DNx-MF-101
User Manual

Multifunction I/O Board
for the PowerDNA Cube and RACK Series Chassis

March 2025

PN Man-DNx-MF-101

CE

Contacting United Electronic Industries

Mailing Address:
249 Vanderbilt Avenue
Norwood, MA 02062
U.S.A.

Shipping Address:
24 Morgan Drive
Norwood, MA 02062
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: uei.support@ametek.com
Website: www.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 / AMETEK 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 / AMETEK product for that purpose is null and void and United Electronic Industries / AMETEK 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

List of Tables

Chapter 1 Introduction

This manual outlines the feature set and use of the DNx-MF-101, a multifunction board with analog and digital I/O, an I^2C port, and a serial port.

The following sections are provided in this chapter:

- Organization of this Manual (Section 1.1)
- Manual Conventions (Section 1.2)
- Naming Conventions (Section 1.3)
- Related Resources (Section 1.4)
- Before You Begin (Section 1.5)
- DNx-MF-101 Features (Section 1.6)
- Technical Specifications (Section 1.7)

1.1 Organization of this Manual

This DNx-MF-101 User Manual is organized as follows:

- **Introduction**
  Chapter 1 summarizes the features and specifications of the DNx-MF-101.
- **I/O Functional Descriptions**
  Chapter 2 describes the device architecture, logic, and connectivity of the DNx-MF-101 subsystems.
- **PowerDNA Explorer**
  Chapter 3 shows how to explore DNx-MF-101 features through a GUI-based application.
- **Programming with High-level API**
  Chapter 4 describes how to configure the DNx-MF-101, read data, and write data with the Framework API.
- **Programming with Low-level API**
  Chapter 5 provides an overview of C commands for configuring and using the DNx-MF-101.
- **Accessories**
  Appendix A provides a list of accessories available for use with the DNx-MF-101.

1.2 Manual Conventions

The following conventions are used throughout this manual:

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

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

NOTE: Notes alert you to important information.

Typeface	Description	Example
bold	field or button names	Click **Scan Network**
»	hierarchy to get to a specific menu item	**File » New**
`fixed`	source code to be entered verbatim	`session.CleanUp()`
<brackets>	placeholder for user-defined text	`pdna://<IP address>`
italics	path to a file or directory	*C:\Program Files*

1.3 Naming Conventions

The DNA-MF-101, DNR-MF-101, and DNF-MF-101 board versions are compatible with the UEI Cube, RACKtangle, and FLATRACK chassis respectively. These boards are electronically identical and differ only in mounting hardware. The DNA version stacks in a Cube chassis, while the DNR and DNF versions plug into the backplane of a Rack chassis. Throughout this manual, the term DNx-MF-101 refers to both Cube and Rack products.

1.4 Related Resources

This manual only covers functionality specific to the DNx-MF-101. To get started with your Cube or Rack, please see the documentation included with the software installation. On Windows, these resources can be found from the desktop by clicking **Start » All Programs » UEI**

UEI's website includes other user resources such as application notes, FAQs, tutorials, and videos. In particular, the glossary of terms may be helpful when reading through this manual: https://www.ueidaq.com/glossary

Additional questions? Please email UEI Support at uei.support@ametek.com or call 508-921-4600.

1.5 Before You Begin

No Hot Swapping!

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.

Check Your Firmware

Ensure that the firmware installed on the Cube or Rack CPU matches the UEI software version installed on your PC. The IOM is shipped with pre-installed firmware and a matching software installation. If you upgrade your software installation, you must also update the firmware on your Cube or RACK CPU. See *"Firmware Update Procedures.pdf"* for instructions on checking and updating the firmware. These instructions are located in the following directories:

- On Linux: *PowerDNA_Linux_<x.y.z>/docs*
- On Windows:
  Start » All Programs » UEI » DNx Firmware Update Procedures

1.6 DNx-MF-101 Features

The DNx-MF-101 Multifunction I/O Board is an ideal measurement solution for a variety of automotive, aerospace and power generation applications. This multifunction I/O board includes the following channels:

- 16 single-ended or 8 fully differential analog inputs
- 2 analog outputs
- 16 industrial digital I/O bits
- 6 TTL digital bits (4 I/O, 1 input, 1 output)
- 2 counter/timers, routable to TTL or industrial digital I/O
- 1 RS-232/422/485 port
- 1 I^2C port

1.6.1 Analog Input

The DNx-MF-101 is equipped with 16 independently configurable analog input channels and an 18-bit A/D converter. Inputs are buffered to eliminate multiplexer-based settling time issues. Each channel supports a sampling rate of up to 2000 samples/s (32 kS/s aggregate), and channels can be paired to measure in differential mode.

The board offers software-selectable A/D ranges between ±80 V to ±0.156 V. The upper end eliminates the need for external signal conditioning, while the lower end allows for precise measurements down to 1.19 microvolts resolution.

To improve noise immunity, an Embedded Averaging engine automatically acquires as many samples as possible for the given gain/speed and calculates the average.

1.6.2 Analog Output

Two 16-bit analog output channels are independently configurable as either voltage output or current output. Users may choose among software selectable ranges up to ±10 V or 0-20 mA.

For applications requiring higher output current or voltage, please refer to the DNx-AO-308-35x series boards.

1.6.3 Digital I/O

The DNx-MF-101 includes 16 channels of industrial digital I/O and 6 channels of logic-level I/O (4 configurable as input or output in pairs, 1 input, and 1 output).

1.6.3.1 Industrial Bits

The industrial digital I/O subsystem operates across a wide range, from 3.3 V to 55 VDC. Each industrial bit is independently configurable as either input or output. Voltage is supplied in groups of 4 bits (up to 4 different VCCs across 16 bits).

Inputs: Each input is sensed with a dedicated 200 kHz A/D converter. High and low thresholds are therefore programmable and state changes can be detected with 5 microsecond resolution. Programmable pull up/down resistors allow inputs to monitor contacts connected to a supply voltage or ground. In the absence of an external supply voltage, the lines are weakly pulled up to an internal 60V supply (via a 2 MΩ resistor); this ensures that inputs allow the full 0-55 V range, but can be easily overdriven by an external source.

Outputs: Each output may be configured as either current sourcing (connect output to Vcc) or sinking (connect output to Gnd). Outputs are rated for continuous operation at 500 mA with an output voltage drop of less than 600 mV. Each channel is protected with a 1.25 Amp fast-blow fuse.

Industrial digital outputs are equipped with an optional pulse-width modulated (PWM) “soft-start” or “soft-stop” feature. This allows power to be applied/removed gradually, greatly increasing the reliability of devices like incandescent bulbs where thermal shock reduces life expectancy. The ‘soft-start” parameters are selectable on a per-channel basis.

PWM can also be configured to run continuously for low speed, high voltage/current applications. The board supports pulse-width resolution up to 16-bits and frequency up to 10 kHz.

1.6.3.2 TTL Bits

A total of 6 logic-level channels are provided: four channels are configurable as input or output in pairs, one is a dedicated input, and one is a dedicated output. Outputs use 5 V logic, but inputs are compatible with either 2.5 V, 3.3 V, or 5 V.

1.6.3.3 Counters

Two 32-bit counters perform up/down counting. Several flexible modes are available including event counting, pulse-width/period measurements, and quadrature decoding. Counter inputs and outputs can be routed to your choice of industrial DIO or TTL DIO pins.

1.6.4 Communication Ports

Two serial communication ports round out the board’s capabilities: one meets RS-232/422/485 standards and the other is designed for I^2C.

1.6.4.1 RS-232/422/485

The serial port is software configurable to RS-232, 422, or 485. The on-board UART supports programmable baud rates from 300 baud to 2 Mbaud, character width, parity, and stop bits.

1.6.4.2 I^2C

The I^2C port is fully compliant with UM10204. The port may be configured as either a Master, Slave, or Bus Monitor running at either Standard, Fast, or Fast+, or a custom rate from 2 kHz to 100 kHz. SDA and SCL pins are pulled up to +5V TTL with built-in 4.99 kΩ resistors

1.6.5 Guardian Diagnostics

The DNx-MF-101 includes the following built-in diagnostic features:

- **Analog Inputs** - monitor PGA and report out-of-range error with every data sample
- **Analog Outputs** - monitor output voltage, supply voltage, and temperature on each channel and timestamp the start of each scan
- **Industrial Digital Outputs** - monitor output voltage and timestamp the start of each scan
- **I^2C Port** - monitor the master using the slave module

1.6.6 Isolation & Over-voltage Protection

The DNx-MF-101 offers 350 Vrms of isolation between itself and other I/O boards as well as between the I/O connections and the chassis. The analog and digital sections of this board are also isolated.

1.6.7 Environmental Conditions

Like all UEI I/O boards, the board offers operation in extreme environments and has been tested to 5g vibration, 100g shock, from -40 to +85 °C temperatures and will function at altitudes up to 70,000 feet.

1.6.8 Accessories

The DNx-MF-101 is supported by UEI's DNA-CBL-MF-1M splitter cable and DNA-STP-MF-101 screw terminal panel. The shielded cable runs the digital and analog signals through separate bundles to minimize noise. The STP board includes terminals for all I/O pins, a DB-9 connector for the serial port, an RJ-11 connector for the I^2C port, and a built-in CJC temperature sensor. For those wishing to create their own cables, all connections are through a standard 62-pin "D" connector, allowing OEM users to build custom cabling systems with off-the-shelf components.

1.6.9 Software Support

The DNx-MF-101 includes a software suite supporting Windows, Linux, QNX, VXWorks, RTX, and most other popular real-time operating systems. Windows users may use the UeiDaq Framework, which provides a simple and complete software interface to all popular programming languages and DAQ applications (e.g., LabVIEW, MATLAB). All software includes example programs that make it easy to copy-and-paste the I/O software into your applications.

1.7 Technical Specifications

The following tables list the technical specifications for the DNx-MF-101 board. All specifications are for a temperature of 25°C±5°C unless otherwise stated.

1.7.1 Analog Input

Table 1-1 Analog Input Specifications

Number of channels	16 single-ended or 8 fully differential	
Input configuration	Multiplexed	
ADC resolution	18 bits	
Sampling rate	2000 samples/second per channel	
High voltage mode	**Resolution**	**Accuracy (at 25°C)**
±80 V	610 µV	±24 mV
±20 V	153 µV	±6 mV
±5 V	38.1 µV	±2.5 mV
±1.25 V	9.54 µV	±700 µV
Input impedance	> 1.13 MΩ Diff / 1565 kΩ SE	
Input offset current	< 72 µA	
Overvoltage protection	± 100 Vdc	
Low voltage mode	**Resolution**	**Accuracy (at 25°C)**
±10 V	76.3 µV	±1.125 mV
±2.5 V	19.1 µV	±300 µV
±0.625 V	4.77 µV	±170 µV
±0.156 V	1.19 µV	±115 µV
Input impedance	> 10 MΩ	
Input offset current	±1 nA max, ±0.5 nA typical	
Overvoltage protection	± 100 Vdc	
Common mode rejection	100 dB typical (differential mode)	
Isolation	350 Vrms (analog in and out share one gnd)	

1.7.2 Analog Output

Table 1-2 Analog Output Specifications

Number of channels	2 channels
Resolution	16-bit resolution
Voltage Output mode	
Voltage output ranges	±10 V, ±5V at ±5 mA
Output accuracy	3 ppm/°C typical, 10 ppm/°C max
±10 V	±3 mV
±5 V	±1.5 mV
Output impedance	< 0.1 Ω not including any cables
Current Output mode	
Current outputs	0-20 mA, 4-20 mA, -1-22 mA
Output accuracy	3 ppm/°C typical, 10 ppm/°C max
0-20 mA	±3 µA
4-20 mA	±2.6 µA
-1-22 mA	±3.5µA
Maximum load resistance	750 Ω
Update rate	2000 updates/sec max, per channel
Settling time	100 µS to 0.03%
Isolation	350 Vrms (analog in and out share one gnd)

1.7.3 Industrial Digital I/O

Table 1-3 Industrial Digital I/O Specifications

Number of channels	16 bits
I/O direction	independently selectable per bit
Digital Input	
Input range	0-55 VDC
Input high / low voltage	Programmable from 0-55 VDC
Input impedance	> 1.1 MΩ
Input open circuit state	98 kΩ pull-up or pull-down resistors are software enabled.
Input protection	±100 VDC
Input clock rate	200 kHz
Guardian input accuracy	±275 mV (15 ppm/°C)
Input throughput	1 kHz max
Digital Output	
Configurations	Current sink/source, Ground/open, or Vcc/open (Vcc is user provided in banks of 4 bits)
Output drive	500 mA per channel, continuous
Output protection	1.25 Amp fast-blow fuse on each output
Output voltage drop	< 600 mV at 500 mA (Incl std 3' cable)
Output Off impedance	> 1.1 MΩ
Output Off leakage current	< 50 µA (with 55 V input)
Output throughput	1000 updates per second, max
PWM output	0 to 100% in 0.0015% increments (16-bit resolution)
PWM cycle rate	up to 10 kHz

1.7.4 TTL Digital I/O

Table 1-4 TTL Digital I/O Specifications

Number of channels	6 bits
I/O direction	4 bits selectable in groups of 2 1 bit input, 1 bit output
Logic level	5 V logic

1.7.5 Counter/Timer

Table 1-5 Counter/Timer Specifications

Number of counters	2
Resolution	32 bits
Max frequency	66 MHz for internal input clock 16.5 MHz for external input clock 33 MHz for outputs
Min frequency	no lower limit
Internal 66 MHz timebase	Initial accuracy: ±10 ppm Temp drift: ±15 ppm over full temp range Time drift: ±5 ppm year one, then lower
Pulse-width/period accuracy	2 internal clock cycles (30 ns) on one or multiple periods
External gate/trigger inputs	1 per counter, programmable polarity

1.7.6 Serial Port

Table 1-6 RS-232/422/485 Port Specifications

Number of Ports	1 port
Configuration	software selectable RS-232, 422 or 485
Max baud rate	RS-232: 256 kb/s, RS-422/485: 2 Mb/s
Baud rate selection	300 to 2 Mbaud, 0.01% or better accuracy
RS-232/485 transceiver	MAX3160E with fail-safe RS-485 RX term
FIFO size	2048-word TX, 2048-word RX

1.7.7 I2C Port

Table 1-7 I^2C Port Specifications

Number of Ports	1 port
Configuration	Master, Slave or Bus Monitor capability
Interface specification	Complies with UM10204
Max SCL speed	1 Mbit/S (compliant with SM: 100kb, FM: 400 kb and FM+: 1Mb
Logic Level	5V
Baud rate base clock	66 MHz, 24 MHz or PLL Based
FIFO size	Master Mode: 1024/1024 input/output Slave Mode: 512/512 input/output

1.7.8 General

Table 1-8 General and Environmental Specifications

Electrical Isolation	350 Vrms All analog signals share one ground All digital/communications signals share one ground All analog and digital signals are isolated from the chassis and all other I/O boards
Power Consumption	< 5 W (not including output loads)
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, broadband 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
Altitude	70,000 feet, maximum
MTBF	140,000 hours
Weight	5.6 oz (160 grams)

*Shock and vibration specifications assume appropriate mounting/installation.

