United Electronic Industries

The High-Performance Alternative

DNx-I2C-534

—

User Manual

4-port, I^2C Interface Board
for the PowerDNA Cube and RACK Series Chassis

April 2020

PN Man-DNx-I2C-534

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/

CE

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 a member of 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

List of Figures

Chapter 1 Introduction

This document outlines the feature set and use of the DNx-I2C-534, an ultra secure 4-port interface board for serial applications using the I^2C protocol, as per UM10204 specification.

The following sections are provided in this chapter:

- Organization of this Manual (Section 1.1)
- I2C-534 Board Overview (Section 1.2)
- Features (Section 1.3)
- Specification (Section 1.4)
- Indicators (Section 1.5)
- Device Description (Section 1.6)
- I^2C Master Description (Section 1.7)
- I^2C Slave / Bus Monitor Description (Section 1.8)
- I2C-534 CRC Checker & Status Reporting (Section 1.9)
- Termination Pull-up Resistors (Section 1.10)
- Wiring & Connectors (pinout) (Section 1.11)

1.1 Organization of this Manual

This *DNx-I2C-534 User Manual* is organized as follows:

- **Introduction**
  Chapter 1 provides an overview of DNx-I2C-534 features, device architecture, connectivity, and logic.
- **Programming with the High-Level API**
  Chapter 2 provides an overview of the how to create a session, configure the session, and interpret results with the high-level framework API.
- **Programming with the Low-Level API**
  Chapter 3 is an overview of low-level API commands for configuring and using the I2C-534 series board.
- **Appendix A - Accessories**
  This appendix provides a list of accessories available for use with the DNx-I2C-534 board.
- **Index**
  The index provides an alphabetical listing of the topics covered in this manual.

NOTE: 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.

No HOT SWAP

Always turn POWER OFF before performing maintenance on a UEI system. Failure to observe this warning may result in damage to the equipment and possible injury to personnel.

Usage of Terms

Throughout this manual, the term “Cube” refers to either a PowerDNA Cube product or to a PowerDNR RACKtangle™ 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.

1.2 I2C-534 Board Overview

The DNx-I2C-534 boards are 4-port I^2C interface boards.

DNA-I2C-534, DNR-I2C-534, and DNF-I2C-534 boards are compatible with the Cube, RACKtangle, and FLATRACK chassis respectively. These board versions are electronically identical and differ only in mounting hardware. The DNA version is designed to stack in a Cube chassis. The DNR/F versions are designed to plug into the backplane of a RACK chassis.

1.2.1 I^2C Interface

The I^2C interface is compliant with the UM10204 standard. Each of the four ports include both a master and slave port.

Users can program DNx-I2C-534 inputs and outputs to use 3.3 V or 5 V TTL levels, and the baud rate is configurable to run at Standard-Mode (100 kbit/s), Fast-Mode (400 kbit/s), Fast-Mode+ (1 Mbit/s) rates as well as custom clock rates from 2kHz to 100kHz.

The SCL and SDA pins include 4.99 kΩ pull-up resistors; additionally, each port can be configurable to relay in or out an additional on-board 1.5 kΩ resistor in parallel, or users can add resistors externally to better align with RC requirements of your system.

The standard I^2C transaction includes sending a packet and then receiving an acknowledge. Failure to receive an ACK is an error condition. However, some I^2C devices do not respond with an ACK. The DNx-I2C-534 can be set in a mode where it does not wait for an ACK, and does not generate an error message if the standard ACK is not received.

1.2.2 Guardian Diagnostic Support

The DNx-I2C-534 board is part of UEI's Guardian series, providing additional diagnostic capabilities. The master of each physical port can be connected to its own slave port in bus monitor mode via on-board switches. This allows the user software to confirm the correct data has been sent out from each master.

1.2.3 CRC-ensured Data Transfers

To ensure data security and reliability from the host PC and/or embedded CPU, a CRC checksum is added to each I^2C transaction written. This checksum remains with the data as it moves through the system and is checked by the DNx-I2C-534 FPGA prior to it being written to the output drivers.

On data reads, the chassis CPU adds a CRC checksum to the data and this is confirmed by the host CPU prior to presenting the data to the application software. Note the FPGA is the last device in the data chain and reads and writes directly to the I^2C receiver/transmitter.

1.2.4 Accessories

For ease of connection UEI offers the DNA-CBL-COM which brings the four I^2C ports out to 9-pin dSub connectors. UEI also offers the DNA-CBL-I2CM-3M that brings each of the four I^2C ports out to easy to use RJ-10 connectors. Optionally the DNA-CBL-37S series brings the four I^2C ports out to a 37-pin D-Sub connector.

1.2.5 Software Support

The DNx-I2C-534 includes all software required for operation in both UEIPAC and PowerDNA chassis deployments. There are no license or royalty payments ever required and software revision updates are always available on the UEI web site at no charge.

Software included with the DNx-I2C-534 provides a comprehensive yet easy to use API that supports all popular operating systems including Windows, Linux, real-time operating systems such as QNX, RTX, VxWorks and more.

1.3 Features

A summary of features of the I2C-534 I^2C interface is provided below:

- Up to 4 independent I^2C interfaces can be used simultaneously: each interface offers a master and a slave port
- Fully conforms to UM10204 at Standard-Mode (SM), Fast-Mode (FM) and Fast-Mode+ (FM+) bit rates.
- Additionally, a “custom” bit rate mode is available for non-standard devices and supports bit rates from 2kHz to 100kHz.
- Guardian read-back of master transmissions can be used to confirm validity of transmit data
- Full data path integrity confirmed with CRC
- Standard D-Sub 37 connectivity
- Includes all software including C source code
- No royalties or license required

1.4 Specification

Technical specifications for the DNx-I2C-534 board are listed in **Table 1-1**.

Table 1-1 DNx-I2C-534 Technical Specifications

General Serial Specifications	
Number ports	4, each provides Master/Slave/Bus monitor capability
Serial Interfaces	I^2C, complies with UM10204 specification
Maximum SCL speed:	1 Mbit/s (compliant with I^2C SM: 100kb, FM: 400 kb and FM+: 1 Mb)
Logic Level	5 V / 3.3 V compatible
Protection	350 V port-to-port; 15 kV ESD protection
Baud rate clock resolution	15.15 ns
FIFO Size	Master Mode: 1k / 1k input / output Slave Mode: 512 / 512 input / output
General and Environmental	
Isolation	350 Vrms port-to-port and port-to-chassis
Power Consumption	< 4 W
Operating Temp. (tested)	-40 °C to +85 °C
Operating Humidity	95%, non-condensing
Vibration *IEC 60068-2-6* *IEC 60068-2-64*	5 g, 10-500 Hz, sinusoidal 5 g (rms), 10-500 Hz, broad-band random
Shock *IEC 60068-2-27*	100 g, 3 ms half sine, 18 shocks @ 6 orientations 30 g, 11 ms half sine, 18 shocks @ 6 orientations
MTBF	350,000 hours

1.5 Indicators

The DNx-I2C-534 indicators are described in **Table 1-2** and illustrated in **Figure 1-1**.

Table 1-2 I2C-534 Indicators

LED Name	Description
RDY	Indicates board is powered up and operational
STS	Indicates which mode the board is running in: • **OFF**: Configuration mode, (e.g., configuring ports, running in point-by-point mode) • **ON**: Operation mode

