNC_8347.
-- identifies the chassis type as a standard 1G cube or RACK.

b. Set sync_source to DQ_SYNCCLK_SYNCIN0.
-- configures which input pin (In0) will route in 1PPS.

c. Set sync_line to DQ_SYNCCLK_SYNC0.
-- routes the sync_source 1PPS to line 0 of the internal SYNC bus.

d. Set sync_mode to DQ_SYNCCLK_SYNC.
-- programs CPU to synchronize with external 1PPS.

e. Set nPPS to 1.
-- configures one pulse per second signal.

f. Set nPPS_us to 100.
-- sets expected accuracy of 1PPS (outside ranges are ignored).

Section B Synchronization Output Configuration:

g. Set sync_server to 0.
-- configures this chassis as a slave (not a master).

h. Set srv_param to 0.
-- configured NULL for slave.

i. Set trig_server to 0.
-- reserved.

© Copyright 2018
United Electronic Industries, Inc.

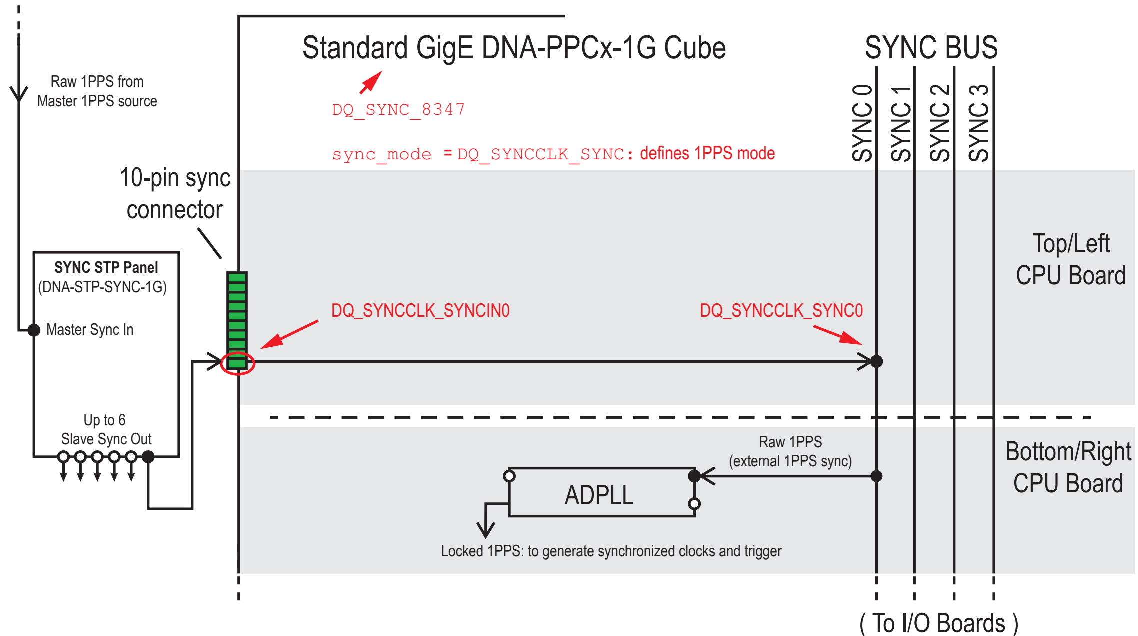


Figure 3-3 Block Diagram of Slave Configuration

The following shows updated code for initializing 1PPS-slave parameters (the block diagram above highlights how hardware will be configured):

```
DQ_SYNC_SCHEME sync_scheme_1PPS_slave = {
// ==== section A ===================================================================
// IOM synchronization
DQ_SYNC_8347,     // sync_device: 8347 standard chassis; 8347S is -02/-03 rev chassis
DQ_SYNCCLK_SYNCIN0, // sync_source: raw 1PPS is routed into chassis via this pin
DQ_SYNCCLK_SYNC0,   // sync_line: raw 1PPS is routed to this SYNC line on backplane
DQ_SYNCCLK_SYNC,    // sync_mode: DQ_SYNCCLK_SYNC is external 1PPS mode
1,                  // nPPS: N - number of pulses per second for input nPPS clock
100,                // nPPS_us: Expected accuracy of the nPPS clock in us,
                    //          clocks outside of range will be ignored, 0=default

// ==== section B ===================================================================
// Synchronization output (Master configuration)
0,    // sync_server: 0 configures chassis as 1PPS slave
0,    // srv_param: NULL for slave
0,                  // trig_server : <reserved>
// ==== section C, D, E not shown here
};
```

STEP 5: Locate the API that starts data acquisition (e.g., `DqRtVmapStartTr()`).

For your trigger to be synchronized with the 1PPS, verify you are using an API that does not send a software trigger (otherwise your board will already be triggered and not use the synchronized trigger). For example, `DqRtVmapStartTr` starts RtVMAP data acquisition mode but does not issue the trigger to collect samples.

NOTE: Refer to section Section 4.2 on page 85 for more information about data acquisition modes.

STEP 6: Locate the `DqSyncDefineSyncScheme` API.

Verify that `DqSyncDefineSyncScheme` APIs are called after the API that starts data acquisition (located in the previous step).

STEP 7: Add/update call to `DqSyncDefineSyncScheme` that passes the `*_slave` sync structure definitions.

```
DqSyncDefineSyncScheme(handle, &sync_scheme_1PPS_slave, &status);
```

STEP 8: Configure synchronized clocks, timestamps, and triggers as required by your application.

- “Configuring Synchronized I/O Board Clocks” on page 57
- “Configuring Synchronized Triggers & Timestamps” on page 62

3.1.4 Configuring Synchronized I/O Board Clocks

The following section provides steps for configuring synchronized clocks to multiple I/O boards in a UEI chassis.

The Event Module on the CPU board produces synchronized clocks from the locked 1PPS reference.

Each I/O board can receive the Event Module clock via the SYNC bus and can then further divide down the SYNC bus clock locally, as needed.

This tutorial provides instructions for programming the Event Module and routing the synchronized clock to I/O boards:

STEP 1: List all the I/O boards that require synchronization in a single chassis and at what clock rate those boards must run.

As an example, this tutorial will configure a slave chassis with 2 I/O boards that require the following sample rates:

- an analog input board (AI-207) requiring an output clock rate of 10 Hz
- an analog input board (AI-217) requiring an output clock rate of 1000 Hz

STEP 2: Determine the Event Module clock rate.

The clock rate produced by the Event Module must be evenly divisible by the required input clock rate of each I/O board that will use the synchronized clock.

Important Note about Oversampling I/O Boards

Several analog input boards use oversampled successive approximation (SAR) A/D converters. The clock source on these boards must be 8x the desired output rate of the ADC.
Boards that use 8x oversampling are the AI-211, AI-217, AI-218, and AI-228[1].

- In this example, the AI-217 requires an 8x input clock[2]; therefore, to get a 1000 Hz output sample rate, you must provide an 8000 Hz input clock. (Refer to the DNx-AI-217 User Manual for more information).
- In this example, the AI-207 requires a 10 Hz input clock.

To satisfy the above requirements, your Event Module clock rate can be set to 8000 Hz (both board rates, 8000 Hz and 10 Hz, are divisible).

1.Note that these boards also provide a 1x clock mode (called Scan-per-clock Mode) as an alternative. See the I/O board user manual for more information.
2.When configuring I/O channels for the AI-217 and other 8x oversampling boards, many users choose to disable the onboard digital FIR filter and bypass decimation for better control of timestamp alignment.

STEP 3: Determine clock dividers for each I/O board.

For this example, the Event Module is producing an 8000 Hz clock, and the local I/O board clock dividers are as follows:

- AI-207 @ 10 Hz output rates: The local divider will be 8000/10 = 800.
- AI-217 @ 1000 Hz output rates requires an 8x or 8000 Hz input clock: The local divider will be 8000/8000 = 1.

STEP 4: Open your application and/or a UEI synchronization code example.

The following tutorial assumes you are continuing from “Configuring a UEI Chassis as 1PPS Slave” on page 53; however, you could also continue from either of the following sections:

- “Configuring a UEI Chassis as 1PPS Master” on page 49
- “Configuring a UEI Chassis for PTP Synchronization” on page 76

STEP 5: Locate the `DQ_SYNC_SCHEME` structure in your code for the chassis with the I/O boards that your are programming.

NOTE: In this example, we will program the `*_slave` clocks.

The parameters that set the clock frequency and route a clock to I/O boards are highlighted in red:

```
typedef struct {
    // ==== section A ===================================================
    // IOM Sync Source Configuration
    uint32 sync_device;       // IOM CPU type (5200,8347,or 8347S with PTP capability)
    uint32 sync_source;        // where to get nPPS clock to synchronize system
    uint32 sync_line;          // which SYNC line to route external 1PPS clock
    uint32 sync_mode;          // mode of synchronization
    uint32 nPPS;              // N - number of pulses per second for input nPPS clock
    uint32 nPPS_us;            // Expected accuracy of the nPPS clock in us, clocks
                               //   outside of the range will be ignored, 0=default

    // ==== section B ===================================================
    // synchronization output: tell IOM to become 1PPS master
    uint32 sync_server;        // Identify chassis as 1PPS master
    uint32 srv_param;          // which sync connector pin routes 1PPS out to
                               //  the 1PPS slave chassis in the system
    uint32 trig_server;        // <Reserved>

    // ==== section C ===================================================
    // clocks: select clock source for each SYNC line (0 thru 3)
    uint32 clock_src[DQL_SYNC_LINES];      // clock source for each SYNC line
    uint32 clock_tmr[DQL_SYNC_LINES];      // PLL and external clock can be
                                           //   divided on TMR0 or TMR1
    uint32 clock_frq[DQL_SYNC_LINES];      // clock frequency (for PLL/EM)
    uint32 clock_div[DQL_SYNC_LINES];      // clock divider for EMx
                                           //   (0 == divide by 1, 2, 3 etc.)
```

© Copyright 2018
United Electronic Industries, Inc.

```
    // ==== section D ====================================================
    // trigger: tell IOM where to get (or generate) and route trigger signal
    uint32 trig_source;         // where to take trigger to start acquisition
    uint32 trig_line;           // <reserved>
    uint32 trig_start;          // start trigger mode selection
    uint32 trig_delay;          // offset of the trigger pulse from nPPS clock\
                                //    (microseconds)
    uint32 trig_period_ms;      // period in ms to issue start trigger
    uint32 trig_stop;           // source for the stop trigger
    uint32 trig_stop_src;       // stop source for stop trigger upon N-count
    uint32 trig_duration;       // milliseconds before issuing stop trigger or N-count

    // ==== section E ====================================================
    // destination to route signals: from SYNC lines or to the outside SyncOut0/1
    uint32 clclk_dest[DQL_SYNC_LINES];  // where to feed CL clock
    uint32 pps_dest;                    // where to feed 1PPS clock to
    uint32 trig_dest;                   // where to feed start/stop trigger

} DQ_SYNC_SCHEME, *pDQ_SYNC_SCHEME;
```

STEP 6: In the `DQ_SYNC_SCHEME` *_slave structure, modify the following:

Section C `Clock Configuration:`

a. Set `clock_src` to `{0, 0, DQ_CLOCKSRC_EM0,0}`.
-- routes the Event Module clock to SYNC line 2 (SYNC line 0, 1, and 3 are programmed with 0, which means no output clock resources are routed to them, but other resources, such as the PPS or triggers may be).

b. Set `clock_tmr` to `{0, 0, 0, 0}`.
-- <Reserved, set to 0>.

c. Set `clock_freq` to `{0, 0, 8000, 0}`.
-- programs the Event Module frequency on SYNC line 2 to 8000 Hz.

d. Set `clock_div` to `{0, 0, 0, 0}`.
-- programs no CPU clock dividers. Dividers will be programmed locally on each I/O board in a later step.

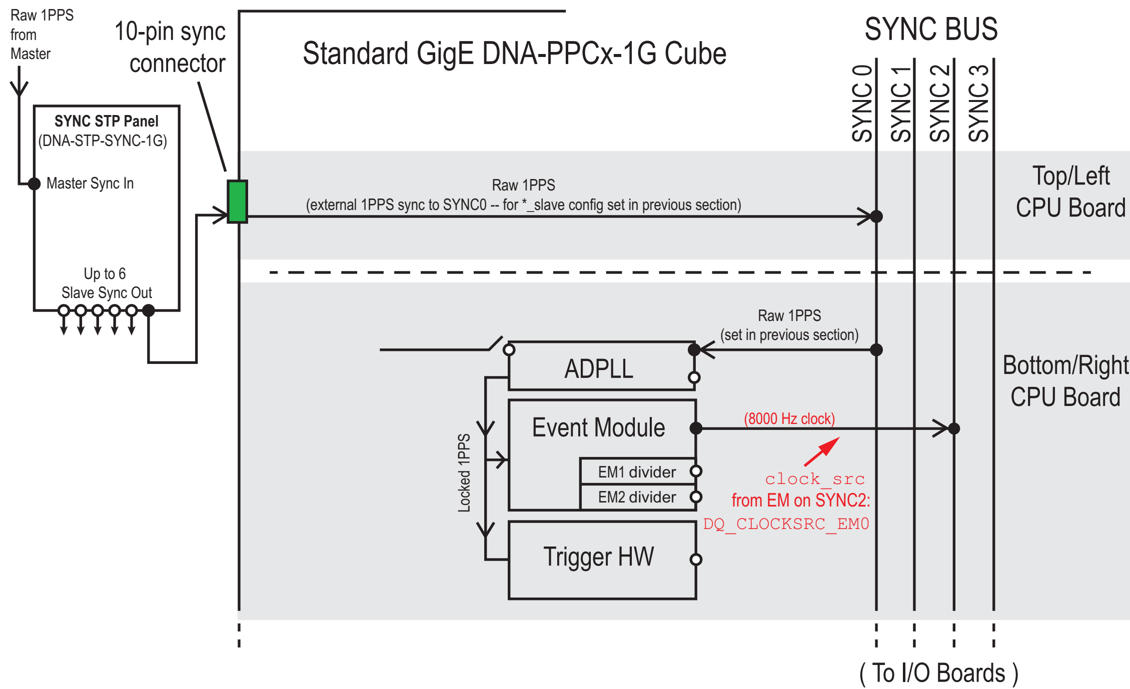


Figure 3-4 Block Diagram of Slave Configuration

The following code initializes clock parameters for the slave chassis (the block diagram above highlights the hardware connection):

```
DQ_SYNC_SCHEME sync_scheme_1PPS_slave = {
// ==== section A & B not shown here =============================================
...

// ==== section B ==================================================================
// Synchronization output (Master configuration)
{0, 0, DQ_CLOCKSRC_EM0, 0}, // clock_src (0 sets no clks connecting on SYNC 0,1,3)
{0, 0, 0, 0},               // clock_tmr: <Reserved, set to 0>
{0, 0, 8000, 0},            // clock_frq: programs clk on SYNC2 (EM) to 8000 Hz
{0, 0, 0, 0},               // clock_div: 0 sets no dividers on CPU board used

// ==== section D & E not shown here
...
};
```

STEP 7: Locate the `DqSyncDefineSyncScheme` API in your sample code.

```
DqSyncDefineSyncScheme(handle, &sync_scheme_1PPS_slave, &status);
```

STEP 8: For each I/O board requiring synchronized clocks, set clock parameters in a `DQ_SYNC_DEF_CLOCKS` structure to parameters calculated in steps 1 through 3, and then call the `DqSyncDefineLayerClock` API to configure hardware:

NOTE: `DqSyncDefineLayerClock()` must be called after `DqSyncDefineSyncScheme()`.