Chapter 2 I/O Functional Descriptions

This section describes the device architecture and hardware of each of the DNx-MF-101 board's functional blocks. The following sections are provided in this chapter:

- Analog Input (Section 2.1)
- Analog Output (Section 2.2)
- Digital I/O (Section 2.3)
- Serial Port (Section 2.4)
- I2C Port (Section 2.5)
- Indicators and Connectors (Section 2.6)
- Pinout (Section 2.7)
- Wiring Guidelines (Section 2.8)

2.1 Analog Input

The DNx-MF-101 supports 8 fully differential analog input channels. As shown in **Figure 2-1**, the input lines are connected to 1/8th voltage dividers (140 kΩ/1 MΩ) which may be switched on or off. These dividers allow the board to accept input voltages up to +/- 80 V.

Each input is buffered to reduce multiplexer settling time issues and increase accuracy for high impedance sources. A multiplexer passes the inputs one by one into a programmable gain amplifier (PGA). The 18-bit A/D converter samples the multiplexed channel and performs signal averaging for further noise reduction.

If desired, each differential channel may be configured as 2 single-ended channels for a maximum of 16 single-ended channels. In single-ended mode, an on-board multiplexer connects the negative terminal of the differential A/D converter to ground. When using single-ended mode, we recommend configuring the analog input channels to use a moving average to aid in compensating for any noise that may be present. Both the UeiDaq Framework and low-level API provide support for configuring analog inputs to use moving averages.

The I/O circuitry is optically isolated from the control logic.

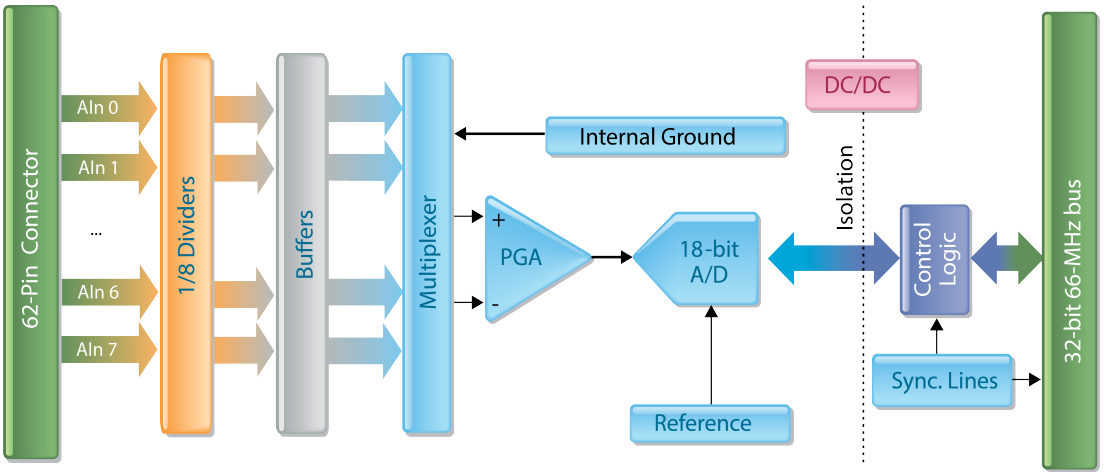


Figure 2-1 Block Diagram of DNx-MF-101 Analog Input

2.1.1 Analog Input Diagnostics

The DNx-MF-101 monitors the PGA output and reports if the currently sampled channel exceeds the input range. Over-voltage suggests that data for this sample and the next could be invalid.

2.2 Analog Output

As shown in **Figure 2-2**, the DNx-MF-101 is equipped with two analog output channels. Each channel may be independently configured to output either voltage or current through its own dynamic 16-bit D/A converter. All analog input and output channels share the same ground and same reference but are isolated from the control logic.The FPGA writes to both DACs simultaneously and the two output channels are synchronized within 1.5 µs.

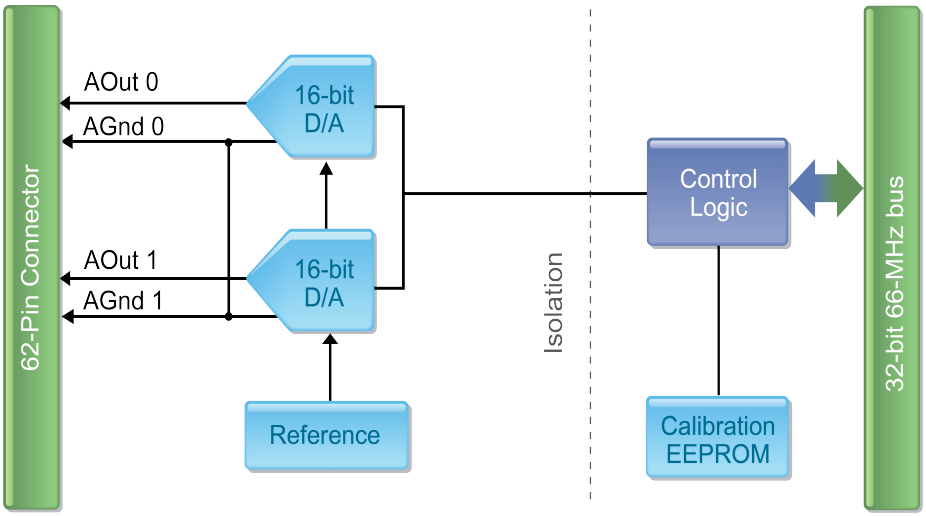


Figure 2-2 Block Diagram of DNx-MF-101 Analog Output

2.2.1 Analog Output Diagnostics

Each output channel is equipped with a diagnostic 12-bit ADC built into its DAC. The diagnostic ADC reports DAC overload and has 4 read back channels:

- DAC temperature
- Voltage on AOut
- Voltage on AGnd
- Supply voltage

In voltage output mode, the supply voltage should read approximately 15 V. In current output mode, the supply voltage is dynamically regulated to 4.95 V or (I_{OUT} × R_{LOAD} + headroom), whichever is greater. The headroom has a minimum value of 2.3 V.

2.3 Digital I/O

The DNx-MF-101 digital I/O subsystem includes 16 industrial I/O channels, 4 TTL I/O channels, a dedicated TTL input, and a dedicated TTL output. Each industrial I/O channel is independently configurable, while TTL I/O channels are configurable in pairs. All digital I/O signals are isolated from the FPGA.

2.3.1 Industrial Digital I/O

Figure 2-3 shows a simplified block diagram of the ADC-based digital I/O subsystem. DIO channels may be configured as either input or output.

Inputs are buffered to protect against input loading and simultaneously sampled at 200 kHz by 14-bit A/D converters (one ADC per DIO channel). The control logic compares the ADC voltage to user-defined High and Low thresholds and returns the digital state. Inputs may also be debounced with programmable delays. The source impedance of digital inputs should be 5k Ω or below.

NOTE: While the ADC can technically read in the DIn lines as if they were analog inputs, this is not a recommended use of these channels.

.

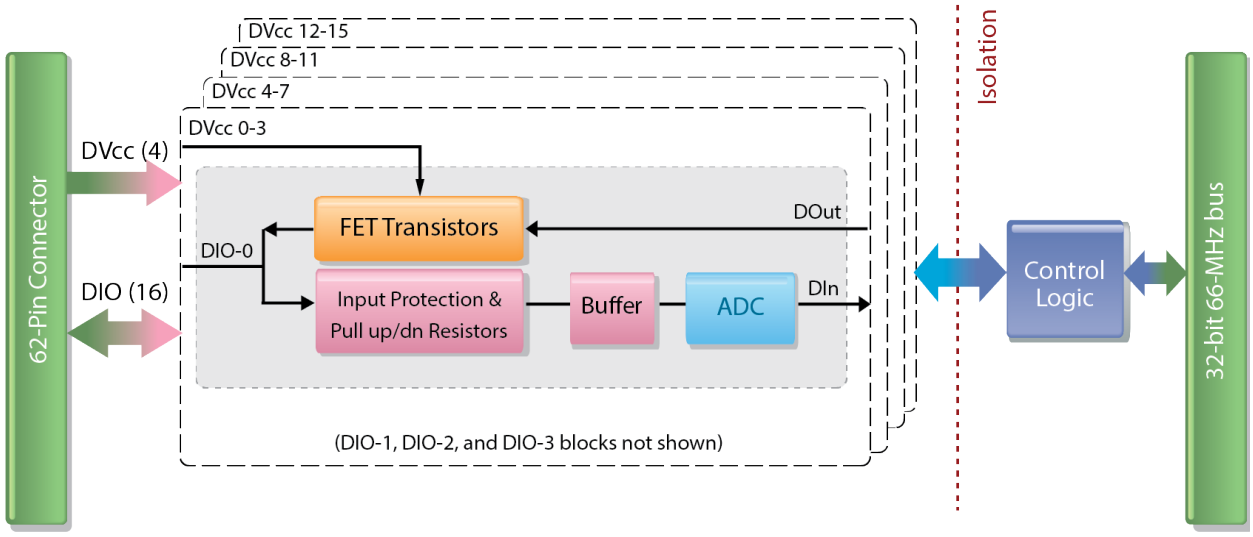


Figure 2-3 Block Diagram of DNx-MF-101 Industrial Digital I/O

Outputs are switched by a FET-based circuit (**Figure 2-4**) and require an external DC power supply. Up to 4 different +DVcc's may be supplied to the DNx-MF-101 board. Users should ensure that each +DVcc can supply enough current for all four channels it powers, up to 500 mA max/channel.

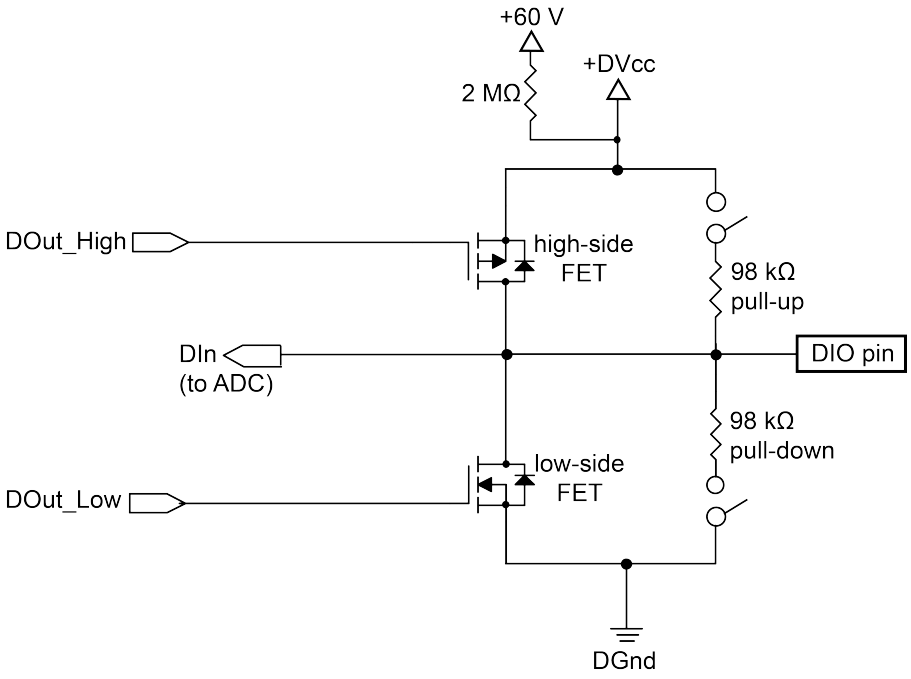


Figure 2-4 Simplified Circuit Diagram of an Industrial DIO Channel

As illustrated in **Figure 2-4**, each output is set to LOW, HIGH, or OFF by a high-side/low-side pair of FETs. When the FPGA writes a 1 on the Dout_HIGH line, the high-side FET turns on and connects the DIO pin to +DVcc (current sourcing). When a 1 is written to the DOut_LOW line, the low-side FET connects the DIO pin to DGnd (current sinking). The control logic prevents both FETs from being on currently. When both high- and low-side FETs are disabled, the pin can be used as a dedicated input.

Each pin's open-circuit state is software programmable to DVcc, Gnd, or DVcc/2. This is achieved by connecting the pin to an internal 98 kΩ pull-up resistor, 98 kΩ pull-down resistor, or both resistors respectively.