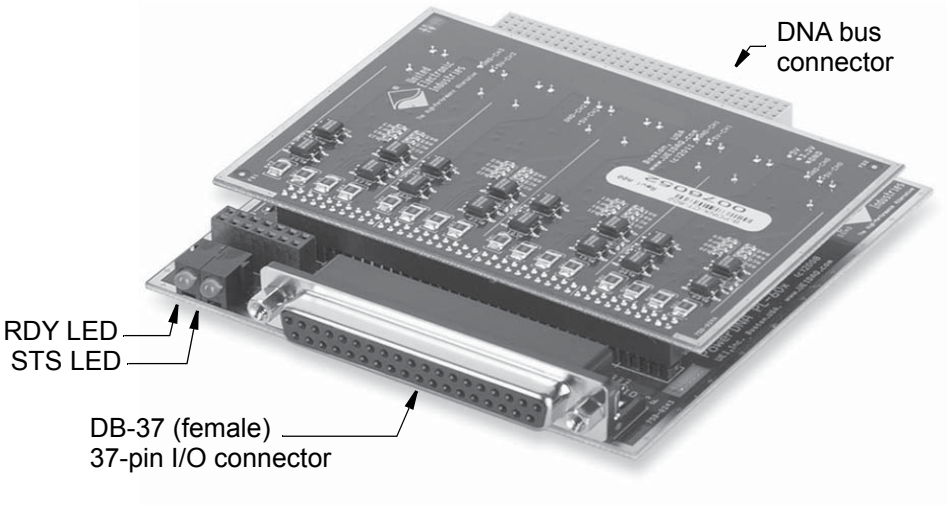


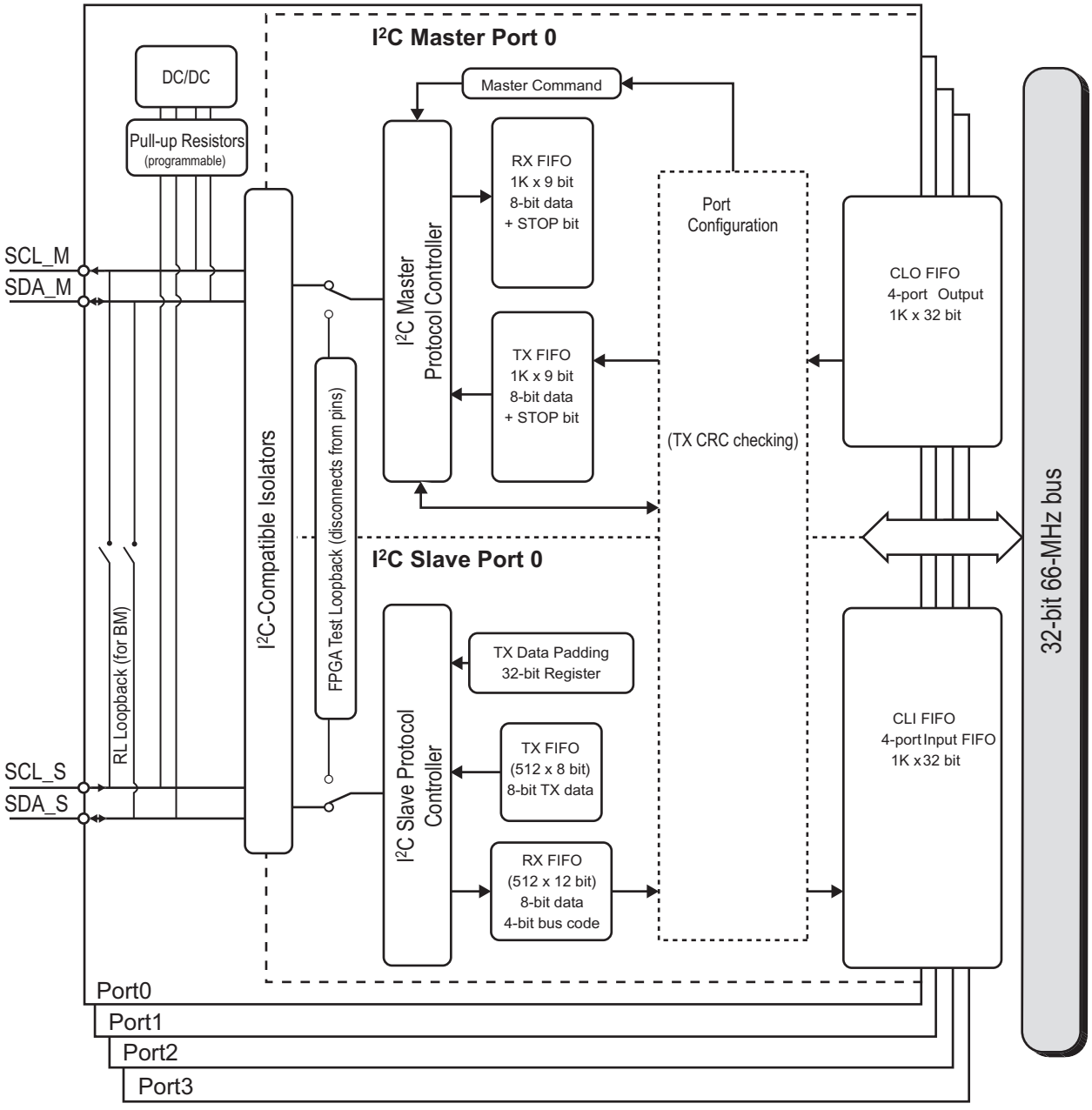
Figure 1-1 Photo of DNA-I2C-534 I^2C Board

1.6 Device Description

Each DNx-I2C-534 port is designed to meet the I²C protocol, as per UM10204 specification. The I2C-534 board offers 4 isolated, independent I²C ports, which support both master and slave capabilities.

Each master and slave have a serial clock line (SCL) and serial data line (SDA), which are used to communicate with all I²C devices in a system that are connected to the I²C SCL / SDA bus.

Figure 1-2 shows a block diagram of the I2C-534. Refer to Section 1.7 through Section 1.10 for descriptions and Section 1.11 for pinout.



Ports are grouped into 4 isolated blocks of I²C master and I²C slave pairings.

Figure 1-2 Block Diagram of I2C-534

1.7 I²C Master Description

Each DNx-I2C-534 port includes an I²C master. I²C masters control the physical I²C bus; masters start and stop a transfer and generate the clock signals on the SCL pin. Each master includes a transmitter that sends data onto the SDA line, and a receiver that receives data from the SDA line.

The I2C-534 master on each DNx-I2C-534 port is designed with a 1K-word RX FIFO and a 1K-word TX FIFO for storing data words to receive from or send to the I²C bus.

1.7.1 I²C Transactions

All transactions begin with a START command and end with a STOP command, which are always issued by an I²C master.

A typical transaction executes as follows:

- The master initiates a data transfer on the bus with a START command (a HIGH to LOW transition on the SDA line while SCL is HIGH).
- The master generates clock pulses that will clock the data through for the transaction.
- The master issues a 7- or 10-bit address to address a specific slave.
- The master issues a R/W bit to indicate whether the slave is commanded to write data or read data.

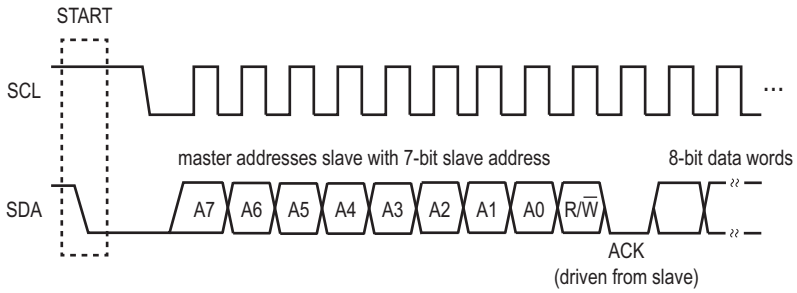


Figure 1-3 Example of I²C Master Commanding a Transfer (7-bit Address)

- The addressed slave issues an ACK to acknowledge the request (the slave pulls the SDA line low).
- If the command was a READ, the slave serially transmits 8-bit data words to the master (MSB first), the data is acknowledged, and the master stores the words in its local master RX FIFO.
- If the command was a WRITE, the master serially transmits 8-bit data words (MSB first) to the slave, the data is acknowledged.
- When the transfer is complete, the master issues a STOP command (a LOW to HIGH transition on the SDA line while SCL is HIGH).

1.7.2 Master Commands

Users configure commands, addresses, and data via UEI API.

The commands include START, STOP, ReSTART, WRITE, READ, and special commands designed specifically for a Renesas XDCP™ protocol device:

- STOP: issue a stop condition once bus is available
- START+WRITE: issue a start and a write, including write multiple
- START+READ: issue a start and a read, including read multiple
- START+WRITE+RESTART+READ: issue a start and a write, and then restart to keep the bus and read (for back-to-back write then read without bus going in to idle state)
  Described in Section 1.7.2.1.
- START+WRITE+READ: issue a start and a write, and then a read (for Renesas XDCP™ protocol, e.g. X9119)
  Described in Section 1.7.2.1.
- START+WRITE+WRITE: issue a start and a write, and then a restart and another write (special for Renesas XDCP™ protocol, e.g. X9259)
  Described in Section 1.7.2.1.

1.7.2.1 Custom Master Command Sequences

UEI provides built-in master command sequences in support of the Maxim potentiometer (e.g. DS3930 potentiometer) and in support of the Renesas XDCP™ protocol, (e.g. X9119 and X9259 potentiometers) that extend beyond the typical I^2C START, STOP, ReSTART, WRITE, and READ.

1.7.2.1.1 START + WRITE + RESTART + READ

UEI provides a START+WRITE+RESTART+READ command sequence in support of devices that require a back to back write then read, (e.g. Maxim DS3930). The single byte read sequence transfers as follows:

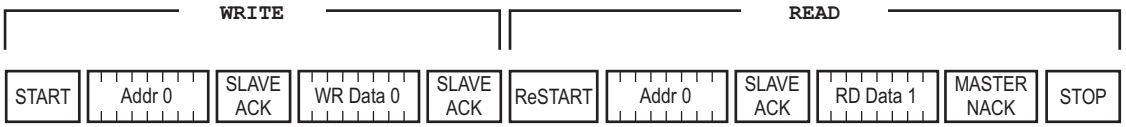


Figure 1-4 Built-in START+WRITE+ReSTART+READ Command

1.7.2.1.2 START + WRITE + READ

UEI provides a START+WRITE+READ command sequence in support of devices that require an instruction opcode write immediately followed by a read without a ReSTART in between, (e.g. Renesas X9119).