The elements in `DQ_SYNC_DEF_CLOCKS` are as follows:

```
typedef struct {
   int clk_line;   // SYNC line to use for clock source
   int divider;    // divider (0=no divider)
   int grp_delay;  // group delay in samples, -1 = auto define
   uint32 flags;   // flags to change mode of operation
} DQ_SYNC_DEF_CLOCKS, *pDQ_SYNC_DEF_CLOCKS;
```

For the AI-217 (DEVN0), program a divider of 0 (no divider) to produce a 1000 Hz clock:

```
DQ_SYNC_DEF_CLOCKS defclocks217 = {DQ_SYNCCLLK_SYNC2,0,-1,0};
DqSyncDefineLayerClock(handle, 0, &defclocks217);
```

For the AI-207 (DEVN0), program a divider of 800 (8000/800) to produce a 10 Hz clock:

```
DQ_SYNC_DEF_CLOCKS defclocks207 = {DQ_SYNCCLLK_SYNC2,800,-1,0};
DqSyncDefineLayerClock(handle, 1, &defclocks207);
```

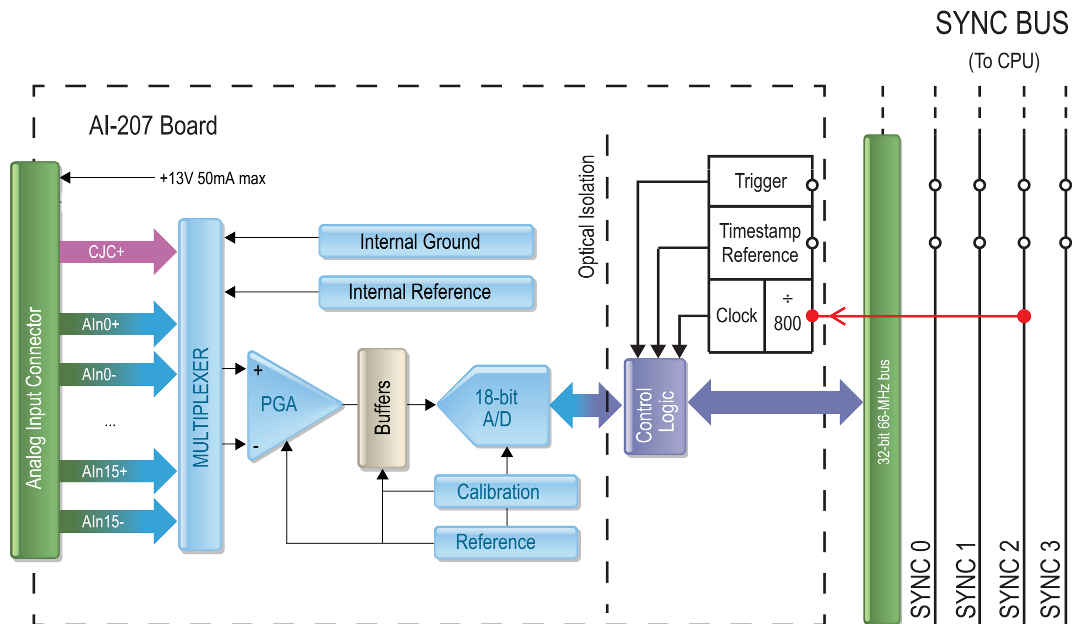


Figure 3-5 Diagram of Connecting Clock from SYNC2 to AI-207

STEP 9: Configure timestamps and triggers as required by your application.

- See “Configuring Synchronized Triggers & Timestamps” as follows.

3.1.5 Configuring Synchronized Triggers & Timestamps

The following section provides steps for configuring a trigger and timestamp reference synchronized with the 1PPS and provided to multiple I/O boards in a UEI chassis.

This tutorial assumes your code is open, and you're continuing from setting up your clocks (Section 3.1.4).

NOTE: For your trigger to be synchronized with the 1PPS, verify the API you use to start data acquisition does not send a software trigger (otherwise your board will already be triggered and not use the synchronized trigger). For example, in RtVMAP mode, use `DqRtVmapStartTr`, which starts data acquisition mode but does not automatically issue the trigger to collect samples.

STEP 1: Locate the `DQ_SYNC_SCHEME` structure in your code.

NOTE: In this example, we will program the `*_slave` chassis.
The steps will be the same for a `*_master` or, if you're setting up PTP synchronization, the `*_IEEE1588` structure.

The parameters that set the trigger and route it to I/O boards are highlighted in red:

```
typedef struct {
    // ==== section A ====================================================
    // IOM Sync Source Configuration
    uint32 sync_device;       // IOM CPU type (5200,8347,or 8347S with PTP capability)
    uint32 sync_source;       // where to get nPPS clock to synchronize system
    uint32 sync_line;         // which SYNC line to route external 1PPS clock
    uint32 sync_mode;         // mode of synchronization
    uint32 nPPS;              // N - number of pulses per second for input nPPS clock
    uint32 nPPS_us;           // Expected accuracy of the nPPS clock in us, clocks
    // ==== section B ====================================================
    // synchronization output: tell IOM to become 1PPS master
    uint32 sync_server;       // Identify chassis as 1PPS master
    uint32 srv_param;         // which sync connector pin routes 1PPS out to
    uint32 trig_server;       // <Reserved>
    // ==== section C ====================================================
    // clocks: select clock source for each SYNC line (0 thru 3)
    uint32 clock_src[DQL_SYNC_LINES];     // clock source for each SYNC line
    uint32 clock_tmr[DQL_SYNC_LINES];     // PLL and external divider
    uint32 clock_frq[DQL_SYNC_LINES];     // clock frequency (for PLL/EM)
    uint32 clock_div[DQL_SYNC_LINES];     // clock divider for EMx

    // ==== section D ====================================================
    // trigger: tell IOM where to get (or generate) and route trigger signal
    uint32 trig_source;        // SYNC line to route acquisition start trigger
    uint32 trig_line;          // <reserved, set to 0>
    uint32 trig_start;         // mode of start trigger mode
    uint32 trig_delay;         // offset of the trigger pulse from nPPS clock\
                               //    (microseconds)
    uint32 trig_period_ms;     // period in ms to issue start trigger
    uint32 trig_stop;          // source for the stop trigger
    uint32 trig_stop_src;      // stop source for stop trigger upon N-count
    uint32 trig_duration;      // milliseconds before issuing stop trigger or N-count
```

```
    // ==== section E ====================================================
    // destination to route signals: from SYNC lines or to the outside SyncOut0/1
    uint32 clclk_dest[DQL_SYNC_LINES];  // where to feed CL clock
    uint32 pps_dest;                    // where to feed 1PPS clock to
    uint32 trig_dest;                   // where to feed start/stop trigger

} DQ_SYNC_SCHEME, *pDQ_SYNC_SCHEME;
```

STEP 2: In the `DQ_SYNC_SCHEME *`_slave structure, modify the following:

Section D `Trigger Configuration:`

a. Set `trig_src` to `DQ_USE_SYNC3`.
-- routes the hardware trigger to SYNC line 3.

b. Set `trig_sync` to `0`.
-- <Reserved, set to 0>.

c. Set `trig_start` to `DQ_TRIGSTART_NPPS`.
-- generate trigger using CPU hardware synchronized to 1PPS.

d. Set `trig_delay` to `0`.
-- asserts trigger with no added delay (0 µs) after the 1PPS rising edge && the trigger is armed via an API.

e. Set `trig_period_ms` to `0`.
-- <Reserved, set to 0>.

f. Set `trig_stop`, `trig_stop_src`, and `trig_duration` to `0`.
-- no stop trigger used.

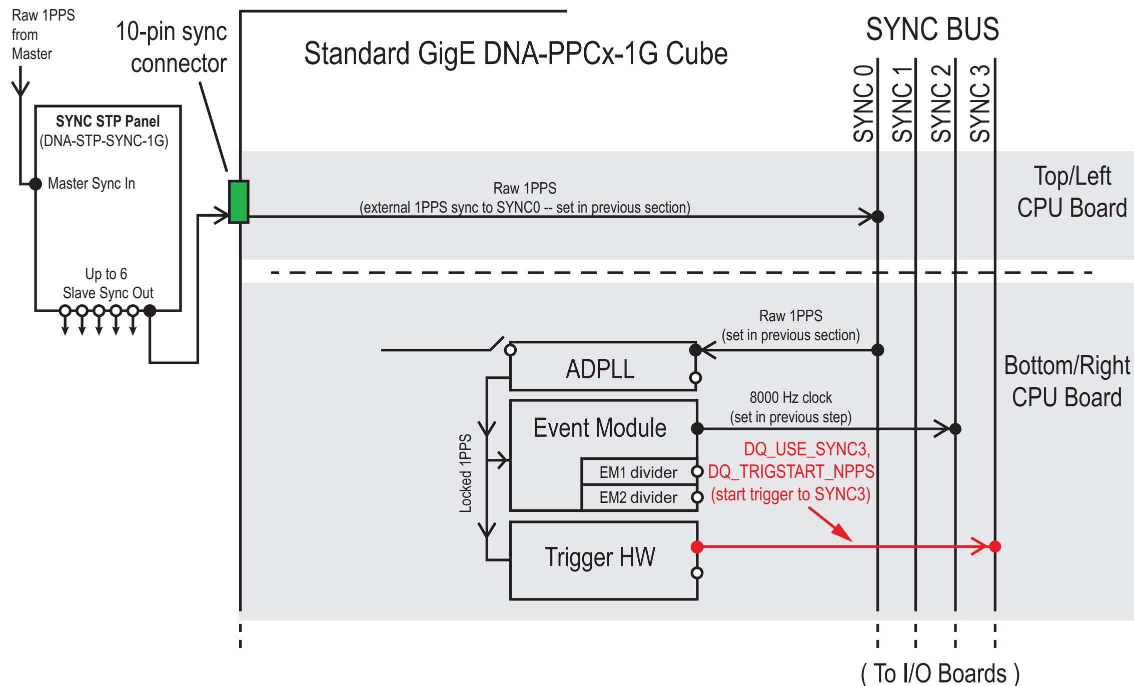


Figure 3-6 Block Diagram of Slave Configuration

The following initializes trigger parameters for the slave chassis:

```
DQ_SYNC_SCHEME sync_scheme_1PPS_slave = {
// ==== section A & B & C not shown here ==========================================
...
// ==== section D =================================================================
    // trigger: tell IOM where to get (or generate) and route trigger signal
    DQ_USE_SYNC3,      // SYNC 3 where to route for acquisition start
    0,                 // <reserved>
    DQ_TRIGSTART_NPPS, // start trigger mode selection
    0,                 // offset of the trigger pulse from nPPS clock (microseconds)
    0,                 // period in ms to issue start trigger
    0,                 // source for the stop trigger
    0,                 // stop source for stop trigger upon N-count
    0,                 // milliseconds before issuing stop trigger or N-count

// ==== section E (since not routing anything out of chassis these are 0s)
    {0, 0, 0, 0},      // <SYNC line clock to be routed out 10-pin sync connector>
    0,                 // <PPS destination out sync connector>
    0                  // <trigger destination out sync connector>
};
```

STEP 3: Locate the `DqSyncDefineSyncScheme` API in your sample code.

```
DqSyncDefineSyncScheme(handle, &sync_scheme_1PPS_slave, &status);
```

STEP 4: For each I/O board requiring synchronized triggers, add a call to the `DqSyncDefineLayerTrigger` API to configure I/O board hardware:

NOTE: `DqSyncDefineLayerTrigger()` must be called after `DqSyncDefineSyncScheme()`.

For the AI-217 (DEVN0):

```
DqSyncDefineLayerTrigger(handle, 0, DQ_SYNCTRG_SYNC3, 0);
```

For the AI-207 (DEVN1):

```
DqSyncDefineLayerTrigger(handle, 1, DQ_SYNCTRG_SYNC3, 0);
```

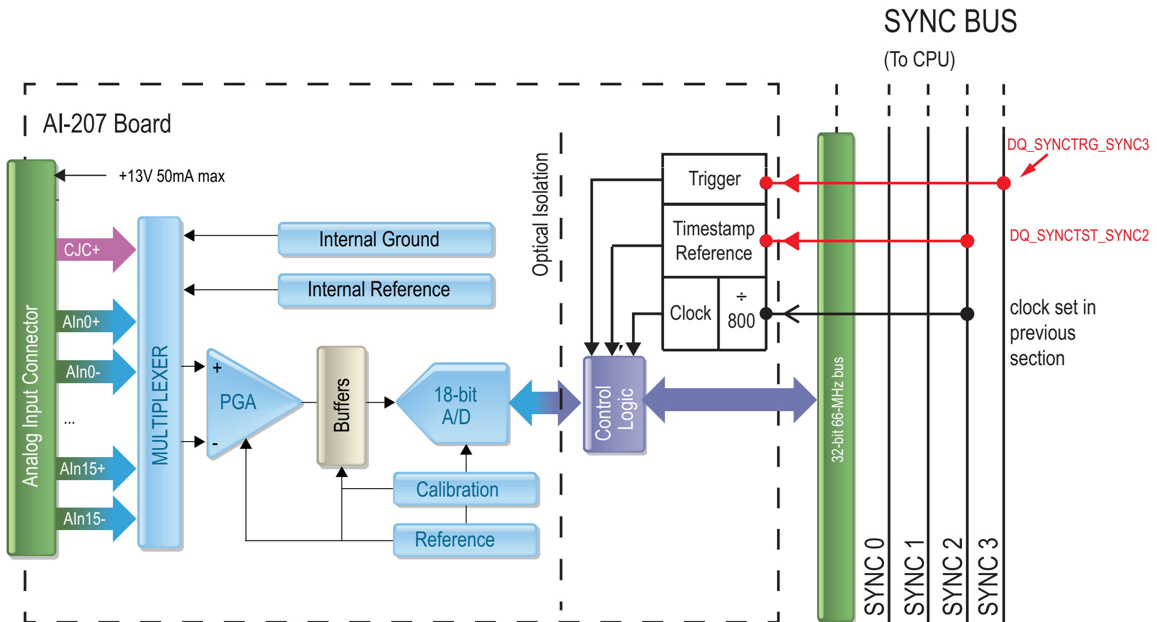


Figure 3-7 Diagram of Connecting Trigger & Timestamp Reference from SYNC to AI-207 Board

STEP 5: For each I/O board requiring synchronized timestamps, call the `DqSyncDefineLayerTimestamp` API to configure I/O board hardware:

NOTE: `DqSyncDefineLayerTimestamp()` must be called after `DqSyncDefineSyncScheme()`.