NOTE: The industrial digital output channels do NOT include built-in anti-kickback diodes. If the channel is used to source or sink an inductive load, we recommend connecting an external diode to protect the FETs against induced voltage spikes (see Section 2.8.2 for wiring information).

If +DVcc is disconnected, the positive rail is automatically pulled up to an internal +60 V supply by a 2 MΩ resistor. The internal supply prevents accidental floating inputs and allows digital inputs to work properly without a user-supplied +DVcc. A user-supplied +DVcc is only required for digital outputs.

When pulled up to the +60 V supply, an unused DIO pin will have some voltage under 60 V (varies with the number of DO pins driving HIGH). The large 2 MΩ pull-up resistance protects user equipment from this voltage. To set the unused pin to zero, you can add an external 100 kΩ pull-down resistor.

2.3.1.1 Pulse Width Modulation

The DNx-MF-101 offers built-in pulse width modulation (PWM) on industrial digital outputs. PWM mode, frequency, duty cycle, and push/pull mode are per channel configurable.

PWM modes include:

- **Continuous PWM** - The duty cycle is constant over the entire period of operation. A typical application for this feature is a dimmer for an incandescent indicator light in which the average voltage applied to a bulb is increased or decreased by varying the PWM duty cycle.
- **Soft Start** - As shown in **Figure 2-5**, a soft start increases the PWM duty cycle gradually from 0% up to the configured steady-state value. This feature is useful in preventing premature burnout of devices (such as incandescent bulbs) caused by too rapid heating on startup.
- **Soft Stop** - Soft stop is the opposite of soft start. The duty cycle decreases gradually down to 0% when the output transitions from HIGH to LOW. The typical application for soft stop mode is a soft start operation that is implemented with inverted logic.

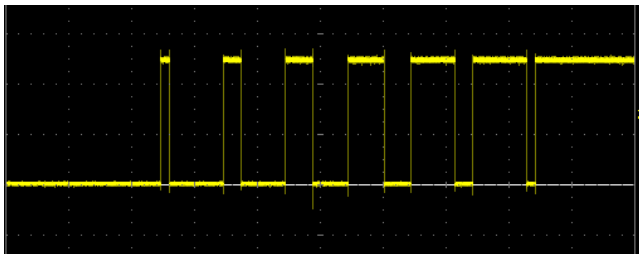
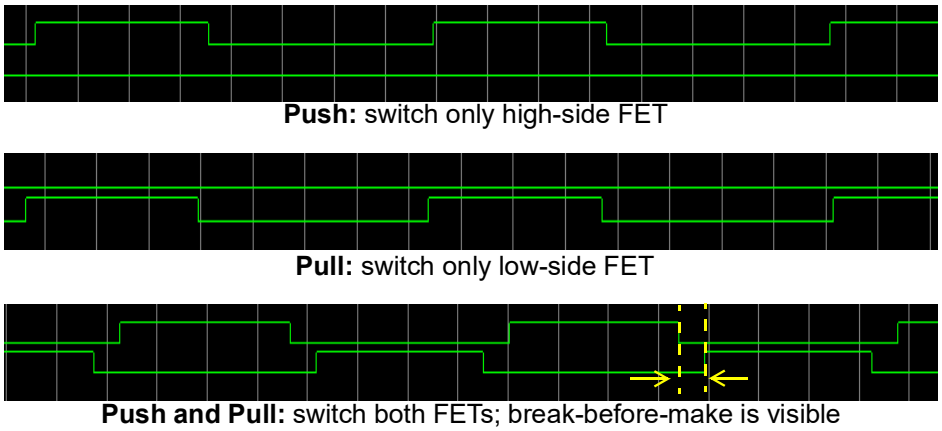


Figure 2-5 Typical PWM Soft Start cycle

A PWM output can be configured to switch one or both FETs in the channel. A break-before-make interval prevents both FETs from being on at the same time, as shown in **Figure 2-6**.



Push: switch only high-side FET

Pull: switch only low-side FET

Push and Pull: switch both FETs; break-before-make is visible

Figure 2-6 PWM Push/Pull output modes

It is also possible to generate pulse trains using the counters described in Section 2.3.3. However, the built-in PWM system is easier to use and therefore recommended for industrial digital outputs.

2.3.1.2 Digital Output Diagnostics

Because DOut and DIn share the same pin, the board can readback DOut voltage through the Din ADC. The board does not currently support output current monitoring, but it does provide over-current protection using a 1.25 A fast-blow fuse on each output channel.

2.3.2 TTL Digital I/O

The TTL bits use 5 V logic levels (an input between 2 V and 5 V is a HIGH, while a voltage below 0.8 V is a LOW). The DNx-MF-101 is capable of single read/write into the registers as well as continuous clock reads and writes. PWM signals can be generated on TTL outputs via the counter subsystem described in Section 2.3.3.

2.3.3 Counters

Industrial and TTL DIO pins may be routed to two 32-bit counters in order to perform a number of customizable operations including:

- **Timer:** count off a user-defined time interval
- **Event Counter:** count the number of rising or falling edges on a signal
- **Bin Counter:** count the number of pulses in the specified time interval
- **Pulse-Width/Period:** measure the width of the positive and/or negative parts of the input signal
- **Timed Pulse Period Measurement:** measure average frequency of incoming pulses over a user-defined time interval
- **Quadrature Decoder:** measures relative position from a quadrature encoder sensor
- **PWM Generator:** output a pulse-width-modulated waveform and update its period and duty cycle on the fly

As shown in **Figure 2-7**, each counter has three lines:

- **Input clock (CLKIN):** takes in the signal to be measured
- **Output clock (CLKOUT):** drives one or more digital output pins according to the counter's mode of operation
- **Gate/Trigger input (GATE):** takes in a gating signal, start/stop/restart trigger, or the quadrature encoder direction

Both input lines are connected to de-bouncers to eliminate unwanted spikes in the signals. The counter counts up to 2^32 and can be clocked by either **CLKIN**, a 66 MHz internal base clock, or a divided version of either clock.

NOTE: If the counter is routed to industrial digital inputs, the measurement resolution is limited by the 200 kHz DIn ADC clock rate (e.g., pulse width will be returned in 2.5 µs increments). TTL-level inputs do not use the ADC and can therefore be measured down to 15 ns.

The counter's behavior is defined according to the values of the registers shown in **Figure 2-7** and described in **Table 2-1**. Refer to Chapter 4 and Chapter 5 for information about configuring the counting modes.

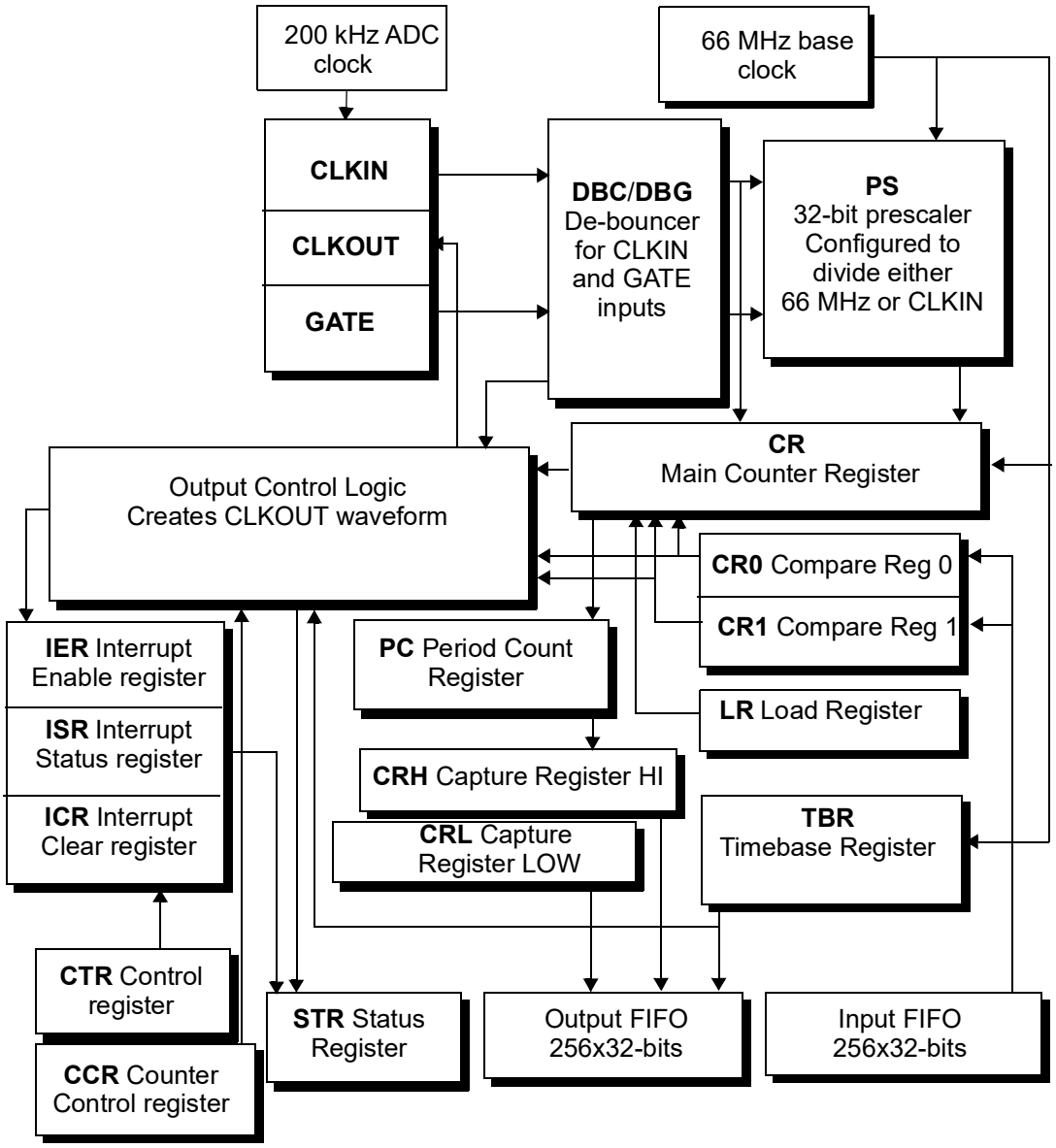


Figure 2-7 Internal Structure of DNx-MF-101 Counter

Table 2-1 DNx-MF-101 Counter Registers

Reg	Name	Description
CCR	Counter Control Register	defines the operation mode of the counter and prescaler
CR	Main Counter Register	stores the count; counts upward in all modes except for quadrature decoder mode which allows both up and down counting
CR0	Compare Register 0	defines how long CLKOUT stays low
CR1	Compare Register 1	defines how long CLKOUT stays high
CRH	Capture Register HIGH	used when the counter measures parameters of the CLKIN signal
CRL	Capture Register LOW	used when the counter measures parameters of the CLKIN signal
CTR	Control Register	enables/disables the counter, enables/disables inversion mode for I/O pins and buffered FIFO operation
ICR	Interrupt Mask Register	clears interrupt condition(s) after a CPU processes them
DBC	CLKIN De-bouncing Register	defines number of 66MHz clock cycles for which the Input Clock signal must be stable
DBG	GATE De-bouncing Register	defines number of 66MHz clock cycles for which the Gate signal must be stable
IER	Interrupt Enable Register	enables/disables interrupt generation; 16 interrupt conditions are available
ISR	Interrupt Status Register	reports status of the enabled interrupts
LR	Load Register	stores the initial value from which the counter starts counting
PC	Period Count Register	used when measuring a signal that is too fast to read every period; data from **CR** is supplied only when measured data has accumulated over N periods
STR	Status Register	reports current status of the counter operation
TBR	Timebase Register	defines the measurement time interval in certain modes

2.4 Serial Port

The DNx-MF-101 offers a fully isolated serial interface which is software-configurable as RS-232 or RS-485 (half or full-duplex). The board is also compatible with RS-422 networks when used in RS-485 full-duplex mode. A block diagram of the serial subsystem is shown in **Figure 2-8**. A MAX3160E transceiver translates voltage levels on the TX and RX lines to logical 0's and 1's. The data stream to/from the MAX3160E is controlled by an emulated UART 16550 serial controller, which reads/writes data from 2048-word FIFOs.

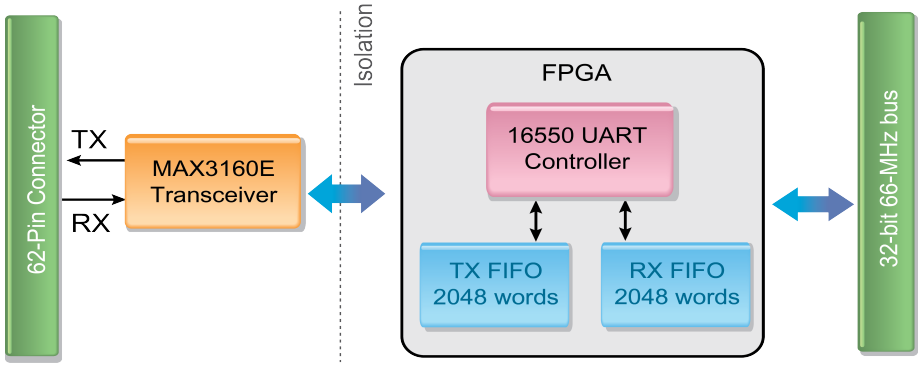


Figure 2-8 Block Diagram of DNx-MF-101 Serial Port

The remainder of this section is intended as a review of serial port concepts to supplement the programming chapters.

2.4.1 What is a Serial Port?

A serial port transfers data one bit at a time over a given line. RS-232/422/485 standards define the hardware connection between sender and receiver, such as the number of lines, the wiring scheme, and the signal's electrical characteristics. Please see Section 2.8.3 for wiring diagrams.