For the X9119 example, the sequence to Read Wiper Counter Register (WCR) is as follows:

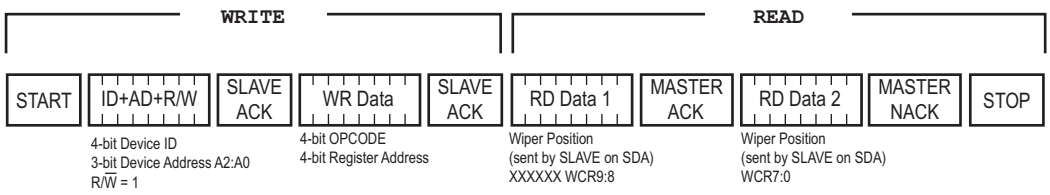


Figure 1-5 Built-in START+WRITE+READ Command

1.7.2.1.3 START + WRITE + WRITE

UEI provides a START+WRITE+WRITE command sequence in support of devices that require a 4-bit Device Type Identifier, 4-bit Address and no WR/RD bit, (e.g. Renesas X9259).

For the X9259 example, the sequence to Read Wiper Counter Register (WCR) is as follows

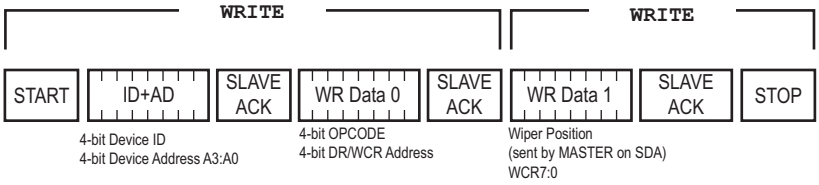


Figure 1-6 Built-in START+WRITE+WRITE Command

1.7.3 Clock Stretching

The I2C-534 supports stretching of the clock (as defined in the UM10204 specification). Clock stretching is a procedure used by the slave to delay the next byte of data from transferring immediately. Though the master controls the transaction, after a byte transfer, the slave has the capability of forcing the master into a wait state by holding the SCL line LOW until it is ready for another byte of data.

1.7.4 Multi-Master Mode, Arbitration, and Synchro-nization

I2C-534 masters can be configured to support Multi-master mode.

Multi-master mode is when more than one master can attempt to control the I^2C bus at the same time without corrupting the message.

Masters decide which master will own the bus through Arbitration and Clock Synchronization, as per UM10204 specification.

The clock synchronization procedure synchronizes the clock signals of two or more devices. Once synchronized, the winning master is determined by which master generates SCL clock with its LOW period the longest clock LOW time and its HIGH period shortest clock HIGH time.

The arbitration procedure ensures that if more than one master simultaneously tries to control the bus, only one is allowed to do so and the winning message is not corrupted. This procedure proceeds bit by bit, with masters comparing SDA serial data to verify the data read matches what was sent. Upon a master reading a low bit value when it expected a high, it loses the arbitration.

Refer to the UM10204 specification for detailed descriptions.

1.8 I²C Slave / Bus Monitor Description

Each DNx-I2C-534 port provides a slave device, which can be used as an I²C slave or configured as a Bus Monitor and / or loopback diagnostic tool.

1.8.1 I2C-534 as an I²C Slave

As an I²C slave, the slave on each I2C-534 port is addressed by a master (external or internal by using loopback) and responds to master commands.

I2C-534 slaves can be configured with a 7-bit or 10-bit address using UEI API.

The I²C slave receiver reads master commands and data from the I²C bus, and the slave transmitter sends data in response to a master request, as described in Section 1.7 (refer to **Figure 1-3**).

1.8.1.1 Slave Data Storage

Slave TX storage: Slave data to be transmitted can be stored in a 512 x 8-bit FIFO or a 32-bit TX data padding register.

Upon receiving a WRITE command, the slave transmitter serially transmits words to the I²C bus from the FIFO, or alternatively, the transmitter can be configured to transmit up to 4 8-bit words from the 32-bit TX data padding register (bits 31:24 transmit first, bits 23:17 transmit next, etc.). If the TX FIFO is empty, data in the padding register will be sent.

Slave RX storage: Upon receiving a READ command, the slave receiver stores received data in a 512 x 12-bit slave RX FIFO. Users can also configure RX data from all 4 ports to be stored in a single 1K x 32-bit CLI FIFO.

Refer to Section 1.8.3 and Section 1.8.3.1 for descriptions of the formatting of received slave data and for an example of a diagnostic data transfer.

1.8.2 I2C-534 Slave as a Bus Monitor

The I2C-534 slave can also be configured as a Bus Monitor diagnostic tool.

If configured as a Bus Monitor, the Bus Monitor slave is connected to the I²C bus and reads transactions on the I²C bus regardless of which slave device the master is communicating with.

In this mode, the Bus Monitor slave does not respond to the master; however, as a test capability, users can configure the Bus Monitor slave to acknowledge any address and provide an ACK. This feature allows testing of master software without connecting it to an actual device.

Refer to Section 1.8.3 and Section 1.8.3.1 for descriptions of the formatting of data stored via the slave Bus Monitor and an example of what that diagnostic data looks like.

1.8.3 Slave RX Data Formatting

The slave RX FIFO provides storage for 512 x 12-bit words received. All data received by a slave is stored in this FIFO:

- If a slave is configured as an I²C slave:
  All 8-bit data written to the slave from the master is stored in the RX FIFO. The FIFO can also include additional information about bus conditions on the I²C bus.
- If a slave is configured as a Bus Monitor slave:
  All activity on the I²C bus including data and bus conditions are stored in this FIFO.

The bus condition is stored in the upper 4-bits of the 12-bit word, and the data written to the slave is stored in the lower 8-bits.

Note that users can also configure the slave to only store the data words.

The following bus conditions are provided:

Table 1-3 Master Command Conditions

Value	Name	Description
0	RSV	Reserved
1	START	I^2C Start condition (ignore bits 7..0)
2	RESTART	I^2C Restart condition (ignore bits 7..0)
3	STOP	I^2C Stop condition (ignore bits 7..0)
4	Address + ACK	I^2C Address with ACK received
5	Address + /ACK	I^2C Address with NACK received
6	Data + ACK	I^2C Data with ACK received
7	Data + /ACK	I^2C Data with NACK received
8	All data received + /ACK	I^2C Last data byte + NACK received
9	Clock stretching error	I^2C Error occurred in relation to clock stretching

1.8.3.1 Example of Reading Bus Conditions on the Slave Receiver

This section provides an example showing what data is stored in the RX FIFO after a write transaction. In this example, the slave address is 0x2A, and it was written with 4 pieces of data (0x1, 0x2, 0x3, 0x4).

In this example, the output from running UEI's low-level sample code that reads the RX FIFO after a master write is formatted as follows:

```
Master: transmitted=4 available=508 crc_stat=0x0
SlaveRx: received=8 available=0
[0]=300 [1]=100 [2]=454 [3]=601 [4]=602 [5]=603 [6]=804
[7]=300
```

where

- `300`: is a STOP condition (FIFO can contain STOP conditions from moment when I^2C transceivers were first powered up. Should disregard this in user application)
- `100`: is a START condition (ignore lower 8-bits)
- `454`: is our address: bit10:8 = 4 (Address+ACK); bit7:1= slave address (0x2A); bit0 = RD or /WR (0 indicates a write)
- `601`: our first piece of data = 1
- `602`: our 2nd piece of data = 2
- `603`: our 3rd piece of data = 3

- `804`: our 4th piece of data (with a All data received condition) = 4
- `300`: is a STOP condition

Users can also configure the slave to only store data words.

1.8.4 I2C-534 Slave Diagnostic Loopback

The I2C-534 master can loopback into the slave on the same port via either of the diagnostic loopback modes listed below (refer to Section 1.6 for a block diagram):

- **RL loopback**: the SDA and SCL pins of a slave and master for a port connect directly through on-board relays at the connector. The master and slave are connected to the I^2C bus through the master pins and respond as in normal operation. If you disconnect the DB-37 connector, this allows the slave to only respond to the master on the same port.
- **FPGA loopback**: connects master to slave internally and disconnects SDA and SCL from the I^2C bus and external devices. The port master and slave only communicate with each other; however, note that the slave module can function as a fully functional slave and can send and receive data to the master, ACK, and stretch the clock. This is a diagnostic configuration if you wish to troubleshoot software but do not want to affect the external system.

1.9 I2C-534 CRC Checker & Status Reporting

The I2C-534 includes CRC checksum verification for each I^2C transaction that is issued from the host PC and/or embedded CPU to the I2C-534 FPGA, the last device in the data chain.

I^2C configuration, enable, and transfer API send command and data packets to the device, which include a pre-calculated CRC value in the transaction that remains with the data as it moves through the system. The I2C-534 FPGA calculates the CRC and compares it with the pre-calculated value that is tagged to the transaction. In the event of a mismatch, an error is returned from the API call.

The following functions support the I^2C CRC checksum:

```
DqAdv534SetConfig()
DqAdv534MasterSendCommandCRC()
DqAdv534Enable()
DqAdv534MasterSendNChanCRC()
DqAdv534MasterReceiveDataCRC()
DqAdv534ReadBMFIFOCRC()
```

Additionally, the I2C-534 reports hardware status, which can be read via UEI API.