For the AI-217 (set first board in chassis, DEVN0, to use clock on SYNC 2 as the timestamp reference):

```
DqSyncDefineLayerTimestamp(handle, 0, DQ_SYNCTST_SYNC2, 0);
```

For the AI-207 (set second board in chassis, DEVN1, to use clock on SYNC 2 as the timestamp reference):

```
DqSyncDefineLayerTimestamp(handle, 1, DQ_SYNCTST_SYNC2, 0);
```

STEP 6: Optionally, verify that the clocks are stable:

The adaptive digital PLL (ADPLL) validation flags are read using the `DqSyncGetSyncStatus()` API.

```
// Check status registers to verify synchronization status states
int passing_validation; // holds validation status for user-determined checks
DQ_SYNC_STATUS astatus;

passing_validation = FALSE;
DqSyncGetSyncStatus(handle, 0, &astatus);

// Bits 2 & 1 of the astatus.adpll_sts.status register provide validation
//    status of the ADPLL 1PPS reference. '1' is passing validation
if ( ( astatus.adpll_sts.status & 4) && (astatus.adpll_sts.status & 2 ) )
    passing_validation = TRUE;
```

NOTE: `DqSyncGetSyncStatus()` returns status information from several status registers. Refer to the *PowerDNA API Reference Manual* for bit descriptions of each register.

STEP 7: Call `DqSyncTrigOnNextPPSBrCast` to broadcast a trigger to an array of UEI chassis.

```
DqSyncTrigOnNextPPSBrCast(handle, 1, 0, array_of_handles);
```

STEP 8: Optionally, send UDP broadcast to reset all timestamps on all I/O boards to 0.

```
DqCmdResetTimestampBrCast(handle, 0);
```

STEP 9: Configure any additional slave or master chassis in your system.

STEP 10: Save, build, and run.

3.2 Configuring Hardware for PTP Synchronization

This section provides instructions for configuring UEI chassis and other 1588 components to synchronize using IEEE-1588 PTP standard.

Network hardware configured in this section includes the following:

Component	Description of Component Used in This Example	Configuration Instructions
IEEE 1588 PTP grandmaster (master clock source)	Spectracom's SecureSync™ PTP grandmaster	Section 3.2.2
IEEE 1588-capable Industrial switch	Perle IDS-509 Managed Industrial Ethernet Switch, configured as an end-to-end boundary clock	Section 3.2.3
UEI slave chassis	-02 and -03 versions of UEI's GigE Cube and RACKtangle	Section 3.2.4
Host PC	For configuring PTP grandmaster and boundary clock and running the user application	

In this tutorial, the host PC, grandmaster clock, and UEI chassis are connected through the IEEE 1588-capable Industrial switch (the boundary clock):

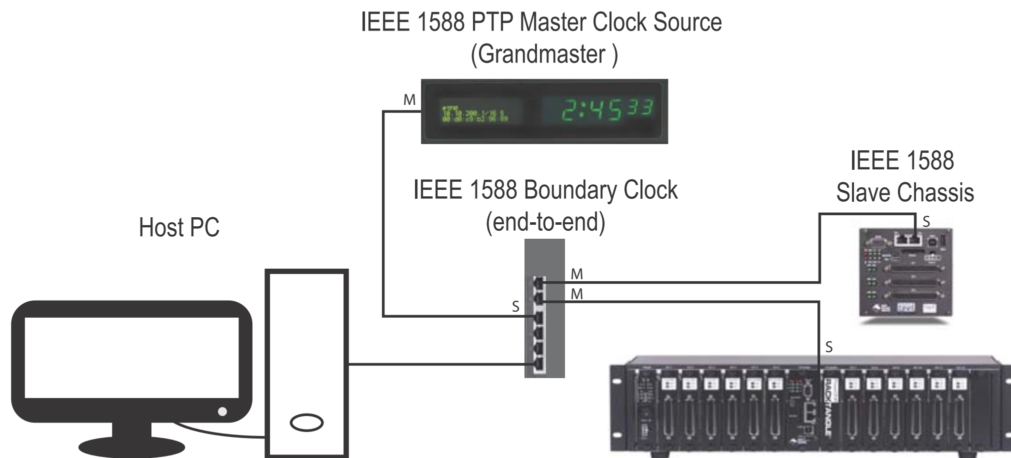


Figure 3-8 Example of PTP Hardware Configuration

3.2.1 Configuring PTP Interface Parameters

Table 3-1 lists IEEE-1588 parameters that must be set consistently for all PTP components in a system.

The table includes naming conventions used in the setup software for each component in this example and lists the value each will be set to in the following configuration instructions.

SecureSync™ Grandmaster Parameter	Perle Boundary Clock Parameter	UEI API Parameter	Values
Domain Number	Domain	uint8 subdomain	0
Multicast Announce Rate	Announce interval	int8 logAnnouceInterval	2^4 – 16 seconds
N/A	Announce timeout	uint8 annouceTimeout	3 messages
Multicast Sync Interval	Sync Interval	int8 logSyncInterval	2^0 – 1 second
N/A	Delay request interval	int8 logMinDelayRequestInterval	2^1 – 2 seconds

Table 3-1 PTP Interface Parameters for Each Component

The following are brief descriptions of each parameter:

- **PTP Domain**: Subdomain field that specifies the set of clocks in a multiple clock distribution system that are capable of synchronizing with each other.
- **PTP Log Announce Interval**: log2(period of announce messages) How often the PTP master clock sends Announce messages.
- **PTP Announce Receipt Timeout**: Number of announce intervals allowed to transpire without the slave receiving an Announce message from the master. After this delay, the slave will timeout.
- **PTP Log Sync Interval**: log2(period of sync messages) How often the PTP master clock sends Sync messages in multicast mode.
- **PTP Log Min Delay Request**: log2(minimum space between delay requests) Minimum interval allowed between PTP delay-request messages.

Interface parameters of the same type should be set to the same value across components.

For example, the SecureSync™ **Domain number**, the Perle **Domain**, and UEI chassis' **`subdomain`** parameters should all be set to **0** for this system to work.

3.2.2 Configuring a PTP Grandmaster

This tutorial provides configuration steps for setting up a SecureSync™ as a PTP grandmaster.

Note that a UEI chassis can alternatively act as the PTP master (to do this, follow steps in Section 3.2.4, but for the chassis you want as grandmaster, set its Priority to a low value (less than 128)).

To configure the SecureSync™ as a PTP grandmaster, do the following:

STEP 1: Connect the SecureSync™ PTP master port to a port on the boundary clock.

Note that the SecureSync™ PTP master module is optional on the SecureSync™ but required for our configuration. The PTP master port requires an SFP transceiver to connect to your Ethernet cable:

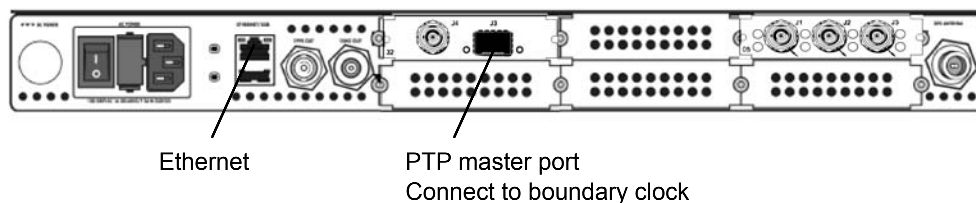


Figure 3-9 Rear of the SecureSync™ PTP Grandmaster

STEP 2: Connect a port on the boundary clock to the NIC port on your host PC.

STEP 3: Open a web browser to access the SecureSync™ web device manager.

STEP 4: Enter the SecureSync™ IP address in the browser's URL bar, press Enter, and log in at the LogIn screen. The SecureSync™ web device manager dashboard will display.

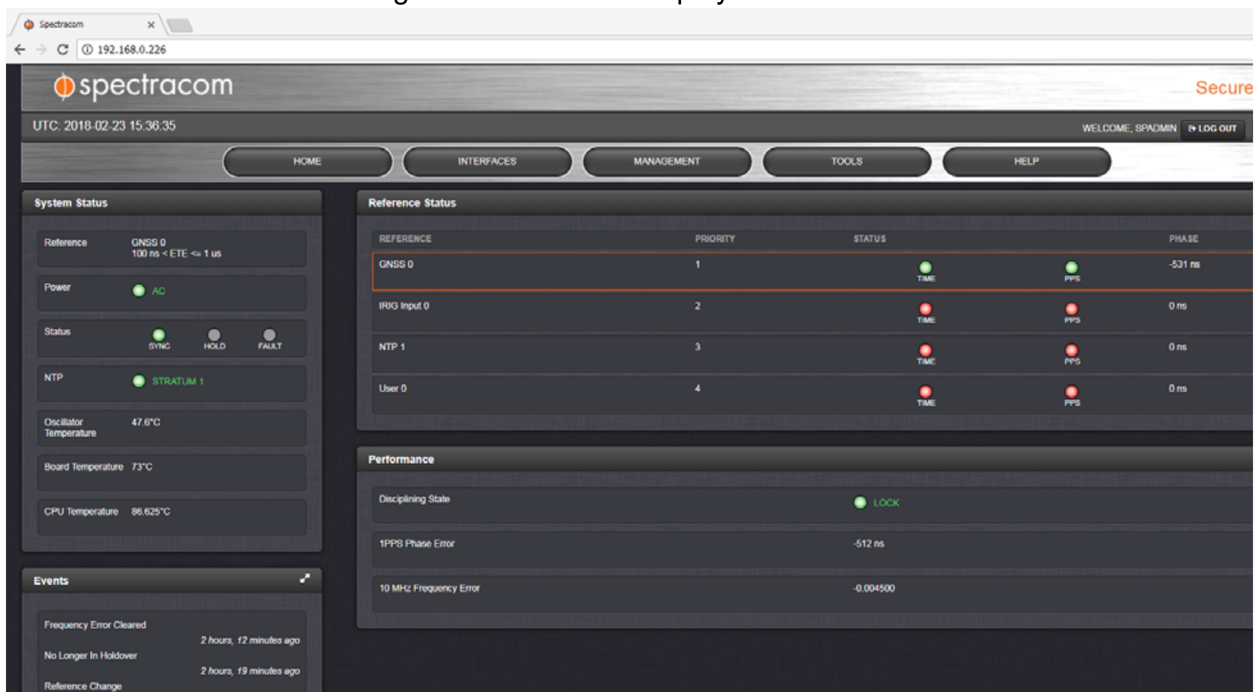


Figure 3-10 Spectracom Grandmaster Dashboard

STEP 5: In the SecureSync™ dashboard, click the **Interfaces** button, and in the pulldown menu under **OUTPUTS**, click **Gb PTP 0**.
A **Gb PTP 0** screen will open.

STEP 6: Enter the following configuration values in the **Gb PTP 0** window:

- **Enable PTP**: click box to enable
- **Profile**: default

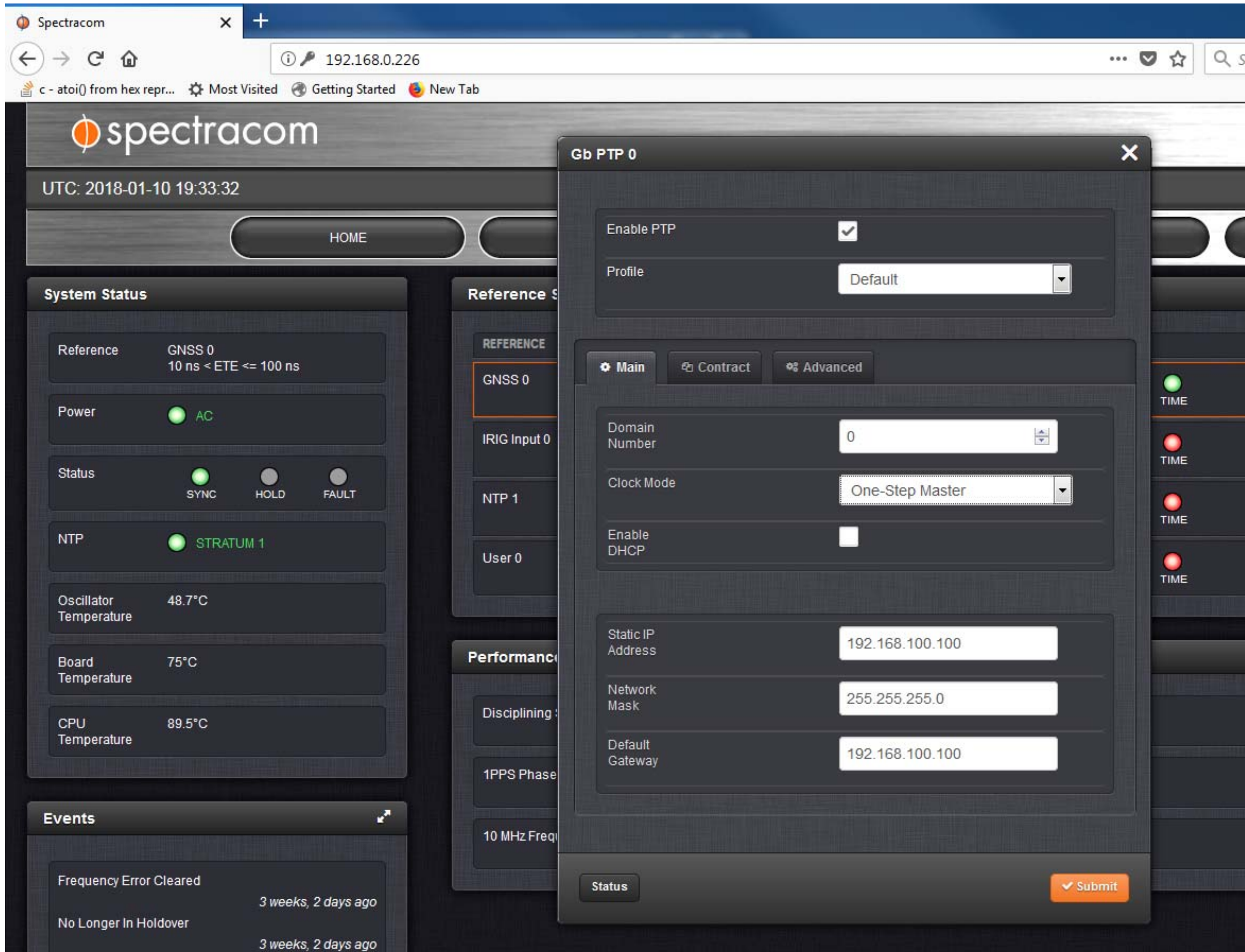


Figure 3-11 Spectracom Grandmaster PTP Config Screen

STEP 7: Under the **Main** tab in the **Gb PTP 0** window, enter the following (use defaults for parameters not listed):

- **Domain Number**: 0 (must be set to the same value on boundary clock and UEI chassis -- refer to **Table 3-1**)
- **Clock Mode**: One-Step Master
- **Static IP Address, Network Mask, Default Gateway**: set to what master PTP port addressing will be

STEP 8: Click the **Advanced** tab.