2.4.1.1 RS-232 Overview

An RS-232 interface provides a bidirectional, full-duplex, serial connection from 1 transmitter to 1 receiver over short distances. RS-232 requires three wires: RX, TX, and a common ground. Voltages on TX and RX are bipolar (±5V on the DNx-MF-101) and measured relative to the ground wire An example TX signal is shown in **Figure 2-9**. The EIA/TIA RS-232-C (1969) standard recommends distances of less than 50 feet at signaling rates below 19200 baud; noise becomes a problem as baud rate and line length increase.

2.4.1.2 RS-422 Overview

The RS-422 specification was designed to provide a unidirectional, full-duplex, serial connection from 1 transmitter to up to 10 receivers. RS-422 requires four wires for balanced differential signaling: Rx+, Rx-, Tx+, and Tx-. The MAX3160E transceiver drives outputs at 0V and 5V, as shown in **Figure 2-9**, and reads in voltages up to ±7V per the specification. The voltage difference between the two +/- wires represents the signal value, rather than the voltage level of just one wire. This approach eliminates a significant amount of noise and permits higher data rates and cable lengths compared to RS-232. While RS-422 was designed to support a multi-drop topology, in practice it is most commonly used as a long-distance substitute for RS-232 point-by-point topologies.

2.4.1.3 RS-485 Overview

An RS-485 interface provides a bidirectional serial connection between 32 transmitters and 32 receivers. A twisted wire pair is required for balanced differential signaling: Data+ and Data-. The MAX3160E transceiver transmits data at 0V and 5V and accepts voltages over the required common mode range of -7V to +12V. The user designs the access protocol, which usually involves one “master” device that coordinates one slave device (of 31) to transmit at a time.

2.4.2 Serial Transactions

The UART 16550 controller takes characters to be transmitted from a 2048 x 8-bit word TX FIFO and assembles them into UART frames by adding start, parity, stop bits, delays. Received characters are parsed from the frame and stored in a 2048 x 8-bit word RX FIFO.

A typical UART data frame is illustrated in **Figure 2-9**. The frame consists of:

- **Start Bit:** Signals that data bits will follow.
- **Data Bits:** Characters are sent LSB first. Default character width is 8 bits but may be reduced to 5, 6, or 7 bits.
- **Parity Bit:** Optional error correction bit that checks whether the number of 1’s in the data is odd or even.
- **Stop Bit:** Sets line to the idle state so that the next Start Bit can be seen.

The serial port on the DNx-MF-101 is capable of baud rates up to 256Kbits/s for RS-232 and 2Mbits/s for RS-422/485. This rate includes the start, parity, and stop bits.

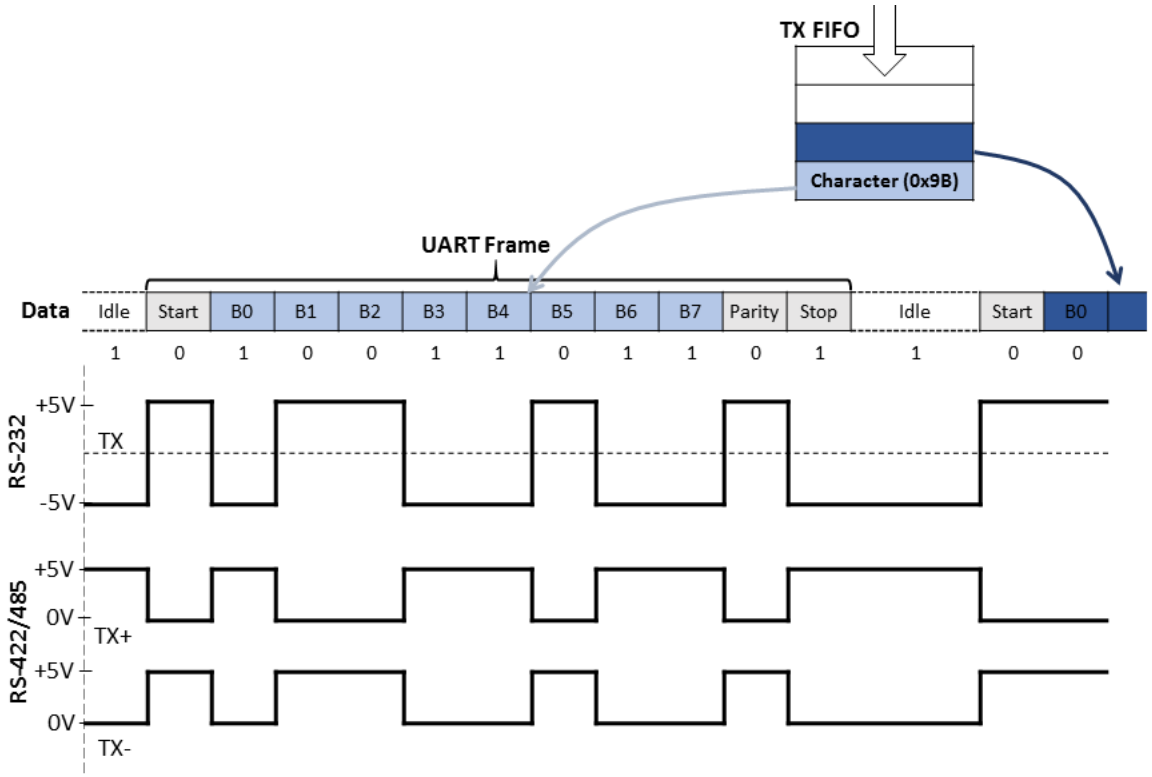


Figure 2-9 Example of Serial Transaction

2.4.3 Minor and Major Frames

UART frames, as described above, can be grouped together into a minor frame. Minor frames can be assembled into a major frame, and the transmitter can be configured to auto-repeat the major frame. The delays between when the next character, minor frame, and major frame are sent to the TX FIFO are all programmable.

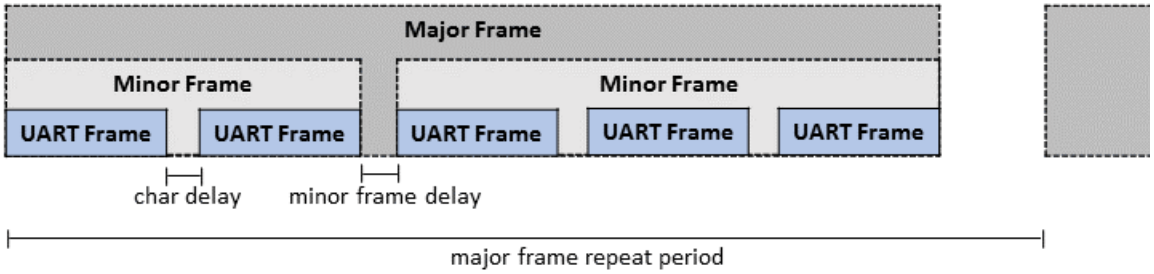


Figure 2-10 Major Frame with Variable-length Minor Frames

2.4.4 Flow Control

Flow control is useful in situations where the transmitter sends data faster than the receiver process it.The DNx-MF-101 serial port supports hardware handshaking in RS-232 mode. The Request to Send (RTS) pin is asserted when the DNx-MF-101 is ready to receive data. RTS is de-asserted when the RX FIFO has filled up to a configurable watermark level. Before sending data, the DNx-MF-101 checks if the receiver has set the Clear to Send (CTS) pin to a positive voltage level.

2.4.5 Loopback Diagnostics

When enabled, the loopback feature connects RX to TX internally and disables external signals from being generated. Software and port settings can then be tested independent of external devices and wiring.

2.5 I²C Port

The DNx-MF-101 I²C port is designed to meet UM10204 specification. The port may be configured to run as a Master, Slave, or Bus Monitor.

As shown in **Figure 2-11**, the port includes a master and slave module which are internally connected to the same serial clock line (SCL) and serial data line (SDA). Both SDA and SCL are bidirectional lines which are internally connected to the positive supply voltage via a 4.99 kΩ termination resistor. When the I²C bus is free, both lines are pulled up to HIGH. The port supports 5 V TTL logic levels.

Please refer to the UM10204 specification for details regarding I²C electrical characteristics and signal timing.

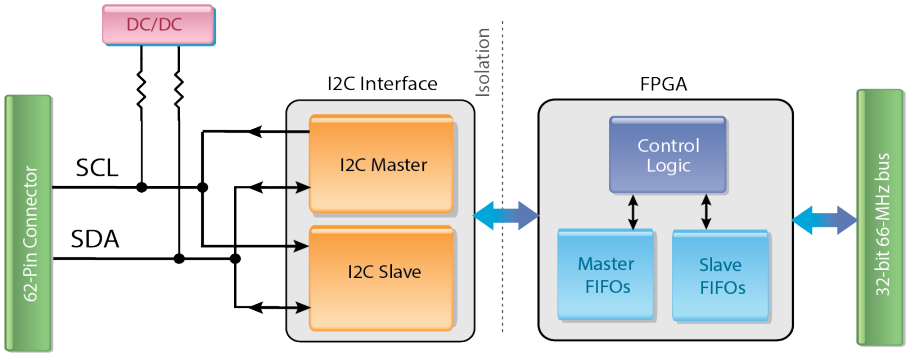


Figure 2-11 Block Diagram of DNx-MF-101 I2C Port

2.5.1 About I²C Transactions

I²C is a synchronous serial communications protocol which allows a master to control multiple slave devices on the bus. Each transaction is initiated by the master and addressed to a specific slave. A typical transaction executes as follows:

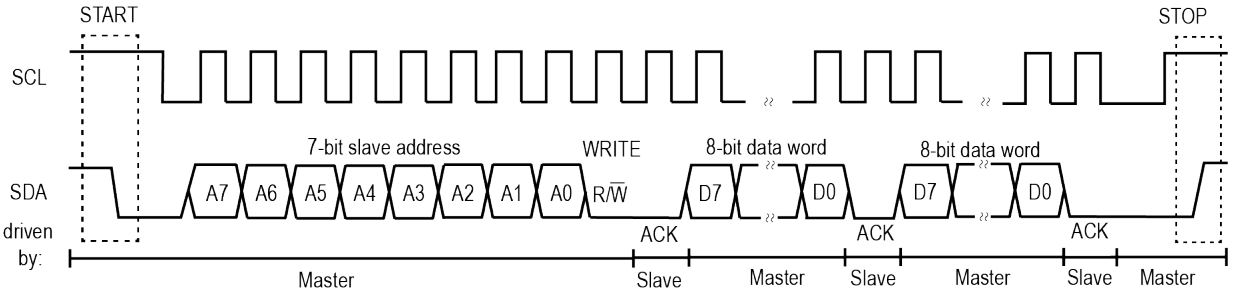


Figure 2-12 I²C Master Writing Two Bytes(7-bit Address)

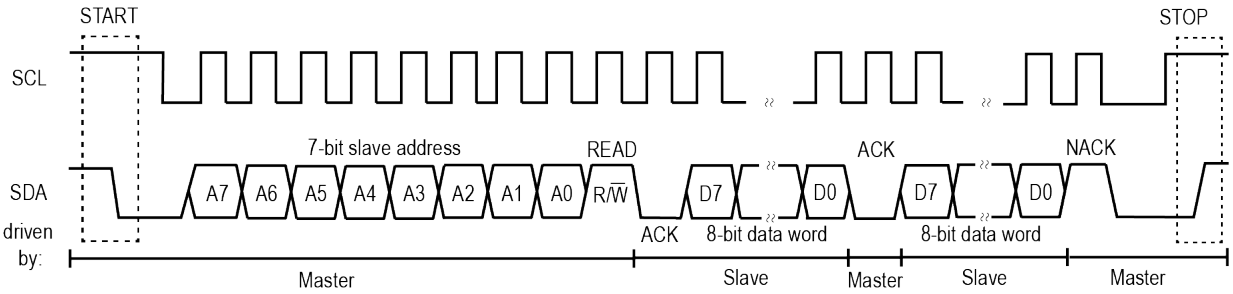


Figure 2-13 I²C Master Reading Two Bytes (7-bit Address)

1. 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).
2. The master generates clock pulses to clock the data through the transaction.
3. The master issues a 7- or 10-bit address to address a specific slave.
4. The master issues a R/$\overline{W}$ bit to indicate whether the slave is to write data or read data. It then relinquishes the SDA line to listen for an acknowledge (ACK) from the slave.
5. The addressed slave acknowledges its address by pulling the SDA line LOW. If SDA remains HIGH, the request was not acknowledged (NACK).
6. If the command was a `WRITE`, as shown in **Figure 2-12**, the master serially transmits 8-bit data words to the slave. Each byte is acknowledged by the slave.
7. If the command was a `READ`, as shown in **Figure 2-13**, the master releases the SDA line and allows the slave to transmit an 8-bit data word. The master issues an ACK after each byte until it has received the expected number of bytes. It issues NACK after the last byte.
8. When the transfer is complete, the master issues a `STOP` command (a LOW to HIGH transition on the SDA line while SCL is HIGH). A `STOP` command can be issued at any time by the master.

2.5.2 Master Module

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, plus a receiver that receives data from the SDA line.

2.5.2.1 Master Commands

The UEI API provides built-in master commands including:

- `TDELAY`: Insert a time delay (NOP command).
- `STOP`: Stop transaction once bus is available.
- `START+WRITE`: Write up to 255 bytes to the slave.
- `START+READ`: Read up to 255 bytes from the slave.
- `START+WRITE+ReSTART+READ`: Write followed immediately by a read without a stop condition in between. Read up to 255 bytes in one command.

In addition to using the built-in commands, users programming with low-level API may assemble raw command sequences and write to the bus with almost unlimited flexibility (see Chapter 5).

2.5.2.2 Master Transmitter

The master module stores commands and outgoing data in the 1024 x 32-bit word master TX FIFO. The built-in `WRITE` command packs 3 data bytes into each word. Therefore, after accounting for command-specific words, the master TX FIFO can hold roughly 3000 data bytes.