1.10 Termination Pull-up Resistors

Both I2C-534 SDA and SCL are bidirectional lines, connected to a positive supply voltage via a pull-up resistor. When the I^2C bus is free, both lines are HIGH. The output stages of any device that is connected to the bus must have an open-collector or open-drain to accomplish the wired-AND function.

The I2C-534 features 4.99 kΩ termination resistors on both the clock and data lines. Users can configure the I2C-534 to switch in an additional 1.5 kΩ resistor in parallel to reduce the resistance to 1.15 kΩ if required to improve RC constants, or add resistance externally.

1.10.1 Electrical Specification for I^2C Bus

Refer to the UM10204 specification for detailed descriptions.

1.11 Wiring & Connectors (pinout)

Figure 1-7 below illustrates the pinout of the I2C-534.

Each of the four ports on the I2C-534 includes a master I^2C port and/or as a slave I^2C port.

The I2C-534 board uses a 37-pin D-sub connector.

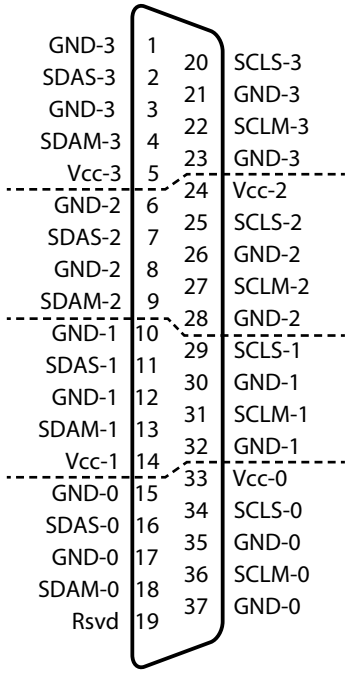


Figure 1-7 Pinout Diagram of the I2C-534 Board

All signals are referenced relative to isolated port ground (GND-x). Dashed line represents isolation barrier between ports. RSVD pin(s) have no internal connection to the I2C-534 board.

NOTE: If you are using an accessory panel with the I2C-534, please refer to the Appendix for a description of the panel.

Chapter 2 Programming with the High-Level API

This chapter provides the following information about using the UeiDaq Framework High-level API to control the DNx-I2C-534:

- About the High-level Framework (Section 2.1)
- Creating a session (Section 2.2)
- Configuring the Resource String (Section 2.3)
- Configuring an I2C Master Port (Section 2.4)
- Configuring an I2C Slave Port(Section 2.5)
- Configuring the Timing (Section 2.6)
- Reading Data (Section 2.7)
- Writing Data (Section 2.8)
- Cleaning-up the Session (Section 2.9)

2.1 About the High-level Framework

UeiDaq Framework is object oriented and its objects can be manipulated in the same manner from different development environments, such as Visual C++, Visual Basic, or LabVIEW.

UeiDaq Framework is bundled with examples for supported programming languages. Examples are located under the UEI programs group in:

- *Start » Programs » UEI » Framework » Examples*

The following sections focus on C++ API examples, but the concept is the same regardless of which programming language you use.

Please refer to the “UeiDaq Framework User Manual” for more information on use of other programming languages.

2.2 Creating a session

The session object controls all operations on your PowerDNx device. Therefore, the first task is to create a session object:

```
// create one session object to handle inputs and outputs
CUeiSession session;
```

2.3 Configuring the Resource String

UeiDaq Framework uses resource strings to select which device, subsystem, and ports to use within a session. The resource string syntax is similar to a web URL:

```
<device class>://<IP address>/<Device Id>/<Subsystem><Channel list>
```

For PowerDNA cube and RACKTangle, the device class is **pdna.**

For example, the following resource string selects I^2C ports on 0,1, and 3 on device 1 at IP address 192.168.100.2: “pdna://192.168.100.2/Dev1/i2c0,1,3”.

The I2C-534 is programmed using the subsystem **i2c** to configure ports as I^2C ports.

2.4 Configuring an I²C Master Port

The method `CreateI2CMasterPort()` is used to configure one or more ports as a master I²C port. The following call configures I²C master ports 0 and 1 of an I2C-534 set as device 1:

```
// configure session's master ports

session.CreateI2CMasterPort("pdna://192.168.100.2/Dev1/i2c0,1",
                            UeiI2CBitsPerSecond100K,
                            UeiI2CTTLLevel3_3V,
                            false);
```

`CreateI2CMasterPort()` configures the following parameters:

- **Bits per second:** port speed of 100k, 400k, or 1M bits per second
- **TTL Level:** TTL level of 3.3V or 5V
- **Enable Secure Shell:** Enables/Disables secure shell mode. If enabled, writing and reading to the master port will use CRC-ensured data transfers.

2.4.1 Configuring Loopback

Users can configure a loopback mode to connect the master and slave (on the same port) either at the front-end connector using a relay or internally at the FPGA level.

The following configures loopback at the connector level, which still generates signals that can be seen externally on the I²C bus:

```
// configure I2C master on port 0

CUeiI2CMasterPort* pMasterPort =
dynamic_cast<CUeiI2CMasterPort*>(session.GetChannel(0));

// Set the loopback mode

pMasterPort->SetLoopbackMode(UeiI2CLoopbackRelay);
```

2.4.2 Configuring Termination

The termination pull-up resistor may be configured to alternate between a resistance of 1.15 kΩ and 4.99 kΩ.

The following enables the resistor in parallel, reducing resistance to 1.15 kΩ:

```
// Set termination resistance

pMasterPort->EnableTerminationResistor(true);
```

2.5 Configuring an I²C Slave Port

The method `CreateI2CSlavePort()`is used to configure one or more ports as a slave I²C port.

The following call configures I²C slave port 3 of an I2C-534 set as device 1:

```
// configure session's slave ports

session.CreateI2CSlavePort("pdna://192.168.100.2/Dev1/i2c3",
                            UeiI2CTTLLevel3_3V,
                            UeiI2CSlaveAddress7bit
                            32);
```

`CreateI2CSlavePort()` configures the following parameters:

- **TTL Level:** TTL level of 3.3V or 5V
- **Slave Address Type:** Use 7-bit or 10-bit addressing mode
- **Slave Address:** Specifies the address to assign to the slave

2.5.1 Configuring Bus Monitoring

Users can configure a slave port to store all data transmitted over the I^2C bus. By default this feature is disabled. To enable bus monitoring, first create a pointer to the slave port you wish to program. Next, enable the bus monitor feature and optionally configure ACK generation.

```
// configure I2C slave on port 0

CUeiI2CSlavePort* pSlavePort =
dynamic_cast<CUeiI2CSlavePort*>(session.GetChannel(0));

// Enable bus monitoring

pSlavePort->EnableBusMonitor(true);

// Optionally configure generating ACKs to simulate device(s) on the bus

pSlavePort->EnableBusMonitorAck(false);
```

2.5.2 Configuring Clock Stretching

Slaves can delay bytes by holding the SCL line LOW during transactions. Clock stretching can be individually enabled for address, transmit, or receive cycles and is configured in 15 nanosecond increments.

The following enables a clock stretching delay of 45 nanoseconds for the address, transmit, and receive cycles:

```
// set clock delay in 15ns increments

pSlavePort->SetClockStretchingDelay(3);

// Configure clock stretching for each cycle type individually

pSlavePort->EnableAddressClockStretching(true);
pSlavePort->EnableTransmitClockStretching(true);
pSlavePort->EnableReceiveClockStretching(true);
```

2.6 Configuring the Timing

The application must configure the I2C-534 to use the “messaging” timing mode.

The following shows how to configure messaging I/O mode:

```
// Configure timing of I2C port

session.ConfigureTimingForMessagingIO(1,0);
```

NOTE: bufferSize and refreshRate are currently unused for the I2C-534. Messages are sent and received when a write or read method is called.

2.7 Reading Data

Reading data form the I2C-534 is done using a *reader* object.

Since there is no multiplexing of data (contrary to what's done with AI, DI, or CI sessions) you need to create one reader object per input port to be able to read from each port in the port list. Note that the reader object can read from both the master and a slave of one port.

The following sample shows how to create a reader object for port 1 and read up to 10 data elements form the I^2C slave and up to 10 elements from the I^2C master.

```
// Create a reader and link it to the session's stream, port 1

reader = new CUeiI2CReader(session.GetDataStream(),1);

// Read up to 10 data element from the slave

tUeiI2CSlaveMessage slaveRx[10];
reader->ReadSlave(10, slaveRx, &numElementsRead);

// Read up to 10 data elements from the master

tUeiI2CMasterMessage masterRx[10];
reader->ReadMaster(10, masterRx, &numElementsRead);
```

2.8 Writing Data

Writing data to the I2C-534 is done using a writer object.

Since there is no multiplexing of data (contrary to what's done with AO, DO, or CO sessions), you need to create one writer object per output port to be able to write to each port in the port list. Note that the writer object can write to both the master and slave of one port.