STEP 9: Under the **Advanced** tab in the **Gb PTP 0** window, enter the following (use defaults for parameters not listed):

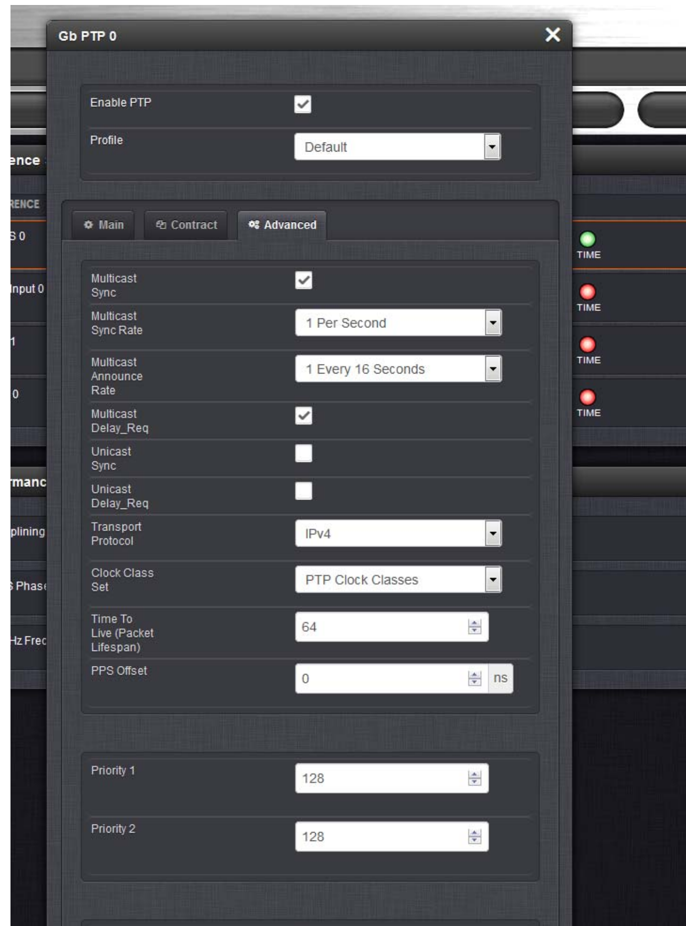


Figure 3-12 Spectracom PTP Grandmaster PTP Advanced Screen

- **Multicast Sync**: click box to enable (current UEI implementation does not support unicast packets)
- **Multicast Sync Rate**: 1 Per Second (must be set to the same value on boundary clock and UEI chassis -- refer to **Table 3-1**)
- **Multicast Announce Rate**: 1 Per 16 Seconds (must be set to the same value on boundary clock and UEI chassis -- refer to **Table 3-1**)
- **Multicast Delay_Req**: click box to enable
- **Transport Protocol**: IPv4
- **Clock Class Set**: PTP Clock Classes
- **Time to Live (Packet Lifespan)**: 64
- **PPS Offset**: 0 ns
- **Priority 1 / Priority 2**: These default to 128. Lower numbers increase priority in BMCA algorithm

STEP 10: Click orange **Submit** button.

3.2.3 Configuring a Boundary Clock (IEEE-1588-capable Switch)

The following configuration steps assume you've already completed an initial configuration of the boundary clock's IP address, username, password, and other non-PTP-related setup.

To configure the boundary clock, do the following:

STEP 1: Verify your host PC and the SecureSync™ grandmaster are both connected to a boundary clock port.

STEP 2: Connect the NIC1 ports on your UEI Cubes and RACKtangles to ports on the boundary clock:

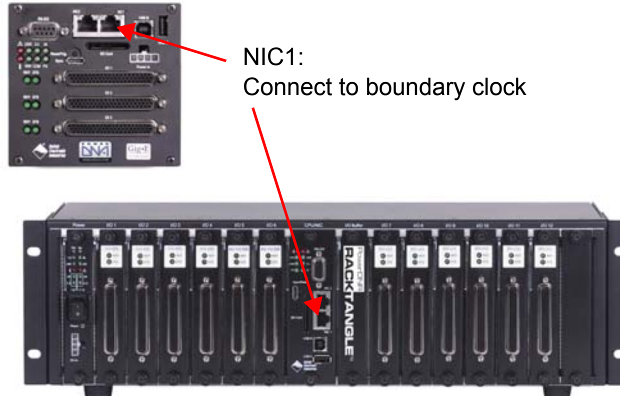


Figure 3-13 UEI NIC1 Ports

STEP 3: Open a web browser to access the boundary clock's web device manager.

STEP 4: Enter the boundary clock's IP address in the browser's URL bar, press **Enter**, and log in at the **LogIn** screen. The web device manager dashboard will display.

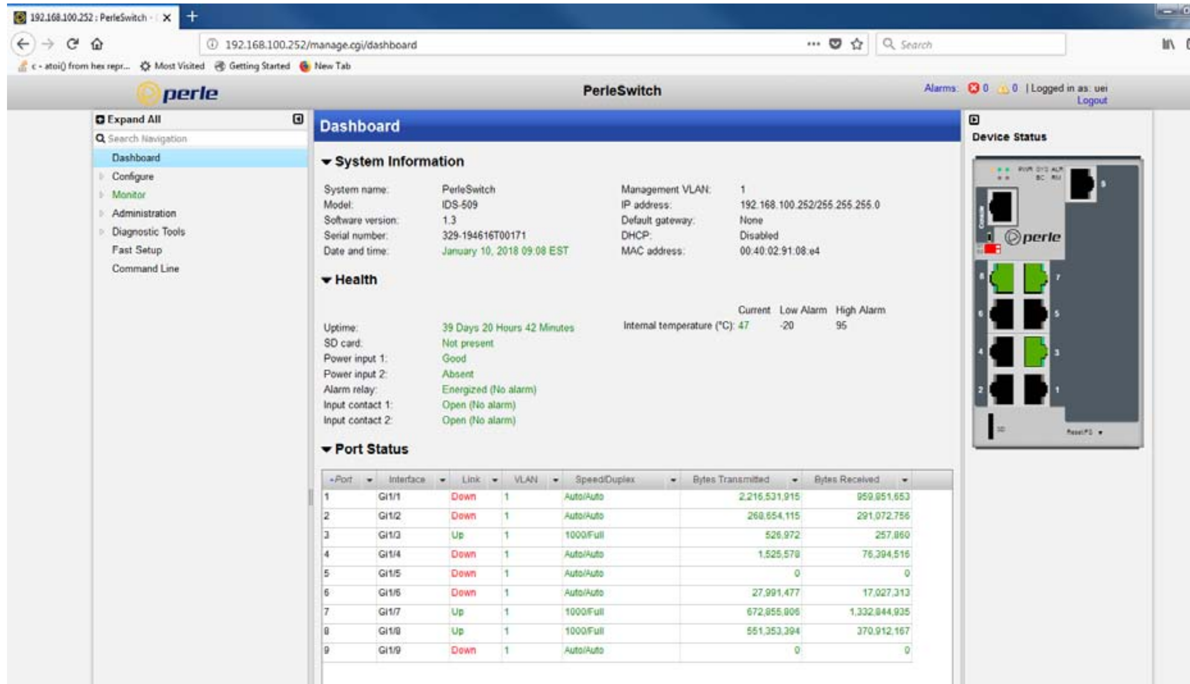


Figure 3-14 Boundary Clock Dashboard

STEP 5: In the left sidebar, click **Configure**, **Time Protocols**, and then **PTP**. The **PTP Settings** panel will display.

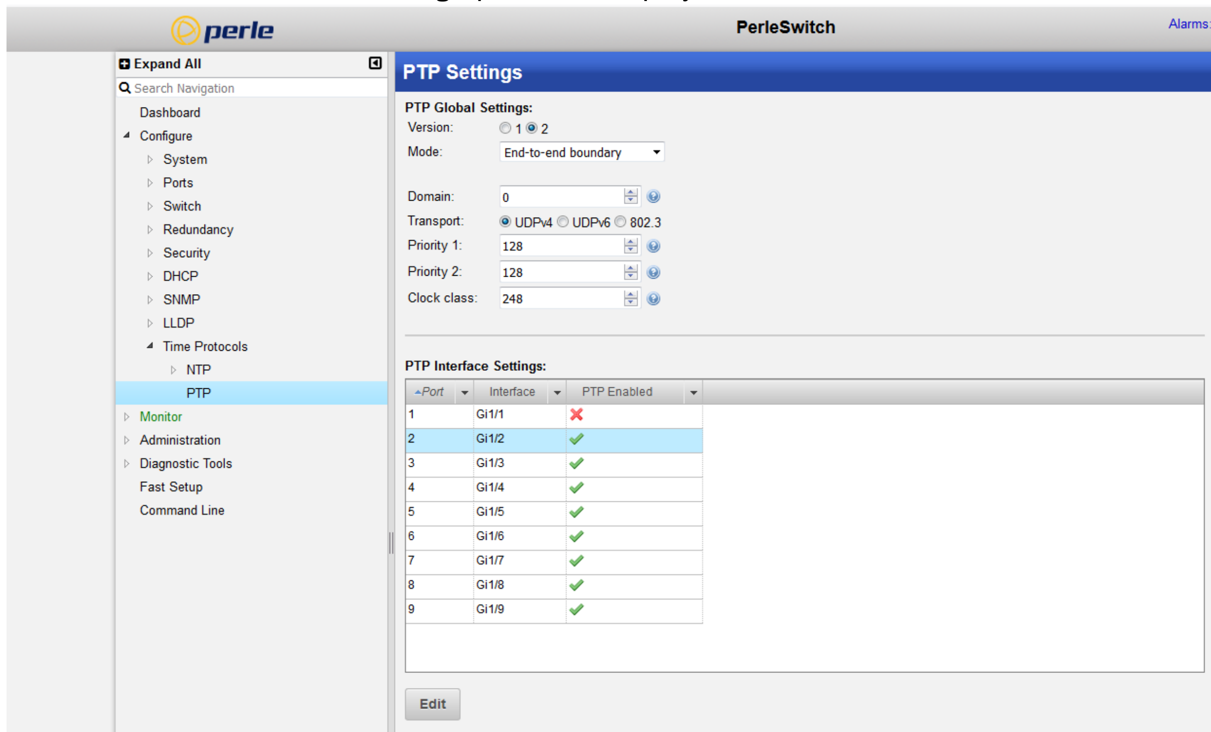


Figure 3-15 Boundary Clock PTP Settings Screen

STEP 6: Enter the following **PTP Global Settings**:

- **Version**: click 2
- **Mode**: End-to-end boundary (this is required for UEI chassis)
- **Domain**: 0 (must be set to the same value on grandmaster and UEI chassis -- refer to **Table 3-1**)
- **Transport**: UDPv4 (this setting is required for UEI chassis)
- **Priority 1 / Priority 2**: These default to 128. Lower numbers increase priority in BMCA algorithm
- **Clock class**: 248 (the default)

NOTE: The lower **PTP Interface Settings** panel is used to configure individual ports on the boundary clock. The ports that the PTP grandmaster and UEI chassis are connected to must show as **PTP Enabled** in this table.

STEP 7: In the **PTP Interface Settings** table, click a port that is connected to a UEI chassis or the PTP grandmaster and then click the **Edit** button. An **Edit PTP Interface Settings** window will open.

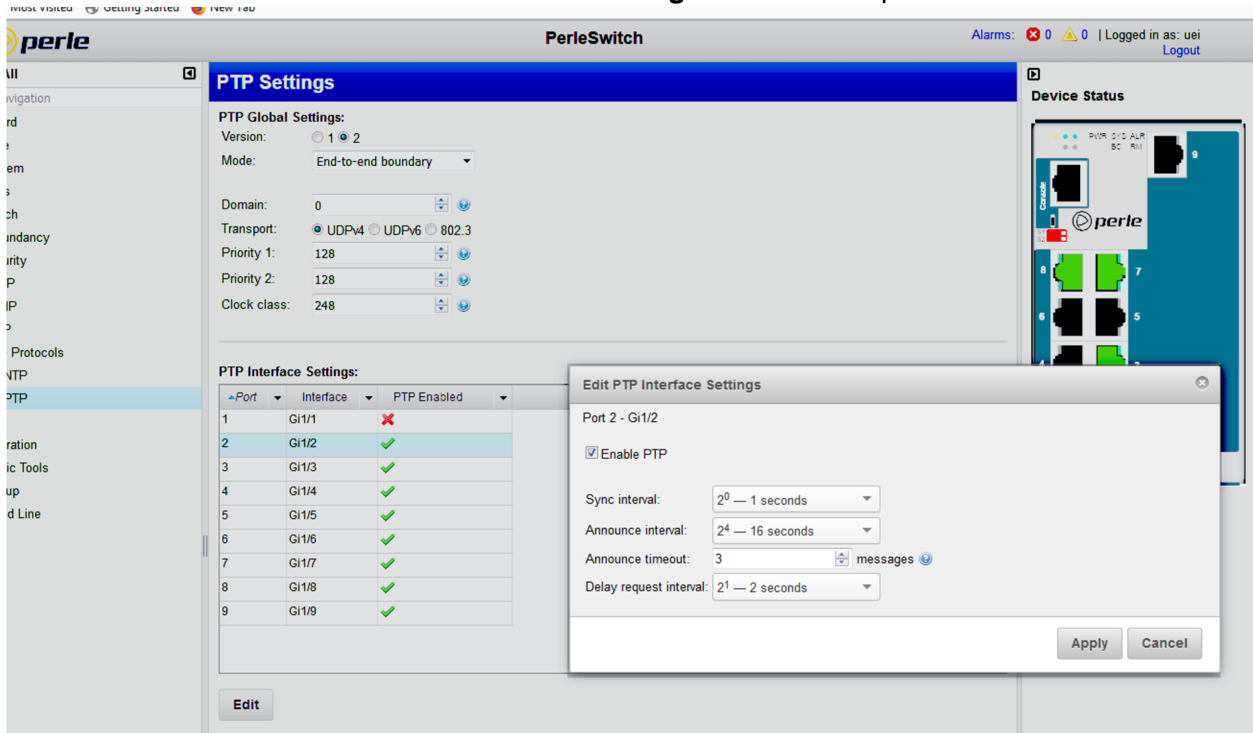


Figure 3-16 Boundary Clock PTP Settings Screen

STEP 8: In the **Edit PTP Interface Settings** window, enter the following parameters:

- **Enable PTP**: click box to enable
- **Sync interval**: 2^0 - 1 Seconds (must be set to the same value on grandmaster and UEI chassis -- refer to **Table 3-1**)
- **Announce interval**: 2^4 - 16 Seconds (must be set to the same value on grandmaster and UEI chassis -- refer to **Table 3-1**)
- **Announce timeout**: 3 messages (must be set to the same value on UEI chassis -- refer to **Table 3-1**)
- **Delay request interval**: 2^1 - 2 Seconds (must be set to the same value on UEI chassis -- refer to **Table 3-1**)

STEP 9: Click **Apply**.