2.5.2.3 Master Receiver

The master module stores incoming data from the slave in the 1024 x 9-bit word master RX FIFO. Each 9-bit word includes the 8-bit data word and a STOP bit in its MSB. The STOP bit is set for the last word received by the `READ` command.

Example:

In this example, the master requests 4 bytes of data from the slave TX FIFO (0xaa, 0xbb, 0xcc, 0xdd). This is the output after reading from the master RX FIFO:

```
Master: received=4 available=0
[0]=aa [1]=bb [2]=cc [3]=1dd
```

2.5.2.4 Multi-Master Mode

The DNx-MF-101 master supports 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.

2.5.3 Slave Module

An I^2C slave includes a receiver that reads master commands and data, as well as a transmitter that sends data in response to a master request. Generally the DNx-MF-101 slave module is used to emulate a device on the bus for some external master, to test master software, or to monitor bus conditions. The slave may be configured with either a 7-bit or 10-bit address.

2.5.3.1 Slave Transmitter

Upon receiving a WRITE command, the slave transmitter serially outputs data bytes from a 512 x 8-bit word TX FIFO. In addition, users may preload a 32-bit TX data padding register with 4 bytes of data. If the slave TX FIFO is empty, the slave sends the TX register data on repeat until new data is available in the FIFO.

Example:

In this example, the slave TX FIFO contains two data bytes (0xaa and 0xbb) and the slave TX register was loaded with 0x12345678. The master requests 8 bytes and sees the following data in the master RX FIFO:

```
Master: received=8 available=0
[0]=aa [1]=bb [2]=56 [3]=78 [4]=12 [5]=34 [6]=56 [7]=178
```

(As described in Section 2.5.2.3, the 1 in front of the last word indicates the end of the read.)

2.5.3.2 Slave Receiver

Upon receiving a READ command, the slave receiver stores incoming data in a 512 x 12-bit word RX FIFO. By default, each 12-bit word includes the bus condition in the upper 4-bits, as shown in **Figure 2-14**. The bus condition, as listed in **Table 2-2**, includes the master command and whether the data or address was acknowledged.

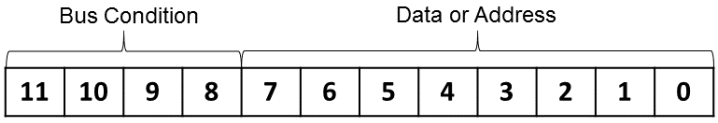


Figure 2-14 Slave RX Data Format

Table 2-2 I2C Bus Conditions

Value	Description	Bits 7...0
0	reserved	n/a
1	`START` transaction	n/a
2	`ReSTART` transaction	n/a
3	`STOP` transaction	n/a
4	address was ACK	slave address + R/$\overline{W}$ bit
5	address was NACK	slave address + R/$\overline{W}$ bit
6	data was ACK	data
7	data was NACK	data
8	NACK to indicate last byte	last data byte in transaction
9	clock stretching error	n/a

Example:

In this example, the slave address is 0x2A, and it was written with 4 bytes of data (0x01, 0x02, 0x03, 0x04). Here is the output after executing this transaction with UEI's low-level example code.

```
Slave: received=7 available=0
[0]=100 [1]=454 [2]=601 [3]=602 [4]=603 [5]=804 [6]=300
```

The received words break down as follows:

- `100`: 1 | 00000000 → `START` | no data
- `454`: 4 | 0101010 | 0 → ACK | slave address | WRITE command
- `601`: 6 | 00000001 → ACK | first piece of data
- `602`: 6 | 00000010 → ACK | 2nd piece of data
- `603`: 6 | 00000011 → ACK | 3rd piece of data
- `804`: 8 | 00000100 → all data received | 4th piece of data
- `300`: 3 | 00000000 → `STOP` | no data

Suppressing Bus Conditions

The slave can be configured to store only data bytes in order to conserve FIFO space (only supported in low-level API). Here is the output of the previous example when bus conditions are suppressed.

```
Slave: received=4 available=0
[0]=601 [1]=602 [2]=603 [3]=804
```

2.5.3.3 Clock Stretching

The DNx-MF-101 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, the slave has the capability of forcing the master into a wait state by holding the SCL line LOW until ready for another byte of data.

2.5.3.4 Slave as a Bus Monitor

When the I^2C port is running in Master Mode, the slave module may be configured as a Bus Monitor for diagnostic purposes. The Bus Monitor slave does not respond to the master; its purpose is to read all activity on the I^2C bus including data and bus conditions (Section 2.5.3.2). The received data is stored in the slave module's 512 x 12-bit word RX FIFO.

2.5.4 Loopback Testing

Because the board's master and slave modules are always internally connected, master software can be easily tested without needing to connect another slave device. The internal slave module is fully functional (i.e. can send and receive data to the master, ACK, and stretch the clock).

2.6 Indicators and Connectors

Figure 2-15 shows the locations of the LEDs and connectors on the DNx-MF-101. The LED indicators are described below in Table 2-3.

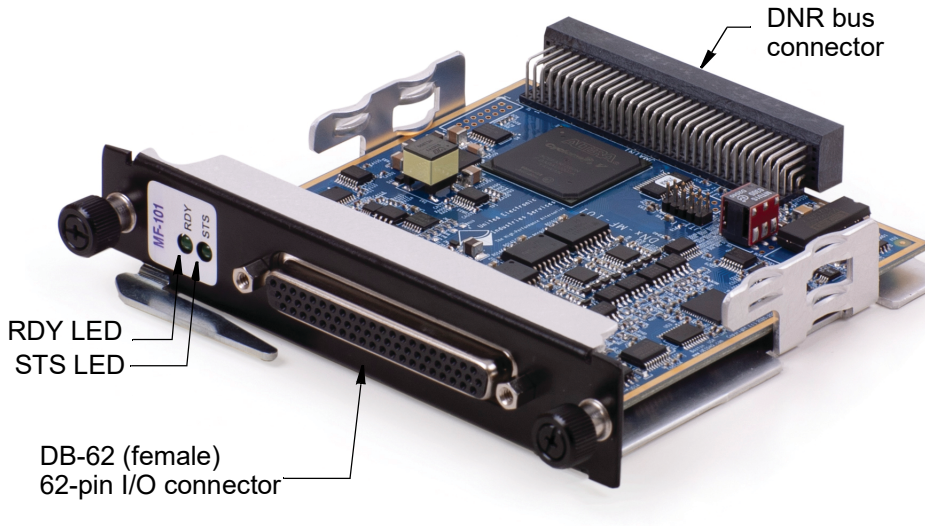


Figure 2-15 Photo of DNR-MF-101 Board

Table 2-3 LED Indicators

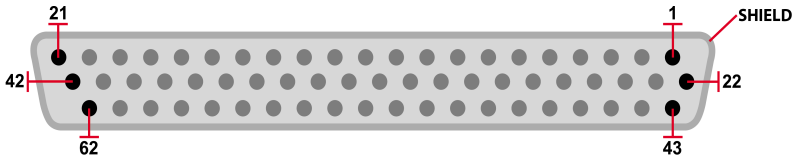
LED Name	Description
RDY	READY: board is powered up and operational
STS	STATUS: **OFF**: Configuration mode (e.g. configuring channels, running in Point-by-Point mode) **ON**: Operation mode (e.g. running in DMap or VMap mode)

2.7 Pinout

Figure 2-16 illustrates the pin configuration for the DNx-MF-101 board. Connections are made through a standard DB-62 female connector.

Signals are isolated in three groups:

- Analog I/O (in blue): AIn returns on AGnd, AOut 0 returns on AGnd 0, and AOut 1 returns on AGnd 1. Refer to **Table 2-4**.
- Industrial DIO (in red): referenced to DGnd. Refer to **Table 2-5**.
- TTL DIO, I^2C, and Serial (in black): referenced to Gnd. Refer to **Table 2-6**.



Pin	*Signal*	*Pin*	*Signal*	*Pin*	*Signal*
1	RTS232/TX485+	22	TX232/TX485-	43	CTS232/RX485-
2	I2C SCL	23	Gnd	44	RX232/RX485+
3	I2C SDA	24	+5V-TTL	45	Trig Out
4	DIO-02	25	DGnd	46	Trig In
5	DIO-00	26	DV 0-3	47	DIO-03
6	DIO-06	27	DGnd	48	DIO-01
7	DIO-04	28	DV 4-7	49	DIO-07
8	DIO-10	29	DGnd	50	DIO-05
9	DIO-08	30	DV 8-11	51	DIO-11
10	DIO-14	31	DGnd	52	DIO-09
11	DIO-12	32	DV 12-15	53	DIO-15
12	TTL 0	33	Gnd	54	DIO-13
13	TTL 2	34	Gnd	55	TTL 1
14	AGnd 1	35	AOut 1	56	TTL 3
15	AGnd 0	36	AOut 0	57	AIn 3/1-
16	AIn 1/0-	37	AIn 0/0+	58	AIn 2/1+
17	AIn 5/2-	38	AIn 4/2+	59	AGnd
18	AIn 7/3-	39	AIn 6/3+	60	AGnd
19	AIn 9/4-	40	AIn 8/4+	61	AIn 11/5-
20	AIn 13/6-	41	AIn 12/6+	62	AIn 10/5+
21	AIn 15/7-	42	AIn 14/7+		

Figure 2-16 Pinout Diagram for DNx-MF-101

No Hot Swapping!

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.

UEI's DNA-CBL-MF-1M cable is designed to ensure good noise performance (see Section A.1). If you design your own cables, we recommend separating the three isolated groups (analog I/O, industrial DIO, and TTL-level DIO) using dedicated wiring and shielding.

Table 2-4 Analog I/O Pin Descriptions

	Pin Name	Pin #	Description
Analog In	AGnd	59-60	Ground for analog inputs.
	AIn n/m+	37-42, 58, 62	Single-ended channel n or positive terminal of differential analog input channel m. Ground any unused pins.
	AIn n/m-	16-21, 57, 61	Single-ended channel n or negative terminal of differential analog input channel m. Ground any unused pins.
Analog Out	AOut n	35-36	Signal pin for analog output channel n.
	AGnd n	14-15	Ground for analog output channel n. AGnd, AGnd 0, and AGnd 1 are internally connected, but AGnd 0/1 are matched to AOut 0/1 respectively on the PCB to minimize noise and voltage drops across the outputs.

Table 2-5 Industrial Digital I/O Pin Descriptions

	Pin Name	Pin #	Description
Industrial DIO	DIO-n	4-11, 47-54	Signal pin for FET-based industrial digital I/O channel n.
	DV n-m	26, 28, 30, 32	User-supplied Vcc for DIO channels n-m. Up to 4 different Vcc's can be supplied to the port in blocks of 4 channels.
	DGnd	25, 27, 29, 31	Ground for industrial DIO port.

Table 2-6 Logic-level Digital I/O Pin Descriptions

	Pin Name	Pin #	Description		
TTL DIO	TTL n	12-13, 55-56	Signal pin for logic-level digital I/O channel n.		
	Trig In	46	An additional TTL input line or as a start trigger for the layer.		
	Trig Out	45	An additional TTL output line or to signal that the layer has been started.		
	+5V-TTL	24	Provides a constant +5 V with max output current 20 mA.		
I^2C	I2C SCL	2	Clock line for the I^2C port.		
	I2C SDA	3	Data line for the I^2C port.		
Serial			**RS-232**	**RS-422 full duplex**	**RS-485 half-duplex**
	RTS232/TX485+	1	Request to Send (RTS)	Send (Tx+)	Data (+)
	TX232/TX485-	22	Send (Tx)	Send (Tx-)	Data (-)
	CTS232/RX485-	43	Clear to Send (CTS)	Receive (Rx-)	n/a
	RX232/RX485+	44	Receive (Rx)	Receive (Rx+)	n/a
	Gnd	23, 33-34	Ground for TTL DIO, I^2C, and Serial.		

2.8 Wiring Guidelines

The following wiring schemes are recommended when connecting external devices to the DNx-MF-101.

2.8.1 Analog Input Wiring

The recommended approach for analog input wiring depends on if the signal source is grounded or floating. Grounded signals are connected to the earth, such as signal generators or an RTD bridge circuit powered by a desktop power supply. Floating signals are isolated from the earth; examples include thermocouples, batteries, or instruments with isolated outputs.

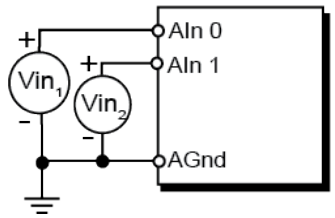
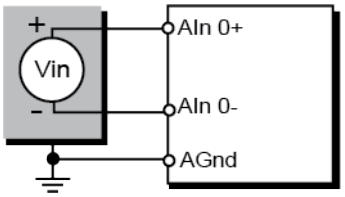
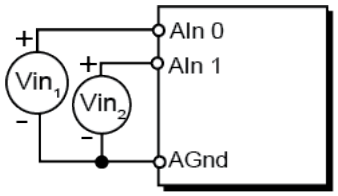
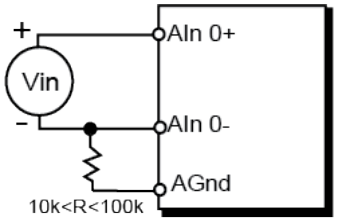
Signal Type	Single-Ended Mode	Differential Mode
Grounded		
Floating		

Figure 2-17 Analog Input Wiring

2.8.1.1 Grounded Signals

As shown in Figure 2-17, all grounded signals should have the signal source ground wired directly to AGnd on the DNx-MF-101. All AIn pins are measured relative to the same AGnd. In differential mode, the AIn+ and AIn- inputs are referenced to AGnd and then subtracted to remove voltages common to both channels.