The following sample shows how to create a writer object for port 1 and write 10 bytes of data to slave followed by a write to a master to request the slave data over the I^2C bus:

```
// Create a writer a link it to the session's stream, port 1

writer = new CUeiI2CWriter(session.GetDataStream(),1);

// Initialize 10 uint16s to 0x5 that will be written to the slave
// NOTE: only lower 8 bits are currently used for data

uInt16 txData[10];
for(int i=0; i<10; i++) txData[i] = 0x5;

// Write data to slave FIFO, which slave uses when replying to master

writer->WriteSlaveData(10, txData, &numElementsWritten);

// Build I2C read command for master

tUeiI2CMasterCommand params;
params.type = UeiI2CCommandRead;
params.slaveAddress = 0x20;
params.numReadElements = 10;

//Write command to master

writer.WriteMasterCommand(&params);
```

2.9 Cleaning-up the Session

The session object will clean itself up when it goes out of scope or when it is destroyed. To reuse the object with a different set of ports or parameters, you can manually clean up the session as follows:

```
// Clean up the session

session.CleanUp();
```

Chapter 3 Programming with the Low-Level API

This chapter provides the following information about programming the I2C-534 using the low-level API:

- About the Low-level API (Section 3.1)
- Low-level Functions (Section 3.2)
- Low-level Programming Techniques (Section 3.3)
- Configuring I2C Interface (Section 3.4)
- XDCP™ Device (Renesas X9119) (Section 3.5)
- Bus Monitor Functionality (Section 3.6)
- Slave Functionality (Section 3.7)
- Controlling DC/DC (Section 3.8)

3.1 About the Low-level API

The low-level API provides direct access to the DaqBIOS protocol structure and registers in C. The low-level API is intended for speed-optimization, when programming unconventional functionality, or when programming under Linux, Windows, or real-time operating systems.

When programming in Windows OS, however, we recommend that you use the UeiDaq high-level Framework API (see **Chapter 2**). The Framework simplifies the use of the low-level API that makes programming easier and faster while still providing access to the majority of low-level API features.

For additional information regarding low-level programming, refer to the "PowerDNA API Reference Manual" located in the following directories:

- On Linux systems:
  <PowerDNA-x.y.z>/docs
- On Windows systems:
  Start » All Programs » UEI » PowerDNA » Documentation

3.2 Low-level Functions

Table 3-1 provides a summary of I2C-534-specific functions. All low-level functions are described in detail in the *PowerDNA API Reference Manual*.

Table 3-1 Summary of Low-level API Functions for DNx-I2C-534

Function	Description
`DqAdv534SetConfig`	sets up layer configuration
`DqAdv534Enable`	enables/disables I2C-534 ports
`DqAdv534Flush`	flushes receive and/or transmit FIFOs on slave and/or master depending on the <flags> for the selected bitmask of ports
`DqAdv534BuildCmdData`	configures the master command to be sent using `DqAdv534MasterWriteTxFIFO`

Table 3-1 Summary of Low-level API Functions for DNx-I2C-534 (Cont.)

Function	Description
DqAdv534MasterWriteTxFIFO	writes to the master FIFO and returns the number of words written and the number of words still available in the FIFO
DqAdv534MasterReadRxFIFO	reads from the master FIFO and returns the number of words retrieved and the number of words still available in the FIFO
DqAdv534SlaveWriteTxFIFO	writes to the slave FIFO and returns the number of words written and the number of words still available in the FIFO
DqAdv534SlaveReadRxFIFO	reads from the slave FIFO and returns the number of words retrieved and the number of words still available in the FIFO
DqAdv534MasterSendCommandCRC	sends data in a secure fashion. If FPGA CRC check failed, the data is discarded. You can program this function to wait for all BM data associated with the transmission and return only after transaction is completed on the bus
DqAdv534MasterReceiveCommandCRC	receives data transmitted from the slave to the master, enveloped in CRC
DqAdv534ReadBMFIFOCRC	reads bus monitor data in a secure fashion. If FPGA CRC check failed, the data is discarded.
DqAdv534GetStatus	retrieves per port status
DqAdv534CalcCustomTiming	Calculates parameters for a custom baud rate between 2kHz and 100kHz for non-standard I2C devices
DqAdv534MasterSendNChanCRC	Sends data at the same time to multiple ports defined in the port mask. The function does CRC check for all ports requested and then simultaneously starts transaction
DqAdv534MasterWriteTxPhyFIFO	Writes to the PHY FIFO - lowest level FIFO. Write is performed in the form of atomic commands on I2C bus
DqAdv534BusControl	configuration per port for enabling/disabling: DC/DC, internal loopback, and parallel pull-up resistors
DqAdv534WriteCLOFIFO	Not fully implemented or tested
DqAdv534ReadCLIFIFO	Not fully implemented or tested

3.3 Low-level Programming Techniques

Application developers are encouraged to explore the existing source code examples when first programming the I2C-534. Sample code provided with the installation is self-documented and serves as a good starting point.

Code examples are located in the following directories:

- On Linux systems: <PowerDNA-x.y.z>/src/DAQLib_Samples
- On Windows: *Start » All Programs » UEI » PowerDNA » Examples*

Code examples specifically for the I2C-534 have 534 specified in the name, (i.e., Sample534.c).

I2C-534 can be operated using the immediate (point-to-point) data acquisition protocol. Sample534.c provides an example of acquiring data using this mode.

3.4 Configuring I²C Interface

The I2C-534 master and slave ports are configured using the DqAdv534SetConfig() API.

```
int DqAdv534SetConfig(int hd, int devn, int port,
pI2C534CFG pCfg);
```

Parameters consist of the following:

- int hd - handle to the IOM
- int devn – device number in the layer stack
- int port – port to apply configuration parameters to
- pI2C534CFG pCfg – structure for programming port configuration

Configuration options are set using the pI2C534CFG structure, as described below.

Note that the <flags> member is used to specify which parameters are able to be changed. Members can accept a single value and some accept a logically grouped combination of constants. Refer to the *PowerDNA API Reference Manual* for descriptions of each parameter.

```
typedef struct {
uint32 flags;       // select active parameters to set/change
uint32 clock;       // clock frequency, 100k, 400k and 1Mbit supported;
                    // 0 = custom parameters
float ttl_level;    // set line voltage (3.3V and 5.1V for now)
uint32 tx_lines;    // enable termination and loopback
MCTPARAM mctprm     // custom timing parameters

// Master configuration
uint32 master_cfg;              // master configuration bitset
uint32 master_idle_delay;       // delay in MM mode before acquiring bus
                                //   in 15ns increments
uint32 master_byte_delay;       // delay between bytes sent by master
                                //   in 1us resolution
uint32 master_max_sync_delay;   // maximum delay in uS slave
                                //   could delay clock
uint32 master_datasz_unfifo;    // <reserved>
uint32 master_to_cfg;           // Maximum timeout delay in uS
                                //  before releasing bus (0 == default)
uint32 master_wait_bm_fifo_ms;  // Maximum wait for BM FIFO to receive
                                //    all expected words, ms
uint32 master_xdcp_device_type; // <reserved>

// Slave configuration
uint32 slave_cfg;               // slave configuration bitset
uint32 slave_addr;              // select 7/10 slave address and 10-bit
                                // address mode
```

```
    uint32 slave_data;              // data to reply when slave Tx FIFO
                                    //    is empty
    uint32 slave_sync_dly;          // length of ACK in 15.15ns clocks
                                    //   (acknowledge cycle stretch)
    uint32 slave_ack_dly;           // 12-bit how long we wait for
                                    //   master ACK (in clocks)
    uint32 slave_max_ack;           // Slave RX max count register
                                    //   (# of words per /ACK)
    uint32 slave_tx_reg_size;       // in unFIFO mode the size of bytes
                                    //   to transmit to master, 1..4
} I2C534CFG, *pI2C534CFG;
```

<flags> define what parts of the configuration structure are valid. If the master needs to be programmed `DQ_L534CFG_MASTER_VALID` flag is required. The same applies to the slave functionality with `DQ_L534_SLAVE_VALID` bit. <flags> also allows to set up the clock, voltage level and termination. Parameters which are not set are replaced with the default values.

If `DQ_L534CFG_MASTER_VALID` or `DQ_L534_SLAVE_VALID` bits are set, values of <master_cfg> fields are used for programming the board, otherwise these fields are ignored.

By using this strategy configuration calls can be additive, so each following call adds or changes a parameter in the card configuration. Calling `DqAdv534Enable()`with the bit set to the programmed port in <port_mask> causes new parameters to take effect.

To reset configuration back to the initial state call `DqAdv534SetConfig()` with `DQ_L534CFG_CLEAR` bit set in <flags>.

3.4.1 Setting Up Custom Clock Rate

The DNx-I2C-534 supports custom rates between 2kHz and 100kHz. This rate is intended for use with custom, slower devices implemented on CPU.

One of the members in the `I2C534CFG` structure is `MCTPARAM mctprm`. While it is possible to calculate and use completely custom timing parameters and program clock rate anywhere from 1kHz to above 1MHz, the current implementation limits custom clock rates to between 2kHz and 100kHz.

NOTE: master and slave controllers on the same port share clock frequency settings

To set up a custom clock rate use `DQ_L534CFG_CLOCK_CUST` in <clock> field (`DQ_L534CFG_CLOCK` bit needs to be set in the <flags> as well).