STEP 10: Repeat steps 7, 8 and 9 for every port that needs PTP configuration (those connected to the grandmaster and UEI chassis).

3.2.4 Configuring a UEI Chassis for PTP Synchronization

UEI chassis are configured for IEEE-1588 synchronization by initializing PTP interface parameters with a `DqSyncDefinePTPServer` API and defining PTP mode in the `DqSyncDefineSyncScheme` API in your user application.

Note that the following tutorial is specifically for setting up the UEI chassis to use PTP synchronization.

NOTE: Instructions for setting up clocks and triggers in PTP mode are the same as in external 1PPS mode (see Section 3.1).

STEP 1: Verify the NIC1 ports of the UEI chassis are connected to boundary clock ports (configured as PTP ports in Section 3.2.3).

STEP 2: Open your application and/or a UEI synchronization code example.

We highly recommend you start with existing UEI synchronization sample code (1PPS or PTP) and update parameters instead of starting from scratch.

Refer to page 84 for location of sync sample code and naming conventions.

The following tutorial assumes you are starting from sample code.

STEP 3: Locate the `DQ_SYNC_SCHEME` structure for each chassis in your code.

Synchronization hardware is initialized using the `DQ_SYNC_SCHEME` structure. Parameters that affect PTP synchronization are highlighted in red:

```
typedef struct {
    // ==== section A =====================================================
    // IOM Sync Source Configuration
    uint32 sync_device;      // IOM CPU type (5200,8347,or 8347S with PTP capability)
    uint32 sync_source;       // external nPPS clock source
    uint32 sync_line;         // which SYNC line to route external 1PPS clock
    uint32 sync_mode;         // mode of synchronization
    uint32 nPPS;             // N - number of pulses per second for input nPPS clock
    uint32 nPPS_us;           // Expected accuracy of the nPPS clock in us

    // ==== section B =====================================================
    // synchronization output: tell IOM to become 1PPS master
    uint32 sync_server;
    uint32 srv_param;
    uint32 trig_server;

    // ==== section C =====================================================
    // clocks: select clock source for each SYNC line (0 thru 3)
    uint32 clock_src[DQL_SYNC_LINES];
    uint32 clock_tmr[DQL_SYNC_LINES];
    uint32 clock_frq[DQL_SYNC_LINES];
    uint32 clock_div[DQL_SYNC_LINES];
    // ==== section D =====================================================
    // trigger: tell IOM where to get (or generate) and route trigger signal
    uint32 trig_source;
    uint32 trig_line;
    uint32 trig_start;
    uint32 trig_delay;
    uint32 trig_period_ms;
    uint32 trig_stop;
    uint32 trig_stop_src;
    uint32 trig_duration;
```

```
// ==== section E ====================================================
    // destination to route signals: from SYNC lines or to the outside SyncOut0/1
    uint32 clclk_dest[DQL_SYNC_LINES];
    uint32 pps_dest;
    uint32 trig_dest;

} DQ_SYNC_SCHEME, *pDQ_SYNC_SCHEME;
```

STEP 4: For each chassis, set DQ_SYNC_SCHEME PTP-related parameters to the following:

- Set sync_device to DQ_SYNC_8347S.
  -- identifies the UEI chassis type as having IEEE-1588 hardware installed.
- Set sync_source to 0.
  -- sets the hardware to generate a 1PPS locally via PTP instead of routing a 1PPS in from an external source.
- Set sync_line to 0.
  -- tells the application that no external 1PPS reference signal needs to get routed internally.
- Set sync_mode to DQ_SYNCCLK_1588 | DQ_SYNCCLK_ETH0
  -- programs IEEE-1588 PTP protocol packet transfers over NIC1. NIC1 (DQ_SYNCCLK_ETH0) is the default; users could alternatively use NIC2 (DQ_SYNCCLK_ETH1).
- Set nPPS to 1.
  -- sets signal generated from PTP timestamps for internal synchronization to be a 1PPS signal.
- Set nPPS_us to 10000.
  -- sets acceptable jitter range of signal generated from PTP timestamps.
- Set sync_server, srv_param, trig_server to 0.
  -- not used in PTP synchronization.

The following code snippet shows initialization of the IEEE-1588-related parameters in the DQ_SYNC_SCHEME structure in sample code:

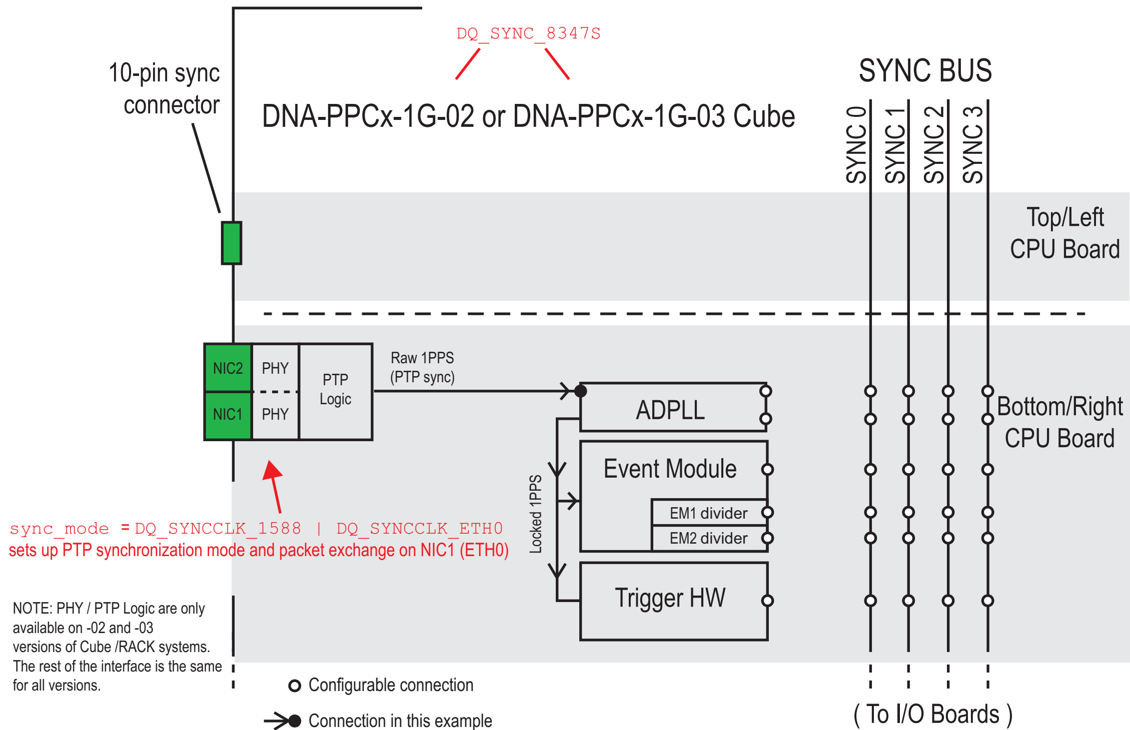


Figure 3-17 Block Diagram of PTP Configuration on UEI CPU Board

```
// Use an 8347 -02 or -03 CPU (1588 support)
// Synchronizes rack using 1588.
// Relies on DqSyncDefinePTPServer for advanced settings and BMC to establish master slave hierarchy.
DQ_SYNC_SCHEME sync_scheme_IEEE1588 = {
    // ==== section A ==========================================================================
    DQ_SYNC_8347S, // sync_device (DQ_SYNC_8347S is the CPU -2 and -3 versions)
    0, // sync_source
    0, // sync_line
    DQ_SYNCCLK_1588|DQ_SYNCCLK_ETH0,    // sync_mode;  mode of synchronization (1588 over NIC1)
    1, // nPPS;       N - number of pulses per second for input nPPS clock
    10000, // nPPS_us;    Expected accuracy of the nPPS clock in us, clocks outside of the range will be ignored, 0=default

    // ==== section B ==========================================================================
    0, // sync_server
    0, // srv_param
    0, // trig_server

    // ==== section C ==========================================================================
    // clocks
```

Figure 3-18 Code Snippet of Synchronization Structure Settings for PTP Sync

STEP 5: Add the PTP interface structure declaration with the other variable declarations if it is not already there:

```
DQ_SYNC_DEFPTP ptp_cfg;
```

STEP 6: Locate API that starts data acquisition:

If you want your trigger to be synchronized with the rising edge of the internal PPS, verify you are using an API that does not send a software trigger (otherwise your board will already be triggered and not use the synchronized trigger). For example, `DqRtVmapStartTr` starts RtVMAP data acquisition mode but does not issue the trigger to collect samples.

NOTE: Refer to section Section 4.2 on page 85 for more information about data acquisition modes.

STEP 7: Locate the `DqSyncDefineSyncScheme` API.

Verify that `DqSyncDefineSyncScheme` APIs are called after the API that starts data acquisition (located in the previous step).

STEP 8: Set the following PTP interface parameters, and call the `DqSyncDefinePTPServer` configuration API.

NOTE: `DqSyncDefinePTPServer` must be called **before** calling the `DqSyncDefineSyncScheme` API.

The following sets up a UEI chassis with the parameters defined in **Table 3-1**:

```
ptp_cfg.subdomain = 0;
ptp_cfg.logSyncInterval = 0;              // This will be 2^0, or 1 second
ptp_cfg.logMinDelayRequestInterval = 1;   // 2^1, or 2 seconds
ptp_cfg.logAnnouceInterval = 4;           // 2^4, or 16 seconds
ptp_cfg.annouceTimeout = 3;

// Used in BMCA: lower numbers have higher priority
ptp_cfg.priority1 = 128;
ptp_cfg.priority2 = 128;

// ptp_cfg.cfg config can be used for debug. 0=normal operation
ptp_cfg.cfg = 0;
ptp_cfg.utcOffset = 37;

DqSyncDefinePTPServer(handle, 0, &ptp_cfg);
DqSyncDefineSyncScheme(handle, &sync_scheme_IEEE1588, &status);
```

NOTE: `DqSyncDefinePTPServer` specifically sets up the PTP Interface parameters, and `DqSyncDefineSyncScheme` configures UEI hardware for synchronization.
Refer to the PowerDNA API Reference Manual for detailed API descriptions or to **Chapter 2** for an overview of synchronization API.

STEP 9: Optionally add monitoring of the PTP status.

You can find status monitoring in a `check_for_sync()` function in UEI synchronization sample code.

To monitor PTP status, add the following to `check_for_sync()`:

a. Declare the PTP status structure:
```
DQ_SYNC_PTP_STAT PTPstat;
```

b. Call API for each UEI chassis:
```
DqSyncGetPTPStatus(hd[i], 0, &PTPstat);
```

c. Evaluate status:
The following prints the PTP state, Grandmaster Clock ID, and Master Clock ID for UEI chassis `i`.

```
printf("State: %d: Grandmaster%d is: %x%x; PTP master%d is: %x%x \n",
                  PTPstat.state,
                  i,
                  (uint32)(PTPstat.grandMasterClockID>>32),
                  (uint32)PTPstat.grandMasterClockID,
                  i,
                  (uint32)(PTPstat.masterClockID>>32),
                  (uint32)PTPstat.masterClockID
);
```

STEP 10: If you have not set up synchronized clocks, timestamps, and triggers, configure as required by your application.

- “Configuring Synchronized I/O Board Clocks” on page 57
- “Configuring Synchronized Triggers & Timestamps” on page 62

STEP 11: Save and build.

3.2.4.1 Setting PTP Parameters Via the Serial Port

In PowerDNA hosted deployments (not UEIPACs), you have the option of initializing PTP configuration parameters over the serial port before starting your application. Note that you will still need to set `DQ_SYNC_SCHEME` parameters as described in the previous section.

Use the following steps to set up PTP interface parameters via the serial port:

STEP 1: Connect the UEI chassis to a host PC over the serial port:

a. Attach a serial cable to the host PC and RS-232 port on the front panel of the UEI chassis.

b. Run a serial terminal-emulation program (e.g., MTTTY) on the PC. Any terminal-emulation program, except HyperTerminal, may be used (MTTTY, Minicom, TeraTerm, PuTTY, etc.)

c. Verify that COM parameters are set at: 57600 baud, 8 bits, no parity, 1 stop bit.

d. Click **Connect** in MTTTY, or use the commands on one of the other terminal-emulation programs to establish communication with the UEI chassis.

STEP 2: Connect power to the UEI chassis, and power up.

Note that as soon as the system powers up, it runs through a self-diagnostic mode and generates output on the serial terminal program.

The boot process finishes with a display of the chassis model number, serial number, and slot positions of boards in the rack enclosure and states the uC/OS is running.

After the boot process completes, you will see a `DQ>` prompt in the serial terminal program for hosted UEI chassis.