2.8.1.2 Floating Signals

Generally speaking, floating differential inputs should have AIn- connected to AGnd via a resistor. If there is no connection to AGnd, the input voltages may float to a value that exceeds the amplifier's common mode range.

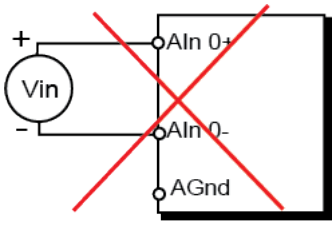


Figure 2-18 Improper Analog Input Wiring

A resistor between 10 kΩ < R < 100 kΩ is small enough to provide a path to ground for input bias current, while large enough to allow AIn- to float relative to the voltage reference. The external resistor may be disregarded if the 1/8th divider is turned ON; this scales down input voltages to be safely within the common mode range.

NOTE: Unused AIn pins should be tied to ground. This can be done internally by enabling the 1/8th divider on unused channels. Disconnected AIn pins will cause the PGA to saturate, which can lead to incorrect readings on subsequent channels in the multiplexer scan list. Other unused pins on the board may be left disconnected.

2.8.2 Industrial Digital Output Wiring

When using the industrial digital output subsystem, ensure that DVcc is connected to the user's power supply (0-55VDC). A disconnected DVcc will not damage the DNx-MF-101 but may cause unexpected digital input readings as the outputs switch ON/OFF.

A load may be wired to the output in any of the following configurations:

- **Push Mode:** DNx-MF-101 acts as a switch between DVcc and the output, sourcing current to the load when the switch is on. An example circuit is shown in **Figure 2-19**a.
- **Pull Mode:** DNx-MF-101 acts as a switch between the output pin and DGnd, sinking current from the load when the switch is on. An example circuit is shown in **Figure 2-19**b.
- **Push-Pull Mode:** DNx-MF-101 connects the output to either DVcc or DGnd, never both at the same time. An example dual-channel circuit is shown in **Figure 2-19**c. Current flows through the solenoid when one channel is set HIGH and the other channel is set LOW. The current is easily reversed by inverting the outputs.

Note that the diagrams in **Figure 2-19** include an optional external anti-kickback/flyback diode. UEI recommends adding the diode when driving inductive loads such as relays or solenoids. Without the diode, a large voltage spike can occur across the inductive load when its supply current is suddenly shut off, potentially damaging the FET switch inside the DNx-MF-101. The anti-kickback diode provides an alternate path for the current and clamps the voltage spike to a safe value.

The diode in Push Mode or Pull Mode can be a general purpose diode rated to handle the steady-state current through the inductor and the desired switching speed. Connect the Push Mode or Pull Mode diode parallel to the load.

In Push-Pull Mode, we suggest using a bidirectional transient-voltage-suppression (TVS) diode such as the P6KE68CA. Connect the TVS diode from the FET line to Gnd.

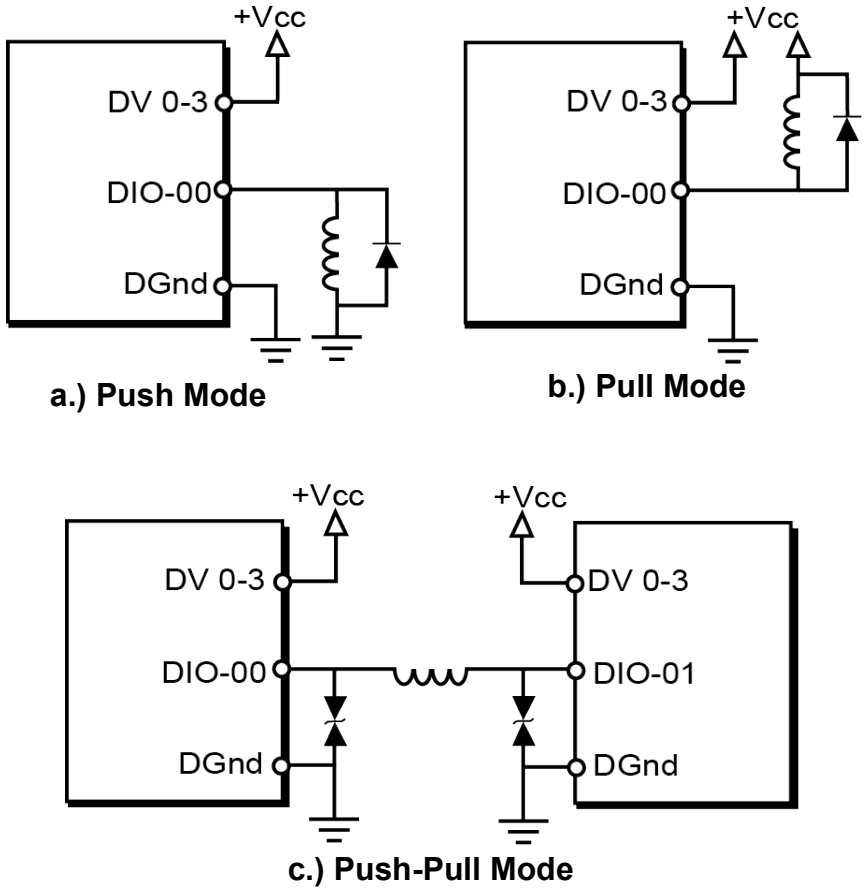


Figure 2-19 Industrial Digital Output Wiring

2.8.3 Serial Port Wiring

The DNx-MF-101 may be wired according to either RS-232, RS-422, or RS-485 standards.

2.8.3.1 RS-232

In **Figure 2-20**, the DNx-MF-101 is wired to an external RS-232 device with optional CTS and RTS lines for flow control. All lines are measured relative to Gnd.

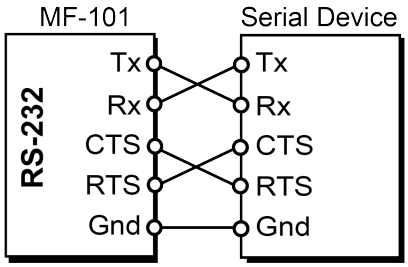


Figure 2-20 RS-232 Wiring

2.8.3.2 RS-422/485 Full Duplex

In **Figure 2-21**, the board is connected as a master in a RS-485 full duplex network. This configuration is also compatible with RS-422. Because signals are measured differentially, the + and - wires are twisted together (e.g., Rx+ and Rx-) so that noise affects the pair equally. The far ends of the cables typically require a termination resistor, shown as 120 Ω resistors in **Figure 2-21**. Otherwise, signal reflections off of the unterminated ends could interfere with the incoming signal and corrupt the data. The DNx-MF-101 provides an on-chip 91 Ω terminator that can be enabled for RS-422/485 modes.

As usual, Gnd should be connected to the reference of each external device.

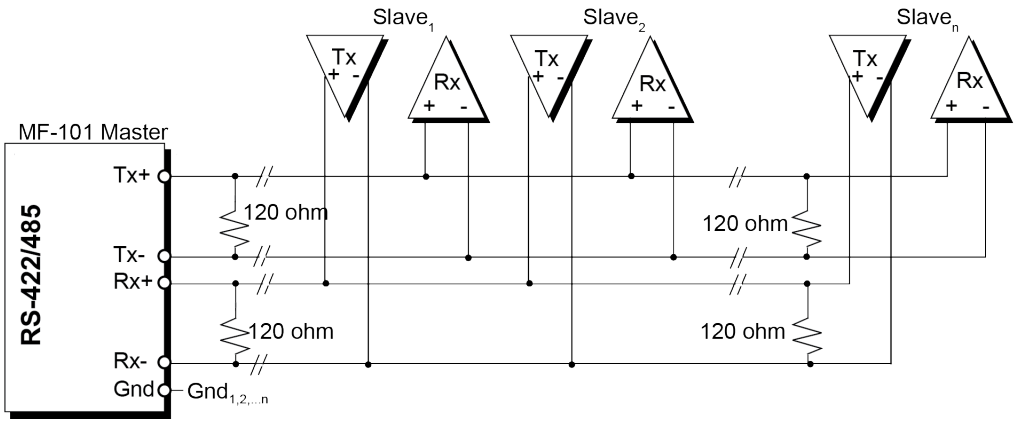


Figure 2-21 RS-422 and RS-485 Full Duplex Wiring

2.8.3.3 RS-485 Half Duplex

Figure 2-22 shows the wiring for a RS-485 half-duplex network. In RS-485 half-duplex mode, the Rx+ and Rx- pins on the DNx-MF-101 are left open because Tx and Rx are connected internally. If an external device on the network does not have an internal Tx/Rx connection, Tx+ should be wired to Rx+ and Tx- wired to Rx-. This external Tx/Rx wiring is not required on the DNx-MF-101. As with full-duplex mode, the wire pair should be twisted together and termination resistors added as needed.

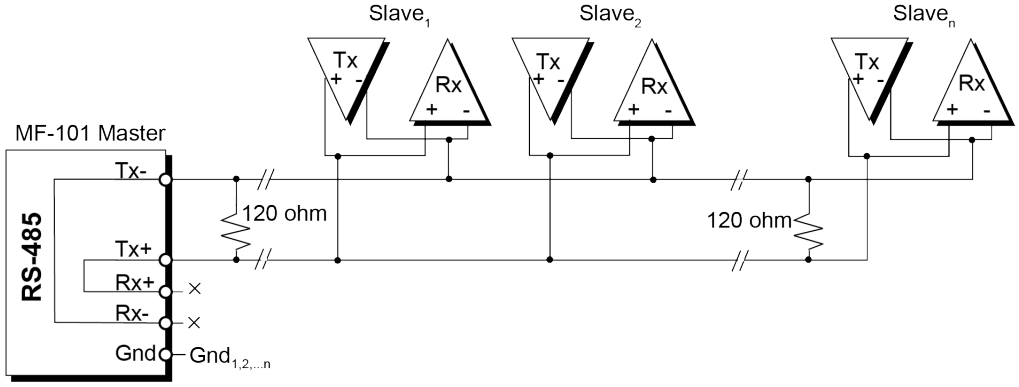


Figure 2-22 RS-485 Half Duplex Wiring

2.8.4 I2C Port Wiring

Figure 2-23 shows an example I^2C network with external 2 kΩ pull-up resistors. The external pull-up resistors are optional depending on your application. At low baud rates, the built-in 4.99 kΩ resistors in the DNx-MF-101 are typically strong enough to restore the signal to logical HIGH before the line is driven LOW.

At fast baud rates (i.e. 400 kbaud and 1 Mbaud), we recommend adding external pull-up resistors to the far ends of the cable. The resistors connect between each signal line and +5V TTL as shown in **Figure 2-23**. The choice of resistor is application-specific and depends on factors such as the cable length and the total bus capacitance. A lower resistance value increases the rise speed but also increases power consumption.

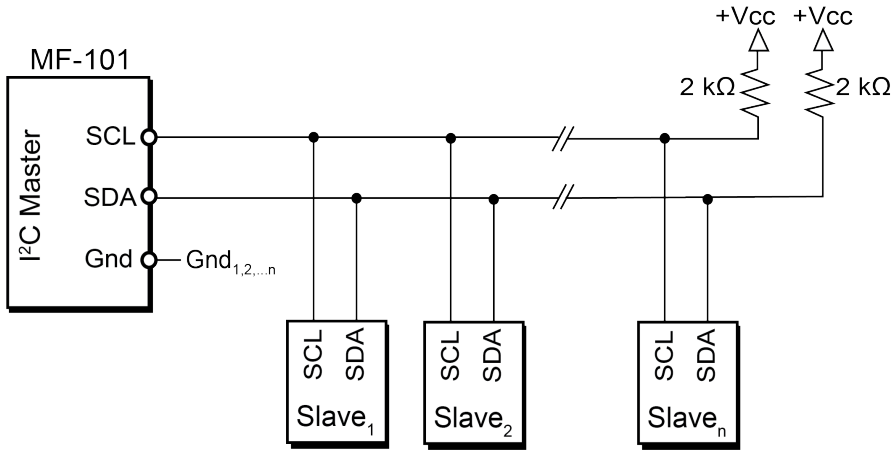


Figure 2-23 I2C Wiring

Chapter 3 PowerDNA Explorer

This chapter provides the following information about exploring the DNx-MF-101 with the PowerDNA Explorer application.

- Introduction (Section 3.1)
- Analog Input (Section 3.2)
- Analog Output (Section 3.3)
- Industrial Digital Input (Section 3.4)
- Industrial Digital Output (Section 3.5)
- RS-232/422/485 Port (Section 3.6)
- I2C Port (Section 3.7)
- Counter/Timer (Section 3.8)
- Logic-Level DIO (Section 3.9)

3.1 Introduction

PowerDNA Explorer is a GUI-based application for communicating with your RACK or Cube system. You can use it to start exploring a system and individual boards in the system. PowerDNA Explorer can be launched from the Windows startup menu:

Start » All Programs » UEI » PowerDNA » PowerDNA Explorer

The DNx-MF-101 is supported in PowerDNA version 5.0.0.29+.

When using PowerDNA Explorer to configure DNx-MF-101 boards, resetting the IOM or changing the DNx-MF-101 configuration outside of PowerDNA Explorer (e.g., via C code or Labview) is not recommended; PowerDNA Explorer will not display changed parameters until **Scan Network** or **Reload Configuration** is clicked again (see **Figure 3-1** below for button locations).