Then call a helper function `DqAdv534CalcCustomTiming()` and pass a pointer `pMCTPARAM` to `MCTPARAM mctprm` in `pI2C534CFG pCfg.` <divider> divides 100kHz clock for up to 50, calculates and stores proper timing parameters into `mctprm.` The following call to `DqAdv534SetConfig()` programs these timing parameters on the card.

As an example the following diagram shows operation with the divider equal to 2.5, i.e. 40kHz.

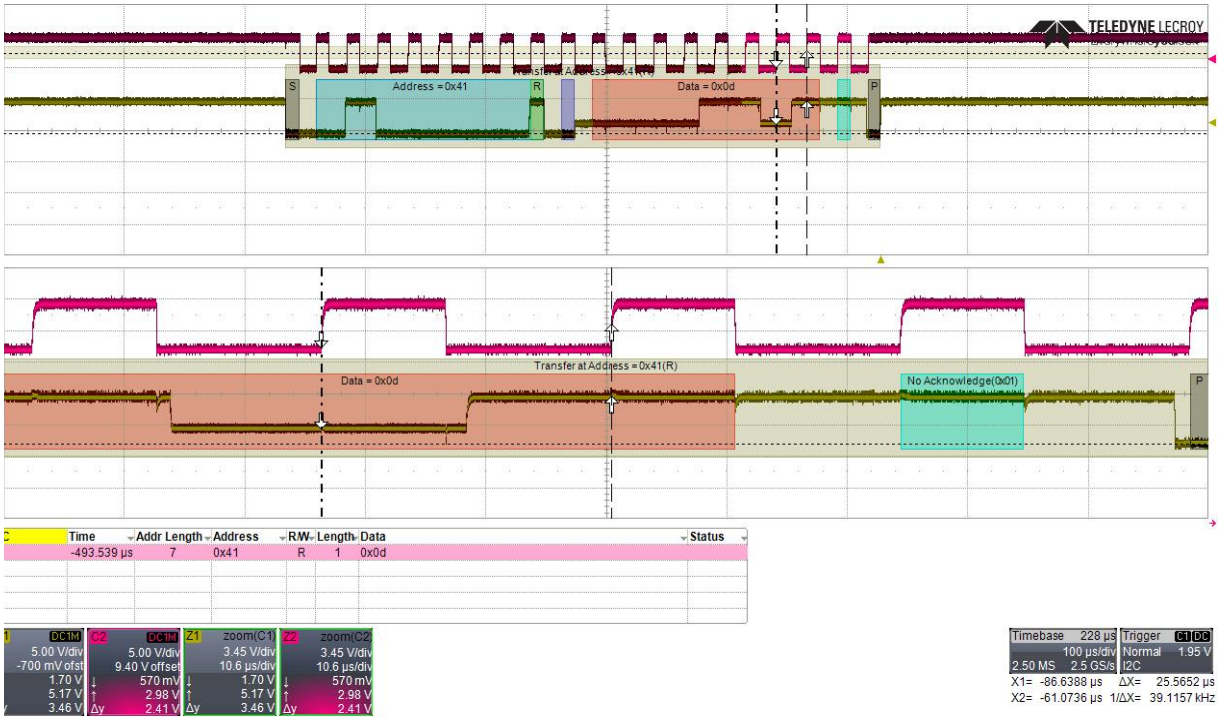


Figure 3-1 Read Command with custom clock rate set to 40.0kHz

3.4.2 CRC-Enabled Functions

DNx-I2C-534 supports a set of functions that give additional CRC checking called “secure shell”. All these functions include CRC suffix at the end of the function name:

```
DqAdv534MasterSendCommandCRC()
DqAdv534MasterReceiveDataCRC()
DqAdv534ReadBMFIFOCRC()
DqAdv534MasterSendNChanCRC()
```

In addition, the following functions verify parameters using the CRC mechanism as well:

```
DqAdv534SetConfig()
DqAdv534Enable()
```

CRC guarding of parameters and data works as follows:

1. Arguments of the function are stored into the request packet and **CRC16 CCITT** is calculated
2. Request packet is sent to the IOM firmware where CRC of the input parameters and data is calculated and compared with CRC passed from the requester.
3. If CRCs do not match, no operation is performed and firmware replies back with the error code `DQ_CRC_CHECK_FAILED`
4. Library replies to the caller with an error

In addition to this, `DqAdv534MasterSendCommandCRC()` and `DqAdv534MasterSendNChanCRC()` generate CRC16 CCITT code to be passed with the command data.

Each master command has the following structure:

[0] Command

[1] CRC16 CCITT

[2] Data...

When command is formed by the function call and data written to the buffer to be transmitted function calculated CRC for each command and stores it into the buffer. On the firmware side, function code writes received data into the FIFO and then reads back calculated CRC of the written data and compares it with the one calculated by the FPGA. If they match, an “execute” command is issued. However, the FPGA will ignore this command should it find CRC code mismatch as well. This protocol secures data to be sent on an I^2C bus from the function call to the FPGA pin.

When data is sent back, data CRC is calculated again for the reply and stored alongside with it in the packet. Library verifies that the received reply CRC matches calculated CRC.

There are three specific errors which can be returned by CRC-secured functions:

`DQ_CRC_CHECK_FAILED` - received/transmitted parameters failed CRC secure shell check

`DQ_DEVICE_BUSY` - FIFO cannot accept more data because it encountered CRC or command failure during the previous call. In this case the user needs to call `DqAdv534Flush()` to reset output FIFOs

`DQ_DATA_ERROR` - this error is only possible when <flags> in `DqAdv534MasterSendCommandCRC()` has `DQ_L534_MSENDCRC_DOUBLECHECK_CRC` bit set. In this case every command word written into the FPGA is read back and verified against already verified command words in the CPU memory. Generally this check is not required to guarantee validity of the data down to the pin.

3.4.3 Command and Raw Mode Functions

There are two ways the master can control I^2C bus - command and raw modes. Command mode issues fully enveloped, CRC-secured commands which are performed on the FPGA and a status and/or data is returned. The second way is to fill the FPGA FIFO with atomic transactions on the bus - what is called “raw” mode. The FPGA cannot secure raw mode commands with CRC by the nature of the operation (library and firmware can still secure command and data down to the FPGA registers)

In the command mode the following commands are supported:

```
#define DQ_L534_CMD_TDELAY     (1<<28)    //Insert NOP command for delay
                                          //sequence
#define DQ_L534_CMD_STOP       (2<<28)    //STOP - issue a stop condition
                                          // once bus is available
#define DQ_L534_CMD_ST_WRITE   (3<<28)    //START+WRITE (including write
                                          // multiple)
#define DQ_L534_CMD_ST_READ    (4<<28)    //START+READ (including read
                                          // multiple)
```

```
#define DQ_L534_CMD_ST_WRRD    (5<<28)    //START+WRITE+RESTART+READ
                                          //(including read multiple)
#define DQ_L534_CMD_XDCP_READ  (6<<28)    //START+WRITE+READ (for Renesas
                                          // XDCP protocol - ex. X9119)
#define DQ_L534_CMD_XDCP_WRITE (7<<28)    //START_WRITE+WRITE (for Renesas
                                          // XDCP protocol - ex. X9259)
```

Two XDCP™ commands are created for Renesas devices which use their own I^2C protocol conventions. Each of these commands can be issued in CRC-secured mode.

If a user doesn't want or doesn't need to secure command with CRC or needs to send more than one command to one port in a single call, it can be accomplished with the combination of DqAdv534BuildCmdData() to create an array of data (in this case use DQ_L534_MSENDCRC_IGNORE_CRC flag to suppress CRC check) and DqAdv534MasterWriteTxFIFO().

Use master_cfg |= DQ_L534MCFG_RAWMODE in pCfg parameter of DqAdv534SetConfig() call to enable raw mode instead of command mode of operations.

In raw mode master TX FIFO is written with atomic commands (using DqAdv534MasterWriteTxPhyFIFO() call):

```
I2C_MRAW_PHY_TX1(B)      // PHY: Set SDA to 1 for one clock (use B = 0)
I2C_MRAW_PHY_TX0(B)      // PHY: Set SDA to 0 for one clock (use B = 0)
I2C_MRAW_PHY_RX(B)       // PHY: Set SDA to 1 for one clock, return SDA at the falling edge
                         // of the clock (use B = 0)
I2C_MRAW_PHY_START(B)    // PHY: START condition on the bus (use B = 0)
I2C_MRAW_PHY_STOP(B)     // PHY: STOP condition on the bus (use B = 0)
I2C_MRAW_PHY_RELEASE(B)  // PHY: Release bus without creating START or STOP conditions
                         // (use B = 0)
I2C_MRAW_DLY_2NUS(B)     // MASTER: Delay execution for 2^n uS (n is in 5 LSBs, B = 0..31)
I2C_MRAW_DLY_NX8US(B)    // MASTER: Delay execution for n*8uS (n is in 8 LSBs, B = 0..255)
I2C_MRAW_BYTE_SEND(B)    // MASTER: Transmit ddd data to the I2C bus
                         // (ddd is in 8 LSBs B = 0..255)
I2C_MRAW_BYTE_RECEIVE(B) // MASTER: Read byte of data from the I2C bus and save to the RX
                         // FIFO (B = 0)
I2C_MRAW_ACK_WAIT(B)     // MASTER: Wait for the ACK, NACK ends sequence (B = 0 )
I2C_MRAW_SEQ_END(B)      // MASTER: Last command in the sequence, write starts execution
                         // (B = 0)
```

DqAdv534MasterWriteTxPhyFIFO() writes command sequence into master PHY FIFO - the lowest accessible level of I^2C state machine implemented on the FPGA. This allows almost unlimited flexibility to control the bus. Please notice that clock stretching and timeout parameters set in DqAdv534SetConfig() remain in full effect.