STEP 3: At the prompt, type `show` to see the IEEE-1588 default parameters:

```
DQ> show

       name: "IOM-1"
      model: 3006
     serial: 0186837
     option: 0002
       fwct: 1.2.0.0
        mac: 00:0C:94:02:D9:D5
        srv: 192.168.100.2
         ip: 192.168.100.35 (1Gbit)
    gateway: 192.168.100.1
    netmask: 255.255.255.0
       mac2: 00:0C:94:F2:D9:D5
       srv2: 192.168.100.102
        ip2: 192.168.100.105 (DOWN)
   gateway2: 192.168.100.1
   netmask2: 255.255.255.0
        udp: 6334
    license: ""
   bond prm: bonding mode: FFFFFFFF
   1588 prm: master IP 255.255.255.255
             domain 255
             Log Announce Interval -1
             Announce Receipt Timeout 255
             Log Sync Interval: -1
   Manufactured 2/1/2018
   Calibrated 2/1/2018
```

NOTE: IEEE-1588 server parameters that are set via the `DqSyncDefinePTPServer` API call will not display with the `show` command. `show` only displays what parameters were updated to using the `set/store` commands.

STEP 4: Type `set 1588 prm` at the prompt, and enter each of the parameters (press Return to leave default values):

```
DQ> set 1588 prm
SETTING PTP PRAMS
fixed master IP [192.168.100.100] >
manually set master y/n >
PTP domain >0
PTP Log Announce Interval >4
PTP Announce Receipt Timeout >3
PTP Log Sync Interval >0
PTP Log min delay request >0
PTP priority 1 >125
PTP priority 2 >125
DQ>
```

NOTE: If you are asked for a password, the default password is `powerdna`.

STEP 5: Store the updated values to your system by typing `store` at the prompt.

STEP 6: Reset your chassis (either in hardware or by typing reset in the serial terminal window).

NOTE: PTP settings set over the serial port will be used by default, instead of requiring users to call `DqSyncDefinePTPServer` API. However, if the `DqSyncDefinePTPServer` API is called in a user application, the parameters set in the `DqSyncDefinePTPServer` API will override parameters set over the serial port.

Chapter 4 Code Examples

This chapter provides example code for reference when programming the PTP / PPS Sync Interface.

The following sections are included:

- About Sync Code Examples (Section 4.1)
- Supported Data Acquisition Modes for Sync Interface (Section 4.2)
- Example Code for Synchronization in RtVMap Mode (Section 4.3)
- Example Code for Synchronization in ACB Mode (Section 4.4)

4.1 About Sync Code Examples

The following sections contain code snippets and provide a brief overview of how to set up and use the synchronization interface using the low-level API.

For best results, use in conjunction with actual code samples, which can be found in the following directories:

- On Windows systems:
  Start » All Programs » UEI » PowerDNA » Examples » C Examples
- PowerDNA installations on Linux systems:
  <PowerDNA-x.y.z>/src/DAQLib_Samples
- UEIPAC installations on Linux systems:
  <ueipac-x.y.z>/sdk/examples

Note that the name of the mode and the name of the I/O boards being programmed are embedded in the sample name. For example, *SampleVMap207_217* contains sample code for synchronizing an AI-207 and AI-217 using RtVMap data acquisition mode.

See Section 4.2 for a brief description of each of the data acquisition modes.

NOTE: For users programming with the DAQLIB framework (C++, C#, LabVIEW, etc.), please refer to the *UeiDaq Framework User Manual*.

4.2 Supported Data Acquisition Modes for Sync Interface

The following data acquisition modes are supported for UEI chassis configured to use PTP or PPS synchronization:

- ACB: Advanced Circular Buffer
- RtDMAP: Data-Mapped I/O data transfers, with the timebase for transfers maintained by host application (1 sample per channel)
- RtVMAP: Variable-data-size Mapped I/O data transfers, with the timebase for transfers maintained by host application
- aDMAP: Data-Mapped input data transfers, with the timebase for transfers maintained by IOM (1 sample per channel)
- aVMAP: Variable-data-size Mapped input data transfers, with the timebase for transfers maintained by IOM

API that implement data acquisition modes are described in the *PowerDNA API Reference Manual*.

4.3 Example Code for Synchronization in RtVMap Mode

This section provides an overview of how to set up and synchronize a UEI chassis to an external 1PPS reference when acquiring data in RtVMap mode.

The following code snippets are provided:

- Initialization
  - Initializing the chassis, I/O boards, synchronization scheme, and RtVMap data transfer mode
- Configuration
  - Clearing any existing configuration from previous runs
  - Configuring channels and RtVMaps and configuring synchronization on each I/O board
- Verification (optional)
  - Verifying 1PPS validation reading ADPLL status
- Arm Trigger & Reset Timestamp
  - Sending broadcast commands to all configured IOMs
- Operation: Send / Receive Messages
  - Sending and receiving messages in RtVMap mode
- Stop Cleanly
  - Disabling boards cleanly

NOTE: Refer to the *PowerDNA API Reference Manual* for detailed descriptions of the API discussed in this section.

4.3.1 Initialization (RtVMap)

First we initialize synchronization and data acquisition settings.

4.3.1.1 Initialize Sync Interface (RtVMap)

UEI chassis are initialized for synchronization by defining parameters in a `DQ_SYNC_SCHEME` structure and passing those definitions via an API call. In UEI-provided sample code, this structure is declared globally and initialized in the declaration.

The following code snippet configures the sync interface as follows:

- a PPCx-1G IOM is configured as the 1PPS master
- I/O board clocks are generated in the Event Module (EM)
- Triggers are generated internal to the IOM and issued to all configured I/O boards on the rising edge of the locked 1PPS after armed by a broadcast message

```
DQ_SYNC_SCHEME sync_scheme_1PPS = {
// ==== section A ================================================
// IOM synchronization
// ADPLL takes 1PPS from sync line 0
    DQ_SYNC_8347,         // sync_device: 8347 CPU inside IOM being synched
    DQ_SYNCCLK_SYNCIN0,  // sync_source: raw nPPS routed in through
                         //    SyncIn0 of external sync connector
    DQ_SYNCCLK_SYNC0,    // sync_line: raw 1PPS routed from sync connector to
                         //    internal SYNC0 bus line
    DQ_SYNCCLK_SYNC,     // sync_mode: synchronization mode is 1PPS
    1,                   // nPPS: raw nPPS input is one pulse per second
    100,                 // nPPS_us: Expected accuracy of the raw nPPS clock
                         //    is 100 us, pulses outside of the range
                         //    will be ignored

// ==== section B ================================================
// Synchronization master configuration (synchronization output)
    DQ_SYNCSRV_1PPS,      // sync_server: IOM will generate its own 1PPS pulse
    DQ_SYNCSRV_SYNCOUT0,  // srv_param: raw 1PPS will route out SyncOut0
                          // on sync connector, which will be routed
                          //   back in over SyncIn0 and used as 1PPS for chassis
    0,                    // trig_server: <Reserved>, set to 0.


    // ==== section C ================================================
    // Clock configuration
    {0, 0, DQ_CLOCKSRC_EM0, 0},   // clock_src[]: PPS synched EM clock routed
                                  //  on SYNC2 internal bus line for
                                  //    distribution to I/O boards
    {0, 0, 0, 0},                 // clock_tmr[]: <Reserved>, set to 0.
    {0, 0, 8000, 0},              // clock_frq[]: clock frequency = 8 kHz
                                  //    on SYNC2
    {0, 0, 0, 0},                 // clock_div[]: no clock divider for EMx
```

© Copyright 2018
United Electronic Industries, Inc.

```
    // ==== section D ====================================================
    // Trigger configuration
    DQ_USE_SYNC3,       // trig_source: trigger will be routed to internal bus
                        //     line SYNC3 for distribution to I/O boards
    0,                  // trig_line: <Reserved>, set to 0.
    DQ_TRIGSTART_NPPS,// trig_start: trigger will be issued upon next nPPS
                        //     after armed by broadcast trigger
    0,                  // trig_delay: no trigger delay
    0,                  // trig_period_ms: <Reserved>, set to 0.
    0,                  // trig_stop: no trigger stop programmed
    0,                  // trig_stop_src: DQ_TRIGSTOP_NCLOCKS not set,
                        //     no src programmed
    0,                  // trig_duration: DQ_TRIGSTOP_NCLOCKS not set, no
                        //     duration programmed

    // ==== section E ====================================================
    // External Sync or Internal SYNC lines routing configuration
    {0, 0, 0, 0},       // clclk_dest[]: external routing already configured
    0,                  // pps_dest: not routing PPS from sync to external
    0                   // trig_dest: not routing trigger out sync connector
};
```

4.3.1.2 Initialize IOM (RtVMap)

To initiate communication with Cubes and RACKs in your system, you must first get a DAQLib handle for each IOM by calling `DqOpenIOM()`:

```
// Connect with all IOM and obtain library handle(s) for the connection(s)
DqOpenIOM(iom_ip,           // chassis IP address
          DQ_UDP_DAQ_PORT,  // host PC UDP port used communication
          1000,             // timeout duration in milliseconds
          &hd,              // pointer to the IOM handle
          &DQRdCfg);        // pointer to store echoed device results
                            //   NULL if not required
```

NOTE: `DqOpenIOM()` sets up communication.
Prior to that, you will also see a `DqInitDAQLib()` function in C examples. This function allocates resources and data structures for UEI's DAQLib libraries. The `DqInitDAQLib()` is not required when programming in a Windows environment; libraries are initialized automatically when loading the DLL on Windows systems. It is required when programming in Linux.

4.3.1.3 Initialize I/O Board Channels (RtVMap)

The following code snippet sets up board-specific initialization, (i.e., which channels are enabled, what gain is required, specific modes required) for the AI-207 analog input board:

```
int n; // temporary index
int clIdx = 0; // channel index
int data_chans; // variable to hold number of AI channels to configure
int cl[CHANNELS]; // CHANNELS is #defined as the total # enabled + TS
int flags[CHANNELS]; // flags will receive the status of FIFO
int avl_size, data_size;

data_chans = CHANNELS - TIMESTAMP; // TIMESTAMP is #defined as 1 for yes, 0 no

// Set up analog input channels: The gain and other modes are ORed in
// with the channel list and configured in a channel list array (cl)
// which gets passed later to the VMAP initialization

    for (n = 0; n < data_chans; n++) {
        cl[clIdx] = n | DQ_LNCL_GAIN(DQ_AI208_GAIN_1) | DQ_LNCL_DIFF;
        flags[clIdx] = DQ_VMAP_FIFO_STATUS;
        clIdx++;
    }

// If timestamps are enabled, the last item in the channel list will be
//  the timestamp value
    if (TIMESTAMP) {
        cl[clIdx] = DQ_LNCL_TIMESTAMP;
        flags[clIdx] = DQ_VMAP_FIFO_STATUS;
        clIdx++;
    }
```

4.3.1.4 Initialize VMAP

RtVMap data transfers are initialized for each IOM. VMAP initialization includes creating a unique VMAP ID.

```
int vmapid = -1; //initialize vmap

// Create VMap - each IOM will use a unique vmapid (hd is IOM handle)
DqrtVMapInit(hd, &vmapid, 1000);
```

© Copyright 2018
United Electronic Industries, Inc.

4.3.2 Configuration (RtVMap)

Configuration consists of setting up a VMAP buffer based on the number of channels in the channel list and then configuring the synchronization settings on each I/O board.

4.3.2.1 Configure RtVMap

This example builds an input buffer to store data from the channels configured on only one I/O board, the AI-207 board; however, VMAPs usually consist of multiple I/O boards.

```
// Add a board to the VMap: sets ‘1’ VMAP channel for AI-207 as last parameter
//  (a single FIFO on the I/O board holds all data from the physical channels)
DqrtVMapAddChannel(hd, vmapid, devn, DQ_SS0IN, cl, flags, 1);

// Set channel list for device included in the VMap:
//   sets physical channels + TS for the AI-207 board
DqrtVMapSetChannelList(hd, vmapid, devn, DQ_SS0IN, cl, CHANNELS);
```

Note that both `DqrtVMapAddChannel()` and `DqrtVMapSetChannelList()` require a channel size as the last parameter passed.

- `DqrtVMapAddChannel()` requires a VMAP channel list: for boards with one FIFO, this VMAP channel size will be 1 (AI, AO, and DIO boards).
- `DqrtVMapSetChannelList()` requires the number of physical channels to acquire data for.

Refer to the *PowerDNA API Reference Manual* for additional explanation of the difference between VMAP channels and physical channels.

```
// Start VMap. This function puts the IOM into OPS mode and suppresses
// immediate software triggers. Trigger can then be issued via hardware or
// broadcast message
DqrtVMapStartTr(hd, vmapid, FALSE);

// Specify that you want to read up to MAX_SCANS bytes from the device
//  -- act_size is returned from the function and holds the maximum # of samples
//  that can be returned (this may be less than what is requested)
int act_size;
DqrtVMapRqInputDataSz(hd, vmapid, 0, MAX_SCANS*CHANNELS*sizeof(uint32),
&act_size, NULL);
```

4.3.2.2 Set up Synchronization (RtVMap)

After the IOM is put into the Operation (OPS) state, synchronization settings can be set up with the `DqSyncDefineSyncScheme()` API.

`DqSyncDefineSyncScheme()` must be called after the IOM is in OPS. Calling it before the IOM is in the OPS state will result in some hardware settings being reset.

```
// Set the sync_scheme structure in hardware after IOM is in OPS state.
DqSyncDefineSyncScheme(hd, &sync_scheme_1PPS, &status);

// Set up clocks for I/O board: initialize clock structure
DQ_SYNC_DEF_CLOCKS defclocks =
    {DQ_SYNCCLLK_SYNC2,    // clocks supplied from SYNC line 2
    8,                     // board sample rate is clock rate on SYNC2 ÷ 8
    -1,                    // use default group delay
    0};                    // set flags to 0

// Set up board-specific synchronization parameters:
// - Set up clocks on device (devn set to slot position the AI-207
//   is installed in the chassis)
DqSyncDefineLayerClock(hd, devn, &defclocks);

// - Set up triggers on device (trigger source found on SYNC line 3)
DqSyncDefineLayerTrigger(hd, devn, DQ_SYNCTRG_SYNC3, 0);

// - Set up timestamps on device (timestamps incremented by
//   sample rate clock on SYNC line 2)
DqSyncDefineLayerTimestamp(hd, devn, DQ_SYNCTST_SYNC2, 0);
```

4.3.3 Verify ADPLL Status (RtVMap)

Before triggering data collection to start, you can optionally check the status of the ADPLL. Once the ADPLL is trained on the raw 1PPS, CPU-generated sample rate clocks will be synchronized to each other for all I/O boards on all configured chassis.