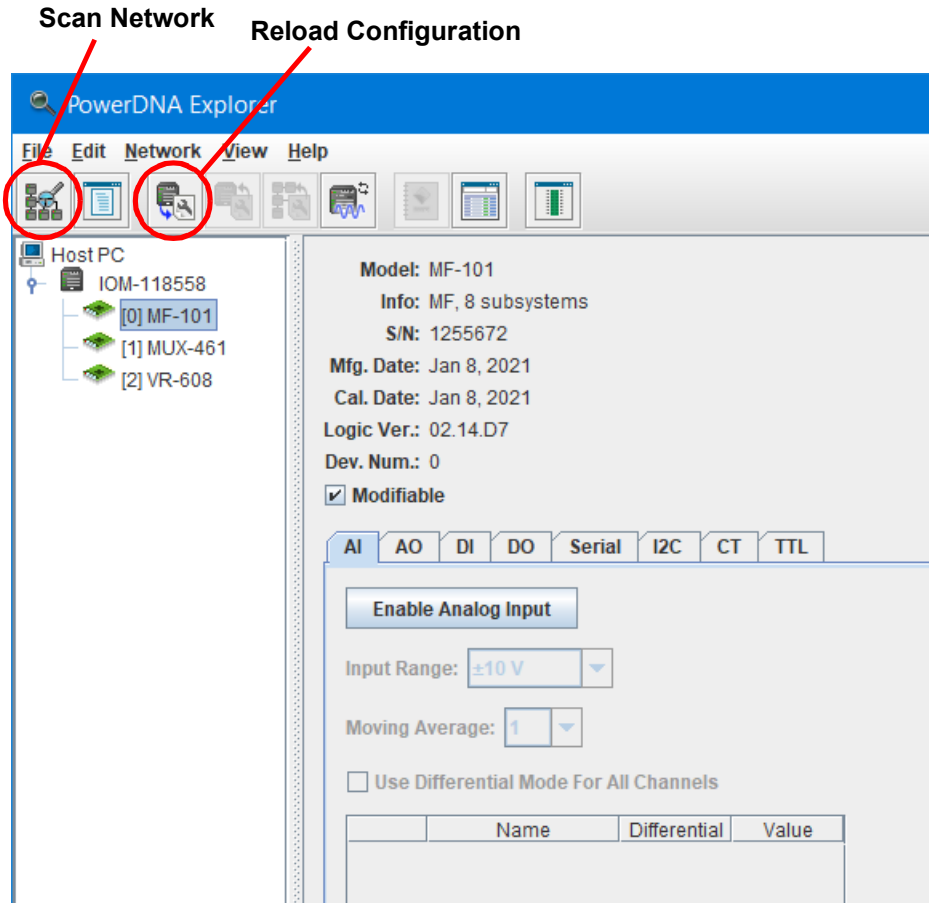


Figure 3-1 PowerDNA Explorer for DNx-MF-101

When the DNx-MF-101 is selected in the left-hand panel, the right-hand panel contains a tab for each subsystem:

- **AI**: read analog inputs
- **AO**: configure analog outputs and read diagnostic ADCs
- **DI**: read digital inputs and diagnostic ADCs
- **DO**: configure industrial digital outputs, including PWM
- **Serial**: send and receive RS-232/422/485 messages
- **I2C**: send I^2C master commands, write data to the slave TX FIFO, and read data received by the master or slave
- **CT**: configure counter/timer sources and counting modes
- **TTL**: configure TTL digital outputs and read input port

NOTE: PowerDNA Explorer only supports basic DNx-MF-101 functionality, and only one subsystem can be active at any given time. Refer to Chapter 4, “Programming with High-level API” or Chapter 5, “Programming with Low-level API” in order to access additional features and to use multiple subsystems simultaneously.

3.2 Analog Input

To explore the analog input subsystem, select the AI tab (**Figure 3-2**) and click the **Enable Analog Input** button.

3.2.1 Configure AI Subsystem

The following settings apply to all 16 analog input channels:

- **Input Range**: programs the gain and voltage divider to achieve the selected range (refer to **Table 4-2**).
- **Moving Average**: sets the number of samples used for the moving average. You must store the configuration for the new moving average to take effect. To save the configuration, click **Store Configuration**.
- **Use Differential Mode for All Channels**: configures all channels to differential mode.

3.2.2 Read AI Data

To start data acquisition, click the **Read Input Data** button. The channel table contains the following columns:

- **AInX**: read-only display of the channel number.
- **Name**: a name or note that you wish to give to the channel.
- **Differential**: sets the individual channel to differential mode. For example, in **Figure 3-2** channels AIn0 and AIn1 read differential data from pins AIn0+/0 and AIn1+/-, while channels AIn2:13 read single-ended data from pins AIn4:15.
- **Value**: displays the analog input data in volts.

NOTE: If the range is set to [-10, 10], [-2.5, 2.5], [-0.625, 0.625], or [-0.15625, 0.15625] (i.e., divider is disabled), ensure that all unused AIn pins are wired to AGnd.

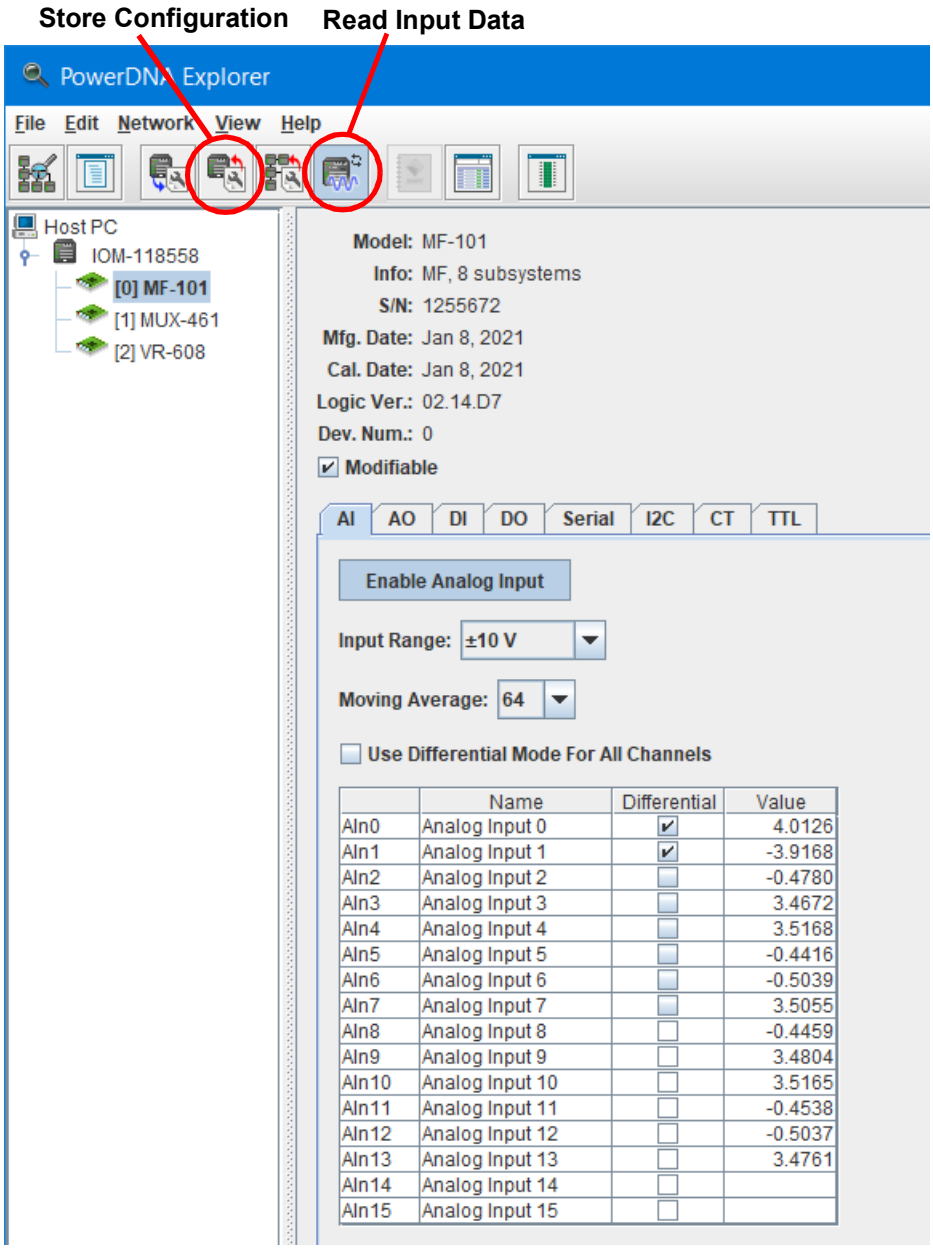


Figure 3-2 PowerDNA Explorer AI Tab

3.3 Analog Output

To explore the analog output subsystem, select the AO tab and click the **Enable Analog Output** button.

3.3.1 Write AO Data

The AO Output subtab (**Figure 3-3**) contains the following:

- **Output Range**: sets the voltage or current range for both channels. When you select a new range, the output value automatically reconfigures to midrange.
- **AOutX**: read-only display of the channel number.
- **Name**: a name or note that you wish to give to the channel.
- **Value**: slider and numeric text field for setting the voltage or current of the corresponding output channel. The valid value range is shown in the **Output Range** display. The output value is written instantaneously when the slider is released or after pressing **Enter** in the numeric field.

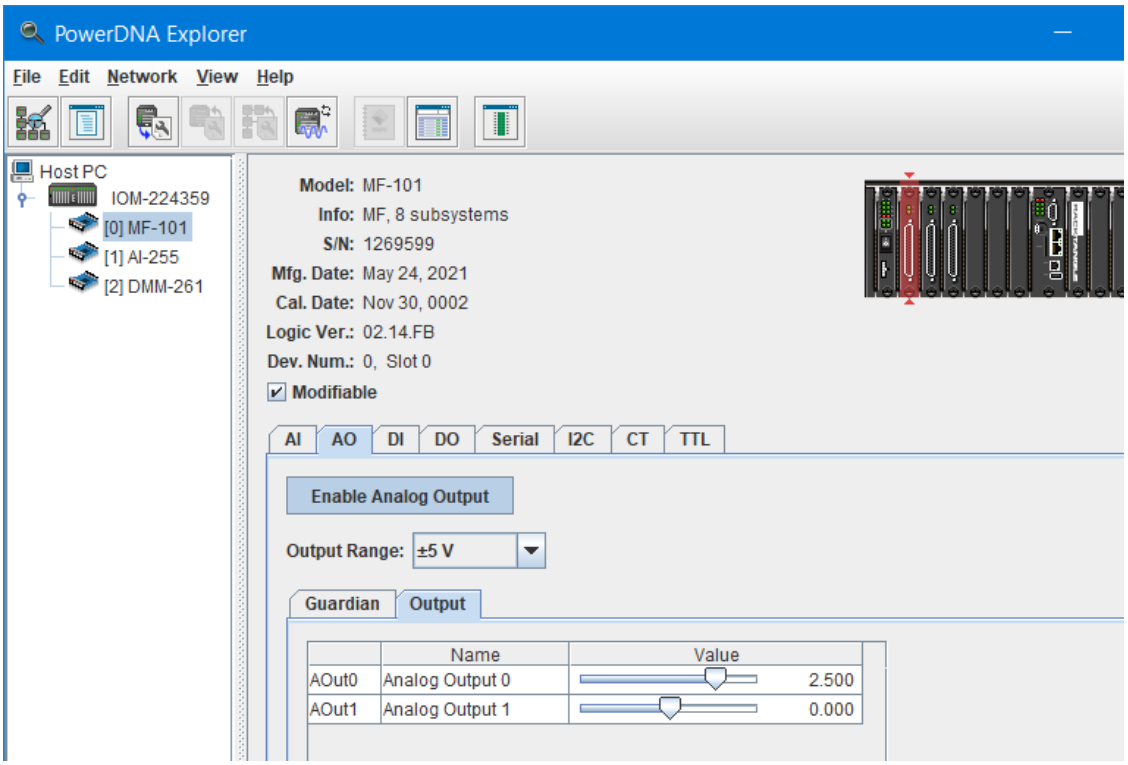


Figure 3-3 PowerDNA Explorer AO Tab, Output Subtab

3.3.2 Read AO Guardian Diagnostics

The AO Guardian subtab (**Figure 3-4**) provides access to diagnostic ADC data for both output channels. To read the Guardian diagnostic values, click the **Read Input Data** button.