As an example, a "START+WRITE+RESTART+READ" (i.e. DQ_L534_CMD_ST_WRRD command) sequence is presented below:

```
I2C_MRAW_PHY_START(0);      // start
I2C_MRAW_BYTE_SEND(0xA0);   // address + write
I2C_MRAW_ACK_WAIT(0);       // slave ACK
I2C_MRAW_BYTE_SEND(0xF0);   // register
I2C_MRAW_ACK_WAIT(0);       // slave ACK
I2C_MRAW_PHY_RELEASE(0);    // release + start = restart
I2C_MRAW_PHY_START(0);
I2C_MRAW_BYTE_SEND(0xA1);   // address + read
I2C_MRAW_ACK_WAIT(0);       // slave ACK
```

```
I2C_MRAW_BYTE_RECEIVE(0);   // receive
I2C_MRAW_PHY_TX1(0);        // master NACK
I2C_MRAW_PHY_STOP(0);       // stop
I2C_MRAW_SEQ_END(0);
```

You can find more examples of raw mode operation in Sample534.

3.5 XDCP™ Device (Renesas X9119)

Renesas X9119 digital potentiometer supports XDCP™ protocol which shares physical and electrical characteristics with I^2C protocol, however read command is accomplished differently. Instead of writing address byte and reading data from the slave, the master has to write address byte, then instruction byte and only after that it reads data.

3.5.1 Write Wiper Counter Register

A write into the wiper counter register is defined as the following taken from the datasheet:

WRITE WIPER COUNTER REGISTER (WCR)

S T A R T	DEVICE TYPE IDENTIFIER				DEVICE ADDRESSES				S A C K	INSTRUCTION OPCODE				REGISTER ADDRESSES				S A C K	WIPER POSITION (SENT BY MASTER ON SDA)								S A C K	WIPER POSITION (SENT BY MASTER ON SDA)								S A C K	S T O P
	0	1	0	1	A2	A1	A0	R/$\overline{W}$ = 0		1	0	1	0	0	0	0	0		X	X	X	X	X	X	WCR9	WCR8		WCR7	WCR6	WCR5	WCR4	WCR3	WCR2	WCR1	WCR0		

Selecting zero address, a write to the wiper register should look like:

0x50 0xA0 0x1 0xB0 (decimal value of 432 = 0x1B0 is selected)

We use the following command to write to the X9119 (please refer to master_ren_x9119() from Sample534MasterDigiPots.c):

```
wr_address = X9119_ID_ADDRESS(X9119_ADDRESS);
n_bytes = x9119_word_3(X9119_WR_WIPER, wiper_pos, set_wiper);
ret = DqAdv534MasterSendCommandCRC(hd, devn, m_port,
                                   DQ_L534_MSENDCRC_WAIT_FOR_BM,
                                   DQ_L534_CMD_ST_WRITE|wr_address,
                                   0, n_bytes, set_wiper,
                                   &transmitted, &available,
                                   &crc_stat);
```

Where <set_wiper> contains 10-bit data for the potentiometer.The diagram below confirms execution.

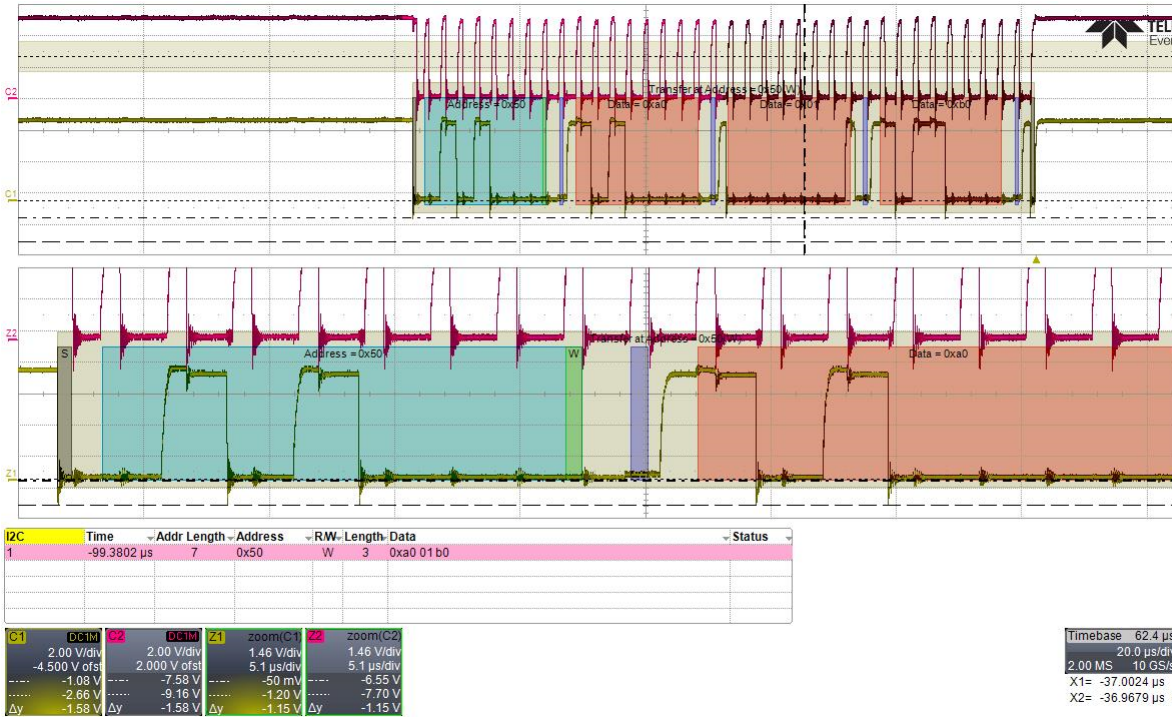


Figure 3-2 Write to Wiper Counter Register using XDCP™

3.5.2 Read Wiper Counter Register

A read from the wiper counter register consists of writing one byte with the address then writing a byte with the instruction opcode following reading of two bytes from the X9119 IC.

READ WIPER COUNTER REGISTER (WCR)

S T A R T	DEVICE TYPE IDENTIFIER				DEVICE ADDRESSES				S A C K	INSTRUCTION OPCODE				REGISTER ADDRESSES				S A C K	WIPER POSITION (SENT BY SLAVE ON SDA)								M A C K	WIPER POSITION (SENT BY SLAVE ON SDA)								M A C K	S T O P
	0	1	0	1	A2	A1	A0	R/W = 1		1	0	0	0	0	0	0	0		X	X	X	X	X	X	WCR9	WCR8		WCR7	WCR6	WCR5	WCR4	WCR3	WCR2	WCR1	WCR0		

For address = 0 we expect writing 0x51 (read bit set) then 0x80 following reading back two bytes 0x1 (notice that bits [7..2] of the first read byte are not defined) and 0xB0. 0x1B0 was the digital potentiometer position written in the previous write.

We use the following command to read wiper position from the X9119 (notice it is formed differently than for write. Please refer to master_ren_x9119() from Sample534MasterDigiPots.c):

```
rd_address = L534_XDCP_CLO_ID_M(X9119_ID)|
             L534_XDCP_CLO_BYTE1(X9119_ADDRESS)|
             L534_XDCP_CLO_BYTE1_RD|
             L534_XDCP_CLO_BYTE2(X9119_RD_WIPER);
n_bytes = 2;
ret = DqAdv534MasterSendCommandCRC(hd, devn, m_port,
                                   DQ_L534_MSENDCRC_WAIT_FOR_BM,
                                   DQ_L534_CMD_XDCP_READ | rd_address, 0,
                                   n_bytes, read_wiper, &transmitted,
```

```
                                    &available, &crc_stat);
```

Oscilloscope confirms read sequence and the position of acknowledges (notice that two are slave and two are master ACKs)