The ADPLL validation flags are read using the `DqSyncGetSyncStatus()` API.

```
// Check status registers to verify status states
int passing_validation; // holds validation status for user-determined checks
DQ_SYNC_STATUS astatus;

passing_validation = FALSE;
DqSyncGetSyncStatus(hd, 0, &astatus);

// Bits 2 & 1 of the astatus.adpll_sts.status register provide validation
// status of the ADPLL 1PPS reference. '1' is passing validation
if ( (astatus.adpll_sts.status & 4) && (astatus.adpll_sts.status & 2) )
    passing_validation = TRUE;
```

NOTE: `DqSyncGetSyncStatus()` returns status information from several status registers. Refer to the *PowerDNA API Reference Manual* for bit descriptions of each register.

4.3.4 Arm Trigger & Reset Timestamp (RtVMap)

In this example, we are arming the IOM with a broadcast message, which causes IOM hardware to issue a trigger to all configured boards upon the rising edge of the next 1PPS pulse. We also reset the timestamp on all chassis to a known value (0) for timestamp alignment.

```
// Arm command to trigger on the next 1PPS
DqSyncTrigOnNextPPSBrCast(hd, // handle to IOM
                              1, // number of IOM broadcasting to
                              0, // <Reserved>, set to 0
                              &hd); // list of handles of IOMs to trigger

// issue a broadcast command to reset timestamp to 0
DqCmdResetTimestampBrCast(hd, 0);
```

4.3.5 Send / Receive Messages (RtVMap)

After synchronization is setup and triggers are armed, we can begin acquisition. Calling `DqrtVMapRefresh()` initiates a packet transfer between the host and IOM and updates the data on the host side with the newly acquired data from the I/O board's FIFO.

```
// Host exchanges packets with the chassis. In the case of the AI-207,
// this function reads data from the FIFO
DqrtVMapRefresh(hd, vmapid, 0);

// Data is unpacked and parsed at the host side, and the A/D data
// that was acquired is copied into the bdata array

// declare bdata
uint32 bdata[MAX_SCANS];

// extract received data from packet: data_size is number of bytes actually read
DqrtVMapGetInputData(hd, vmapid, 0, MAX_SCANS*CHANNELS*sizeof(uint32),
&data_size, &avl_size, (uint8*)bdata);

// data is pulled out of buffer and converted to floating point data
FILE* fo = NULL; // output file for holding data

for (n = 0; (uint32)n < (data_size/sizeof(uint32))/CHANNELS; n++) {
    for (ch = 0; ch < CHANNELS; ch++) {
           bdata[n*CHANNELS+ch] = ntohl(bdata[n*CHANNELS+ch]);
           if (cl[ch] == DQ_LNCL_TIMESTAMP) {
                  printf(fo, "%d", (bdata[n*CHANNELS+ch]));
           } else {
                  DqAdvRawToScaleValue(hd, devn, cl[ch],
(bdata[n*CHANNELS+ch]), &fdata[n*CHANNELS+ch]);
                  fprintf(fo, "%.6f,", fdata[n*CHANNELS+ch]);
           }
    }
    fprintf(fo, "\n");}
```

4.3.6 Stop Cleanly (RtVMap)

The following API stops operation and disables the boards cleanly.

```
//Stop the devices and free all resources:
DqrtVMapStop(hd, vmapid);

DqrtVMapClose(hd, vmapid);
```

4.4 Example Code for Synchro-nization in ACB Mode

This section provides an overview of how to set up and synchronize a UEI chassis to an external 1PPS reference when acquiring data in ACB mode.

The following code snippets are provided:

- Initialization
  - Initializing the chassis, I/O boards, synchronization scheme, and ACB data transfer mode
- Configuration
  - Clearing any existing configuration from previous runs
  - Configuring channels and ACB transfers
  - Configuring synchronization on each I/O board
- Verification (optional)
  - Verifying 1PPS validation reading ADPLL status
- Arm Trigger & Reset Timestamp
  - Sending broadcast commands to all configured IOMs
- Operation: Send / Receive Messages
  - Sending and receiving messages in ACB mode
- Stop Cleanly
  - Disabling boards cleanly

4.4.1 Initialization (ACB)

Synchronization and data acquisition settings are initialized first.

4.4.1.1 Sync Initialization (ACB)

Each chassis is initialized for 1PPS synchronization in the DQ_SYNC_SCHEME structure. In UEI-provided sample code, this structure is declared globally and initialized in the declaration.

Sync initialization is exactly the same for all of the data acquisition modes. The code snippet in the VMAP tutorial (Section 4.3.1.1) configures the IOM as the 1PPS master; for this example, we'll configure the IOM as a slave to show an alternate example.

The following code snippet configures the sync interface as follows:

- RACK IOM is configured as the 1PPS slave
- I/O board clocks are generated in the Event Module (EM)
- Triggers are armed in software and generated in the IOM

```
DQ_SYNC_SCHEME sync_scheme_1PPS = {
// ==== section A ====================================================
// IOM synchronization
// ADPLL takes 1PPS from sync line 0
     DQ_SYNC_8347,      // sync_device: RACK (8347 CPU) IOM being synched
     DQ_SYNCCLK_SYNCIN0, // sync_source: raw nPPS routed in through
                         //    SyncIn0 on external sync connector
     DQ_SYNCCLK_SYNC0,   // sync_line: raw 1PPS routed from sync connector to
                         //    internal SYNC0 bus line
```

```
    DQ_SYNCCLK_SYNC,     // sync_mode: synchronization mode is 1PPS
    1,                   // nPPS: raw nPPS input is one pulse per second
    100,                 // nPPS_us: Expected accuracy of the raw nPPS clock
                         //    is 100 us, pulses outside of the range
                         //    will be ignored

// ==== section B ==================================================
// Synchronization master configuration (synchronization output)
    0,                 // sync_server: slave IOM receives 1PPS pulse externally
    0,                 // srv_param: n/a since this is a slave IOM
    0,                 // trig_server: <Reserved>, set to 0.

// ==== section C ==================================================
// Clock configuration
    {0, 0, DQ_CLOCKSRC_EM0, 0},   // clock_src[]: PPS synched EM clock routed
                                  //  on internal SYNC2 bus line to I/O boards
    {0, 0, 0, 0},                 // clock_tmr[]: <reserved>
    {0, 0, 2000, 0},              // clock_frq[]: clock frequency = 2 kHz
                                  //  on SYNC2
    {0, 0, 0, 0},                 // clock_div[]: no clock divider for EMx


// ==== section D ==================================================
// Trigger configuration
    DQ_USE_SYNC3,      // trig_source: trigger routed to internal SYNC3 for
                       //    I/O board distribution
    0,                 // trig_line: <Reserved>, set to 0.
    DQ_TRIGSTART_NPPS,// trig_start: trigger will be issued upon next nPPS
                       //    after armed
    250,               // trig_delay: 250 us trigger delay
    0,                 // trig_period_ms: <Reserved>, set to 0.
    0,                 // trig_stop: no trigger stop programmed
    0,                 // trig_stop_src: DQ_TRIGSTOP_NCLOCKS not set,
                       //    no src programmed
    0,                 // trig_duration: DQ_TRIGSTOP_NCLOCKS not set, no
                       //    duration programmed



    // ==== section E ==================================================
    // External Sync or Internal SYNC lines routing configuration

    {0, 0, 0, 0},      // clclk_dest[]: external routing already configured
    0,                 // pps_dest: not routing PPS from sync to external
    0                  // trig_dest: not routing trigger out sync connector
};
```

4.4.1.2 I/O Board Initialization of Config Settings (ACB)

In ACB mode, I/O board configuration is passed to the ACB API that starts operations (`DqAcbInitOps()`). In UEI example code, these parameters are often set up as a global #define, as shown below.

```
#define CFG207(DQ_LN_ENABLED \
          | DQ_LN_ACTIVE \
          | DQ_LN_IRQEN \
          | DQ_LN_STREAMING \
          | DQ_AI208_MODEFIFO \
          | DQ_LN_RAW32 \
          | DQ_LN_CLCKSRC1 \   // use SYNC line for clock
          | DQ_LN_STRIGEDGE0)  // wait for digital trigger
```

Refer to the *PowerDNA API Reference Manual* for detailed descriptions of the configuration settings, or refer to ACB example code for specific configuration settings for each type of I/O board.

4.4.1.3 Initialize IOM (ACB)

You will see a `DqInitDAQLib()` function as the first API call in C examples. This function allocates resources and data structures for UEI's DAQLib libraries. The `DqInitDAQLib()` is not required when programming in a Windows environment; libraries are initialized automatically when loading the DLL on Windows systems. It is required when programming in Linux.

The high-level API is based on the use of the DQEngine, a programming environment that takes care of handling streams of data and that performs error correction. The DQEngine must be started explicitly for ACB mode:

```
pDQE pDqe = NULL;

// Start engine
DqStartDQEngine(1000*1,     // main clock period
                &pDqe,      // pointer init parameters (timeouts, retries)
                NULL);      // NULL uses default pDqe (recommended)
```

To initiate communication, you must first get a DAQLib handle for each IOM by calling DqOpenIOM():

```
// Connect with all IOM and obtain library handle(s) for the connection(s)
DqOpenIOM(iom_ip,           // chassis IP address
          DQ_UDP_DAQ_PORT,  // host PC UDP port used communication
          1000,             // timeout duration in milliseconds
          &hd,              // pointer to the IOM handle
          &DQRdCfg);        // pointer to store echoed device results
                            //   NULL if not required
```

4.4.1.4 Initialize I/O Board Channels (ACB)

The following code snippet sets up board-specific initialization, (i.e., which channels are enabled, what gain is required, specific modes required), for the AI-207 analog input board:

```
int n; // temporary index
int CL[CHANNELS]; // CHANNELS is #defined as the total # enabled + Timestamp
int flags[CHANNELS]; // flags will receive the status of FIFO

// Set up channel list
for (n = 0; n < CHANNELS; n++) {
    CL[n] = n | DQ_LNCL_DIFF; // enables differential input mode
}

// allocate a channel as the timestamp: last channel in the list is timestamp
CL[CHANNELS-1] = DQ_LNCL_TIMESTAMP;
```

4.4.1.5 Initialize ACB

Each new advanced circular buffer (ACB) will have a buffer control block (BCB) structure allocated to it.

```
// DqAcbCreate allocates a new buffer and links it with DQEngine and IOM
pDQBCB bcb = {NULL};
DqAcbCreate(    pDqe,       // pointer to previously created DQE instance
                hd,         // handle to IOM
                devices,    // I/O board ACB is allocated to
                DQ_SS0IN,   // subsystem: #define specifying inputs for AI-207
                &bcb);      // newly allocated BCB structure
```

4.4.2 Configuration (ACB)

Configuration consists of setting up the ACB buffer with a channel specific information, setting up events, starting operations, and configuring the synchronization settings on each I/O board.

4.4.2.1 Configure ACB

Set up parameters in an ACB structure and then set it using the `DqAcbInitOps()` API to configure the AI-207 board.

ACB parameters are configured in a structure of type `DQACBCFG`:

`DQACBCFG` Structure Element	Description
uint32 samplesz	raw sample , bytes
uint32 scansz	scan , samples
uint32 framesize	number of scans in the frame, max
uint32 frames	frames in the buffer
uint32 ppevent	packets per DQ_ePacketDone event
uint32 mode	mode of operations: Single,Cycle,Recycled,error handling
uint32 dirflags	transfer direction and additional flags
uint32 maxpkt	how much data to accumulate in the packet before sending (0=default)
uint32 hwbuf	how much data to keep on the cube (0 = default)
uint32 hostringsz	number of packets in the host ring buffer (0 = default)
uint32 wtrmark	percent of the ring buffer queue packets kept in case IOM reports an error

Table 4-1 `DQACBCFG` Structure

```
DQACBCFG acb;  //declare acb

// initialize acb parameters as listed in above table: See example code
//  for recommended values for each DQACBCFG parameter

// Set structure with API
DqAcbInitOps(   bcb,       //set up in DqAcbCreate (see Section 4.4.1.5)
                &Config,   // Config is #define CFG207 (see Section 4.4.1.4)
                0,         // Trig,
                NULL,      // pDQSETTRIG TrigMode,
                &fCLClk,   // 207 uses CLClk, which is generally used for
                           // boards with multiplexed inputs
                           // CVClk is generally used for boards with
                           // simultaneously sampled inputs
                &fCVClk,   // In this example, fCVClk = fCLClk = CLCLOCK =1k
                &CL,       // CL is set to the number of channels enabled
                CL,        // array of channel configuration (Section 4.4.1.4)
                0,         // uint32* ScanBlock,
                &acb);     // acb structure defined using Table 4-1 parameters
```

4.4.2.1.1 Set Up ACB Buffer Variable

The acb circular buffer consists of a number of frames (acb.frames). Each frame consists of a number of scans (acb.framesize). Each scan consists of a number of samples (acb.scansz). The structure DQly205_float holds a uint32 raw piece of data and a float data value. DQly205_float represents a single sample of data.

The following code allocates enough space for the full circular buffer:

```
bufsize = acb.framesize * acb.frames; // bufsize is in scans
data = malloc(bufsize * sizeof(DQly205_float) * acb.scansz);
```

4.4.2.2 Set Up Events (ACB)

Events must be set up to tell the application when to read data from or service the buffer. When acquisition starts, DQE stores data into the circular buffer at the head of the buffer while the application generally reads data from the last frame of the buffer, or tail. Both operations occur asynchronously and are synchronized by triggering a DQE event.

Whenever incoming data crosses a frame boundary, DQE sends an event to the application, which in turn will read the existing data stored in the tail of the circular buffer.

The following API sets up events to do this:

```
DqeSetEvent(bcb,
DQ_eFrameDone|DQ_ePacketLost|DQ_eBufferError|DQ_ePacketOOB|DQ_eBufferDone);
```

Where an interrupt will be generated on any of the following bcb event control flags:

DQ_eFrameDone: One or more frames are filled with data

DQ_ePacketLost: Error is unrecoverable, one or more packets are lost

DQ_eBufferError: Overrun/under-run event (DQACB buffer only)

DQ_ePacketOOB: Packet is out of bounds

DQ_eBufferDone: Buffer is completed (straight buffer only)

4.4.2.3 Starting ACB Operations

ACB operations are started by the DqeEnable() API.

```
DqeEnable(  TRUE, // True to enable operations
            bcb,  // pointer to bcb (or array of bcbs)
            1,    // BcbNum:  # of bcb array
            FALSE); // True to start simultaneously with
                    // software trigger
```

4.4.2.4 Set up Synchronization (ACB)

After the IOM is put into the Operation (OPS) state, synchronization settings can be set up by calling the `DqSyncDefineSyncScheme()` API. `DqSyncDefineSyncScheme` must be called after the IOM is in OPS; calling it before will result in some hardware settings being reset.

```
// Set the sync_scheme structure in hardware after IOM is in OPS state.
DqSyncDefineSyncScheme(hd, &sync_scheme_1PPS, &status);

// Set up clocks for I/O board: initialize clock structure
DQ_SYNC_DEF_CLOCKS defclocks =
    {DQ_SYNCCLLK_SYNC2,      // clocks supplied from SYNC line 2
    8,                       // sync line 2 clock frequency is divided by 8
    -1,                      // use default group delay
    0};                      // no special mode flags

// Set up board-specific synchronization parameters:
// - Set up clocks on device (devn identifies which slot position the AI-207
//   is installed in the chassis)
DqSyncDefineLayerClock(hd, devn, &defclocks);

// - Set up triggers on device (trigger source found on SYNC line 3)
DqSyncDefineLayerTrigger(hd, devn, DQ_SYNCTRG_SYNC3, 0);

// - Set up timestamps on device (timestamps incremented by
//   sample rate clock on SYNC line 2)
DqSyncDefineLayerTimestamp(hd, devn, DQ_SYNCTST_SYNC2, 0);
```

4.4.3 Verify ADPLL Status (ACB)

Before triggering data collection to start, you can optionally check the status of the ADPLL. Once the ADPLL is trained on the raw 1PPS, CPU-generated sample rate clocks will be synchronized to each other for all I/O boards on all configured chassis.

The ADPLL validation flags are read using the `DqSyncGetSyncStatus()` API.

```
// Check status registers to verify status states
int passing_validation; // holds validation status for user-determined checks
DQ_SYNC_STATUS astatus;

passing_validation = FALSE;
DqSyncGetSyncStatus(hd, 0, &astatus);

// Bits 2 & 1 of the astatus.adpll_sts.status register provide validation
// status of the ADPLL 1PPS reference. '1' is passing validation
if ( ( astatus.adpll_sts.status & 4) && (astatus.adpll_sts.status & 2 ) )
    passing_validation = TRUE;
```

NOTE: `DqSyncGetSyncStatus()` returns status information from several status registers. Refer to the *PowerDNA API Reference Manual* for bit descriptions of each register.

4.4.4 Reset Timestamp and Arm Trigger (ACB)

The following code snippet arms the IOM, which causes IOM hardware to issue a trigger to all configured boards upon the rising edge of the next 1PPS pulse. We also reset the timestamp on all chassis to a known value (0) for timestamp alignment.

```
// issue a broadcast command to reset timestamp to 0
DqCmdResetTimestampBrCast(hd, 1, 0, &hd);

// issue a broadcast command to all identified IOM
// to trigger on the next 1PPS
DqSyncTrigOnNextPPS(hd, 0);
```

4.4.5 Send / Receive Messages (ACB)

After synchronization is setup and triggers are armed, the `DqeWaitForEvent()` API waits for an event. Once an event occurs, we determine which events caused the trigger by calling the `DqeGetEvent()` API. If the event was caused because a frame completion, samples are acquired with `DqAcbGetScansCopy()`, which populates the sample values in the `data` buffer created in Section 4.4.2.1.1.

```
DqeWaitForEvent(bcb,        // pointer to bcb (or array of bcbs)
                1,          // BcbNum:  # of bcb array
                FALSE,      // wait mode
                EVENT_TIMEOUT, // wait timeout
                NULL);      // buffer for event flags

uint32 events;  // buffer for event flags associated with bcb

DqeGetEvent(bcb, &events);

// maximum errors allowed at a time #defined as 3 in this example
int errorsallowed = RETRY_ATTEMPTS;

//check reason for event
if (events & (DQ_ePacketLost|DQ_eBufferError|DQ_ePacketOOB)) {
    printf("Device %d: error event 0x%x\n", devices, events);

    // service error events as you want; you may want to retry
    errorsallowed--;
}

uint32 i, size, avail, minrq;
// if the event was caused by a frame crossing, pull data out of ACB tail
if (events & DQ_eFrameDone) {
    errorsallowed = RETRY_ATTEMPTS;
    minrq = acb.frame;
    avail = minrq;
    while (TRUE) {
          DqAcbGetScansCopy(bcb, data, acb.framesize, acb.framesize,
                                         &size, &avail);
    }
```

```
        // AI-207 input data is retrieved and available in the data buffer
        // the following parses data for each channel's sample and prints it
        samples[n] += CHANNELS;
        for (i = 0; i < CHANNELS; i++) {
              if ((i % CHANNELS) == (CHANNELS - 1)) {
                    fprintf(fp[n], "%f\t", *((float*)data[n] + i));
                    fprintf(fp[n], "\n");
        }
        else {fprintf(fp[n], "%f\t", *((float*)data[n] + i));}
    }
    if (events & DQ_eBufferDone) {
        // Occurs when DQACB mode is Single and end of buffer is reached
        break;
    }
```

4.4.6 Stop Cleanly (ACB)

The following API stops operation and disables the boards cleanly.

```
// Stop operations; number of devices is 1 because this example is
// only for one device, AI-207
DqeEnable(FALSE, bcb, 1, FALSE);

DqAcbDestroy(bcb);    // destroy bcb buffer
free(data);           // free malloc'ed data

// close IOM communications
DqCloseIOM(hd);

// stop DQE engine
if (pDqe) {
      DqStopDQEngine(pDqe);
}

// allow library to release resources and clean up allocated structures
DqCleanUpDAQLib();
```

Appendix A

The following cables and STP panels are available for the 1PPS Sync Interface boards:

- DNA-CBL-SYNC-10/R3 Cable & Schematic (Section A.1)
- DNA-CBL-SYNC-RJ-1G/R3 Cable Schematic (Section A.2)
- DNA-STP-SYNC-1G STP Panel (Section A.3)

A.1 DNA-CBL-SYNC-10/R3 Cable & Schematic

For a two-chassis system, UEI offers sync cables (DNA-CBL-SYNC-10/R3), which are 30-inch 8-conductor cables that have the 10-pin sync connectors on both ends. The R3 version of the cable can be used with PPCx cubes and also with PPCx-1G cubes and racks. (The R2 version can only be used with PPCx cubes.) See **Figure A-1** below:

Figure A-1 Photo of DNA-CBL-SYNC-10/R3 Cable

For users who want to fabricate their own cables, the schematic for the DNA-CBL-SYNC-10/R3 cable is shown in **Figure A-2**.

© Copyright 2018
United Electronic Industries, Inc.

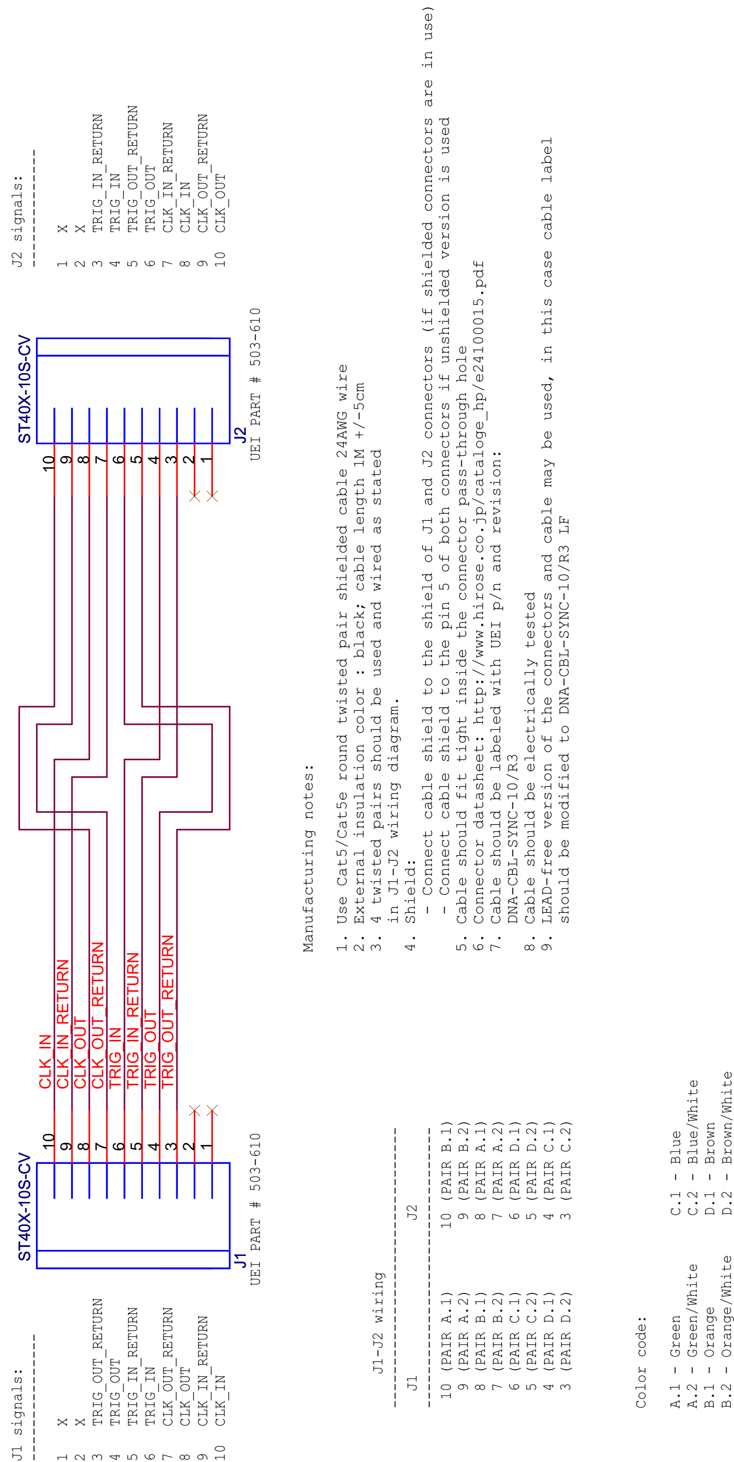


Figure A-2 Schematic of DNA-CBL-SYNC-10/R3 Cable

A.2 DNA-CBL-SYNC-RJ-1G/R3 Cable Schematic

The DNA-CBL-SYNC-RJ-1G cable is a synchronization cable with a male RJ Ethernet connector at one end and a 10-pin sync connector at the other.

See **Figure A-3** below:

Figure A-3 Photo of DNA-CBL-SYNC-RJ-1G Cable

The maximum cable length tested was 800 feet.

For users who want to fabricate their own cables, the schematic for the DNA-CBL-SYNC-RJ-1G/R3 cable is shown in **Figure A-4** below.

NOTE: Note that for sync connections, a cat5e or better cable is required.

© Copyright 2018
United Electronic Industries, Inc.

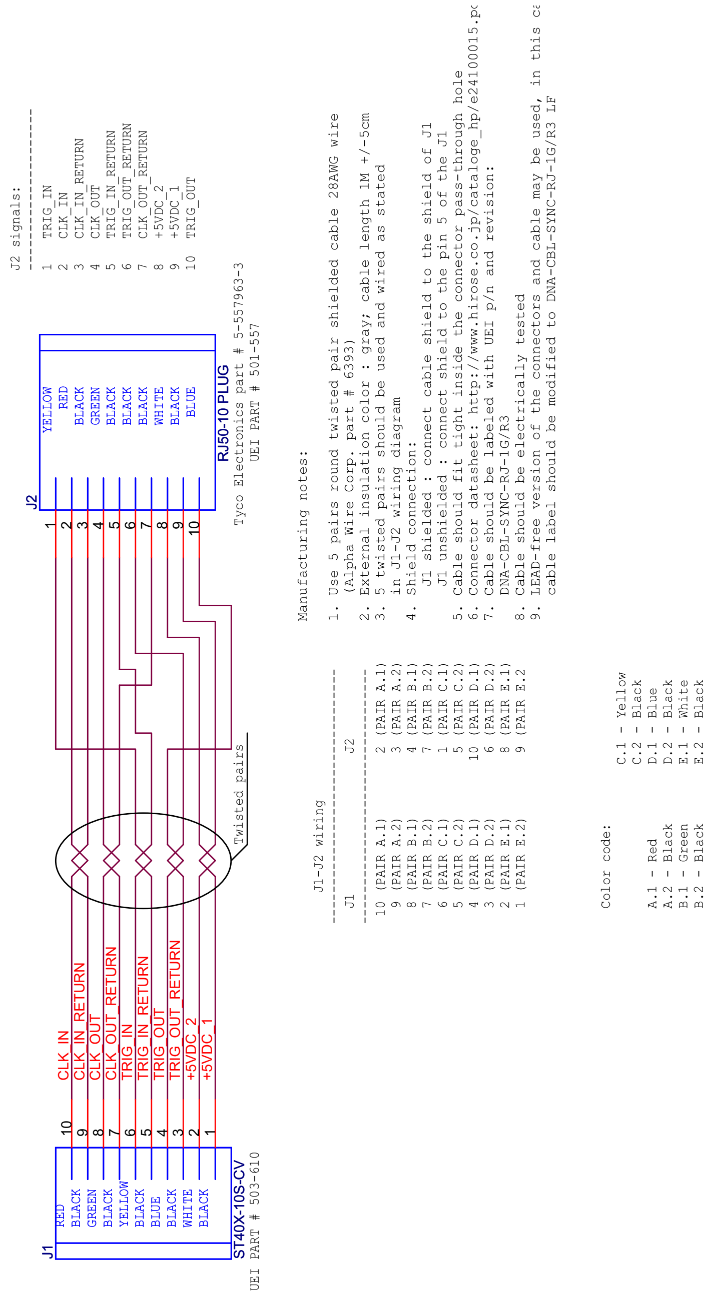


Figure A-4 Schematic of DNA-CBL-SYNC-10/R3 Cable

A.3 DNA-STP-SYNC-1G STP Panel

The UEI DNA-STP-SYNC-1G STP board provides a simple method to connect a single master to serve up to 6 slave ports on the STP. Additional slaves can be added to the system by daisy-chaining STP boards together.

For more information, refer to the UEI DNx-STP-SYNC-1G Synchronization and Screw Terminal Panel User Manual, which is available for download from www.ueidaq.com. The manual also includes schematics for fabricating your own interconnection cables, if desired.

© Copyright 2018
United Electronic Industries, Inc.

Index

Numerics
1PPS synchronization 1, 35

A
ACB 85

C
Cable(s) 102, 104
Clock and Trigger Routing 20
Clock programming 36
Conventions 2

D
Data acquisition mode 85

E
External Sync Connections 7, 9

G
Getting Started with the SYNC API 21

I
Internal Sync Connections 15

O
Organization 1, 33, 34

P
Pinout 8
Programming the Sync Interface 21

S
Schematic 15
Screw-terminal panels 102
STP panel 102
Support 2
Synchronize Multiple Chassis 9, 10, 11, 12, 13, 14

T
Timestamp Reference 38
Trigger 37, 38

V
VMAP 85