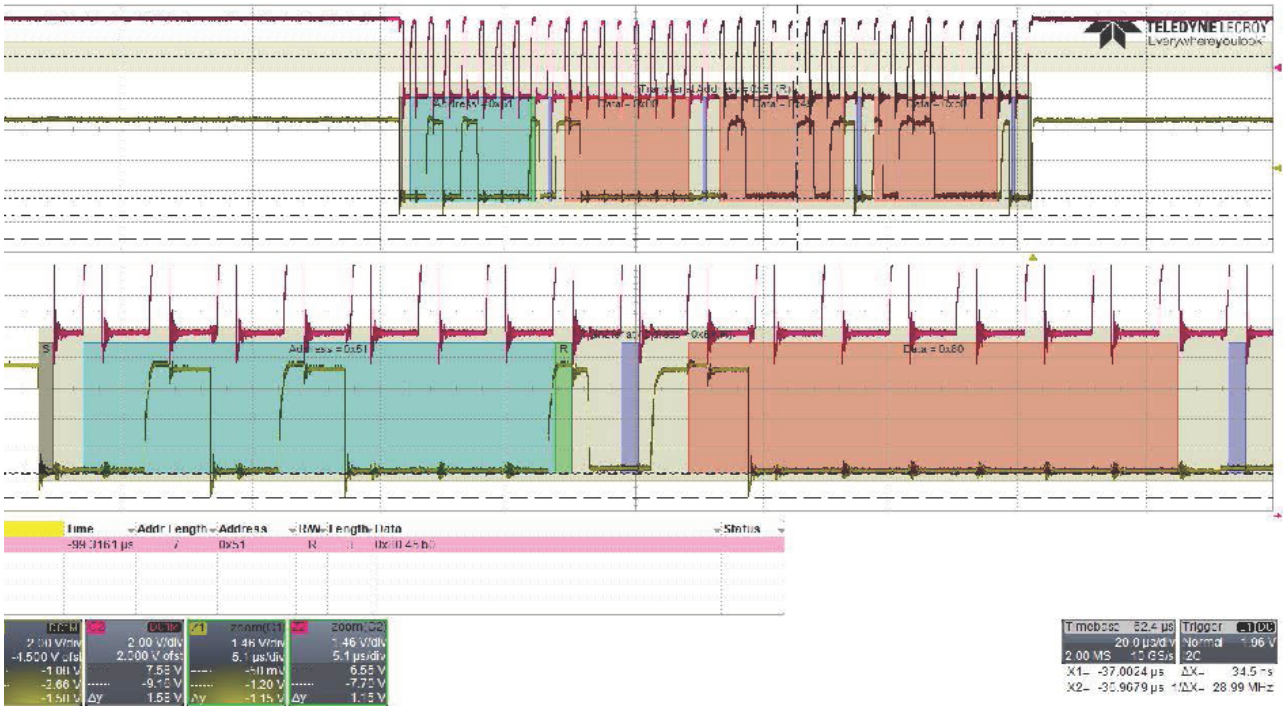


Figure 3-3 Read from Wiper Counter Register using XDCP™

3.5.3 Write-Restart-Read Command

Sometimes a write-restart-read is used to retrieve data. This command writes one byte to an address first and then performs read of a specified number of bytes from the device. Lack of STOP condition between these two separate commands tells slave device to treat them as a single transaction. For example, Maxim DS3930 digital potentiometer requires writing register number first and reading back its content in a single write-restart-read transaction.

The following code shows how to set this up (please refer to master_max_ds3930() from Sample534MasterDigiPots.c):

```
read_bytes = 1;
write_bytes = ds3930_get_pot_1wr(pot_num, set_wiper);
ret = DqAdv534MasterSendCommandCRC(hd, devn, m_port,
                                   DQ_L534_MSENDCRC_WAIT_FOR_BM,
                                   DQ_L534_CMD_ST_WRRD|address,
                                   write_bytes, read_bytes, set_wiper,
                                   &transmitted, &available,
                                   &crc_stat);
```

Figure 3-4 Write-restart-read zoomed-in scope capture

3.6 Bus Monitor Functionality

DNx-I2C-534 implements both master and slave functionality on each port. Generally I^{2}C is expected to answer (i.e acknowledge and/or transmit data in response of master clocking) only when device and register addresses matches those supported by the slave. However, in our case the slave implements a special mode of operation which is called bus monitor (BM). Bus monitor can be implemented on any port.

NOTE: Selecting `cfg.master_cfg |= DQ_L534MCFG_SECURE_SHELL` mode automatically enables BM mode on the same port as the master. However, it is up to the user where to connect SDAM and SCLM pins to MDAM and MCLM lines. It can be done remotely at the end of the I^{2}C physical bus with appropriate wiring or directly on the connector by enabling loopback in software adding `cfg.tx_lines |= DQ_L534_CFG_RL_LOOPBK` bit in to the configuration. There is even deeper loopback available by setting bit `DQ_L534CFG_FPGA_LOOPBK` in the same <tx_lines> field to connect master and slave signals on the same port without ever leaving the FPGA. This mode is specifically designed for debugging software while DNx-I2C-534 layer is connected to an actual device to avoid device damage.

Using `DqAdv534MasterSendCommandCRC()` or `DqAdv534MasterSendNChanCRC()` a flag `DQ_L534_MSENDCRC_WAIT_FOR_BM` can be used to wait until all data is accumulated in BM FIFO before returning from the function call. The data from BM FIFO can be read using `DqAdv534ReadBMFIFOCRC()` with the data secured by CRC or a faster call to `DqAdv534SlaveReadRxFIFO()`.

If the loopback is enabled BM FIFO can be read on the same port. With the external wiring a slave on any port can be set up to act as a BM. To do that, flag `slave_cfg = DQ_L534CFG_ENABLE_BM` needs to be set in the slave configuration.

If there is no actual slave present on the bus, then there is nobody available to drive ACK and master will terminate transaction once timeout period expires. To test master command on the bus without an external device user can use `DQ_L534CFG_ACK_BM` flag in the same <slave_cfg> field. This flag tells slave to simulate slave ACK condition when the slave is in BM mode.

The data from the BM is 12-bits wide (in uint16). Bits [7..0] contain address or data word and bits [11..8] contain bus conditions. Chapter 1 details format of the BM data word.

3.7 Slave Functionality

DNx-I2C-534 implements basic slave functionality. Generally, slave functionality is used to emulate a device on the bus for some external master and receive command/data from the master.

With write commands, if 7 or 10-bit address matches with programmed slave address <slave_addr> field (<slave_cfg> has `DQ_L534CFG_10BIT` flag set in case of 10-bit address) received data ends up in the slave receive FIFO where it can be read using `DqAdv534SlaveReadRxFIFO().`

With read command (addressing stays the same) there are two ways to feed data back to a master:

1. Using a slave Tx FIFO which is filled using `DqAdv534SlaveWriteTxFIFO().` If the master reads more bytes than the FIFO contains the last byte is repeated until new data is available in the FIFO.

2. Very often user needs to emulate a device (i.e. accelerometer or ADC) which returns 1 to 4 bytes of data upon a read command from a certain address. To enable this “Un-FIFO” or register mode add `slave_cfg |= DQ_L534CFG_SLAVE_UNFIFO` and set <slave_data> field for the initial data (all 32-bits are used, byte order is 0x[byte0[byte1][byte2][byte3])

In register mode the value I^2C slave replies to the master with can be changed with:

`DqAdv534SetConfig()` with <flags> set to `DQ_L534CFG_SDATA_ADDR` with fields <slave_addr> and <slave_data> filled and slave address and data will be immediately updated.

3.8 Controlling DC/DC

Each DNx-I2C-534 port has its own DC/DC which can be turned on and off through software. By default, all DC/DCs are on.

Function `DqAdv534BusControl()` allows to control the state of the DC/DC on an as-needed basis to save power or to reset slave device in case it is powered from the I2C-534 port itself.

Set <flags> parameter to `DQL_IOCTL534_BUSCON_DCDC` and pass a port mask of the on and off DC/DCs in <parameter[0]>. After turning DC/DC off and on you might need to reset the FIFOs using `DqAdv534Flush()` with `DQL_IOCTL534_FLUSHALL` parameter.

Since all I^2C state machines reside on the non-isolated FPGA side configuration reprogramming or re-enabling is not required. Another useful feature of `DqAdv534BusControl()` with flag `DQL_IOCTL534_BUSCON_LBENTERM` is to select loopback and termination (i.e. enabling an additional 1.5k pull-up resistor or connecting master and slave lines of the same port at the connector) on the fly without the need to reconfigure the board.

Appendix

A.1 Accessories

The following cables and STP boards are available for the I2C-534 board.

DNA-CBL-37

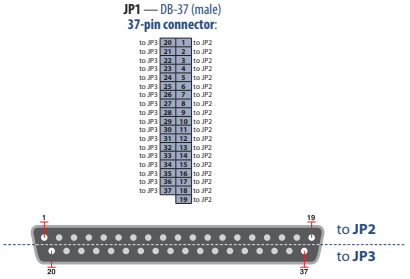
3ft, 37-way flat ribbon cable; connects I2C-534 to panels to DNA-STP-37.

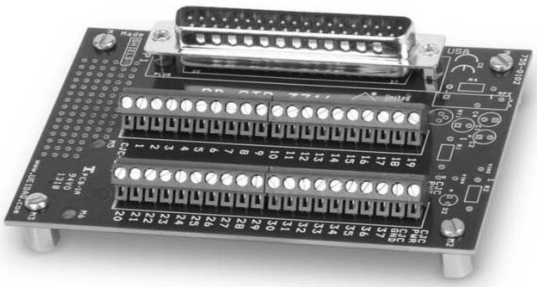
DNA-CBL-37S

3ft, 37-way shielded cable; connects I2C-534 to panels to DNA-STP-37.

DNA-STP-37

37-way screw terminal panel.





Index