The Guardian subtab contains the following columns:

- **AInX**: read-only display of the analog output channel number.
- **Name**: a name or note that you wish to give to the channel.
- **Temp (C)**: DAC temperature
- **Vsense+ (V)**: Voltage on AOutX
- **Vsense- (V)**: Voltage on AGndX
- **Vdpc+ (V)**: Supply voltage

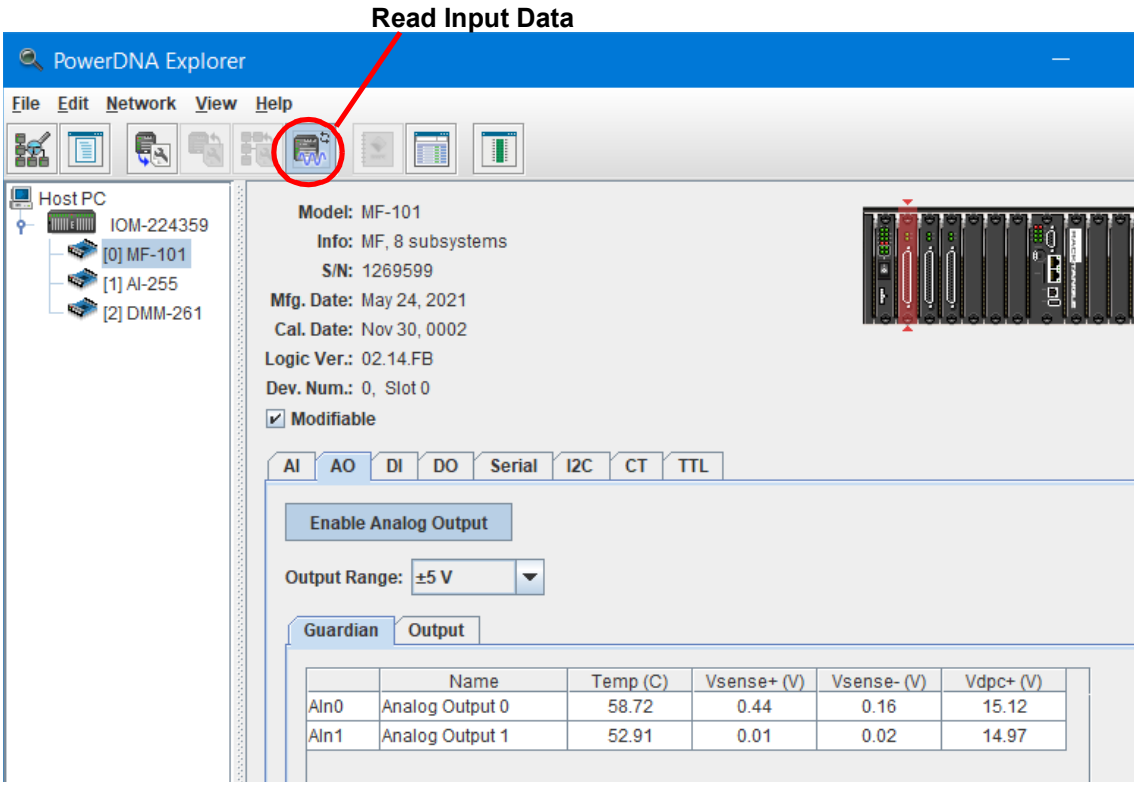


Figure 3-4 PowerDNA Explorer AO Tab, Guardian Subtab

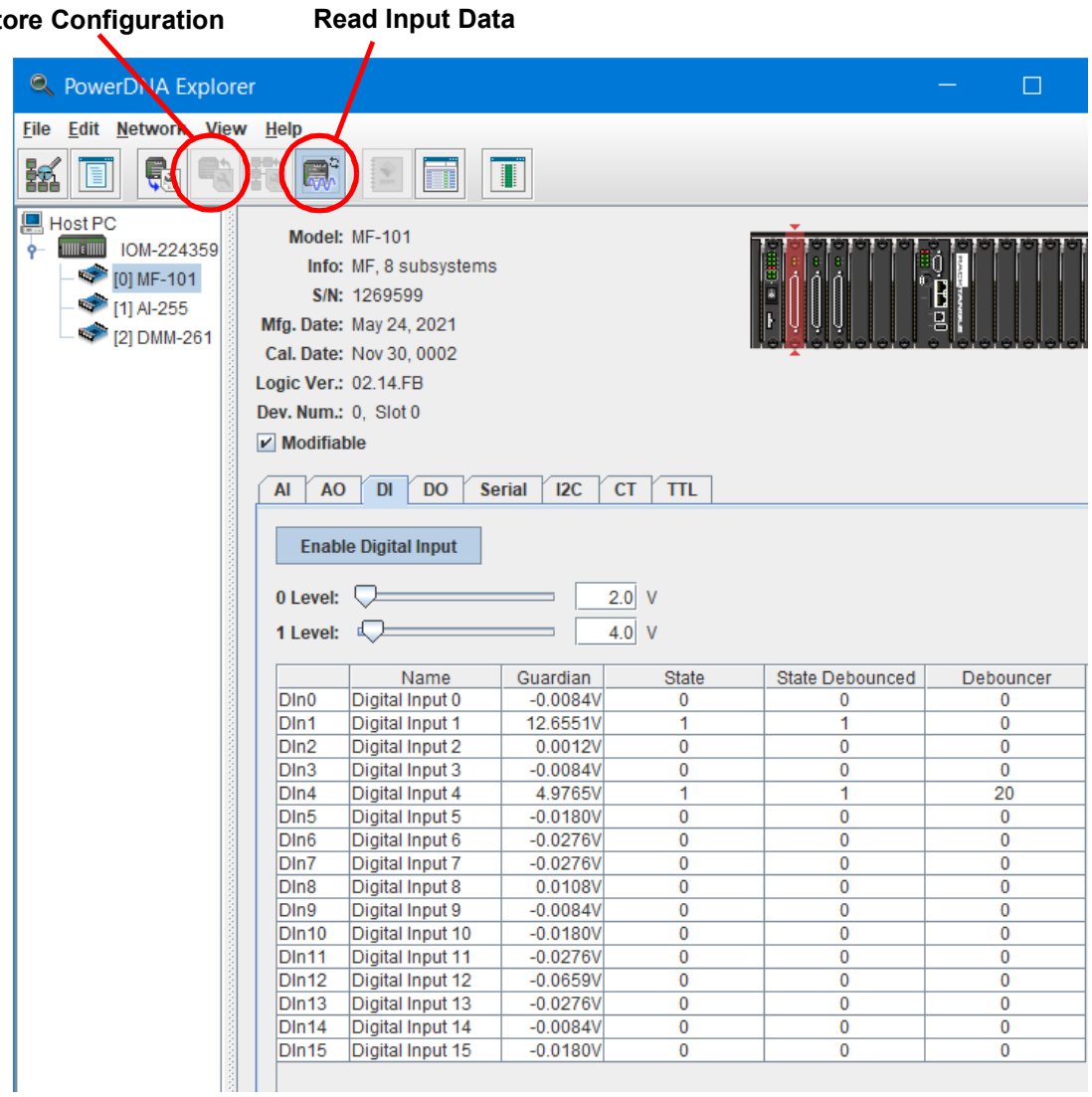
3.4 Industrial Digital Input

To explore the industrial digital input subsystem, open the DI tab (**Figure 3-5**) and click the **Enable Digital Input** button.

Click **Read Input Data** to start data acquisition.

The DI tab contains the following settings and displays:

- **0 Level**: slider and numeric text field for setting the logic level low threshold (between 0 to 55 V). The logic level changes from 1 to 0 when the input voltage transitions below the 0 Level. Click **Store Configuration** for the changes to take effect.
- **1 Level**: Slider and numeric text field for setting the logic level high threshold (between 0 to 55 V). The logic level changes from 0 to 1 when the input voltage transitions above the 1 Level. Click **Store Configuration** for the changes to take effect.
- **DInX**: read-only display of the channel number.
- **Name**: a name or note that you wish to give to the channel.
- **Guardian**: displays the voltage data from the channel's ADC.
- **State**: displays the current state of the channel. This state is determined by comparing the ADC voltage to the configured 0 Level and 1 Level.
- **State Debounced**: displays the debounced state of the channel. This logic level must have held steady over the number of samples defined in the "Debouncer" column. Click **Store Configuration** for the changes to take effect.
- **Debouncer**: numeric text field to set the debouncing interval for the channel. This is the number of ADC samples required for a debounced state change (max 65535).



	Name	Guardian	State	State Debounced	Debouncer
DIn0	Digital Input 0	-0.0084V	0	0	0
DIn1	Digital Input 1	12.6551V	1	1	0
DIn2	Digital Input 2	0.0012V	0	0	0
DIn3	Digital Input 3	-0.0084V	0	0	0
DIn4	Digital Input 4	4.9765V	1	1	20
DIn5	Digital Input 5	-0.0180V	0	0	0
DIn6	Digital Input 6	-0.0276V	0	0	0
DIn7	Digital Input 7	-0.0276V	0	0	0
DIn8	Digital Input 8	0.0108V	0	0	0
DIn9	Digital Input 9	-0.0084V	0	0	0
DIn10	Digital Input 10	-0.0180V	0	0	0
DIn11	Digital Input 11	-0.0276V	0	0	0
DIn12	Digital Input 12	-0.0659V	0	0	0
DIn13	Digital Input 13	-0.0276V	0	0	0
DIn14	Digital Input 14	-0.0084V	0	0	0
DIn15	Digital Input 15	-0.0180V	0	0	0

Figure 3-5 PowerDNA Explorer DI Tab

3.5 Industrial Digital Output

To explore the industrial digital output subsystem, open the DO tab and click the **Enable Digital Output** button.

3.5.1 Configure PWM

The DO PWM subtab (**Figure 3-6**) configures the following output channel properties:

- **PWM Period**: the period of the pulse-width modulated output in microseconds. Type in a number between 5 and 254,200, press the **Enter** key, and click the **Store Configuration** button.
- **DOutX**: read-only display of the channel number.
- **Name**: a name or note that you wish to give to the channel.
- **Mode**: one of the following PWM modes:
  - **PWM Disabled**: disables PWM on the output channel
  - **PWM Output**: enables PWM on the output channel
  - **Soft-Start**: When Output is switched from LOW to HIGH, the duty cycle gradually increases from 0% to the percentage specified in the Duty Cycle column over the specified Duration.
  - **Soft-Stop**: When Output is switched from HIGH to LOW, the duty cycle gradually decreases from the percentage specified in the Duty Cycle column to 0% over the specified Duration.
- **Push/Pull:** one of the following modes:
  - **Off**: No push-pull setting
  - **Push**: act as sourcing switch
  - **Pull**: act as sinking switch
  - **Push-pull:** connect as both push and pull, but never at same time (circuit shown in **Figure 2-19**c)
- **Duty Cycle (%)**: Defines the duty cycle for “PWM Output” mode and the soft start and soft stop modes.
- **Duration (ms)**: Defines the duration of the full “Soft-Start” or “Soft-Stop” cycle in milliseconds. This duration should be set longer than the PWM period.

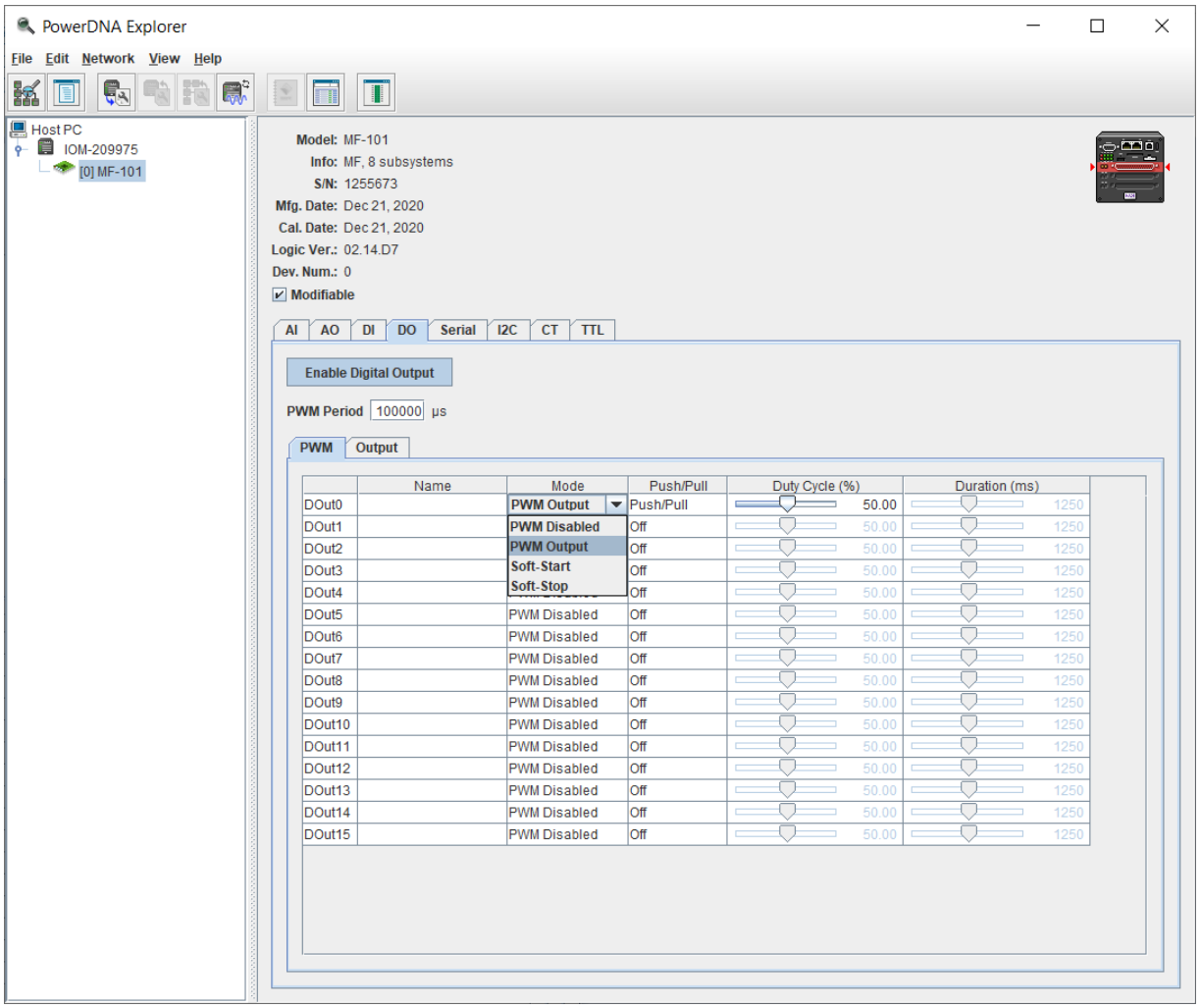


Figure 3-6 PowerDNA Explorer DO Tab, PWM Subtab

3.5.2 Write to Digital Output

The DO Output subtab (**Figure 3-7**) contains the following columns:

- **Ch X**: read-only display of the channel number.
- **Name**: a name or note that you wish to give to the channel.
- **High**: sets the output state to 1 (high-side FET turned on, low-side FET turned off)
- **Low**: sets the output state to 0 (high-side FET turned off, low-side FET turned on)
- **Tri**: configures the channel as input-only (both FETs turned off)

.

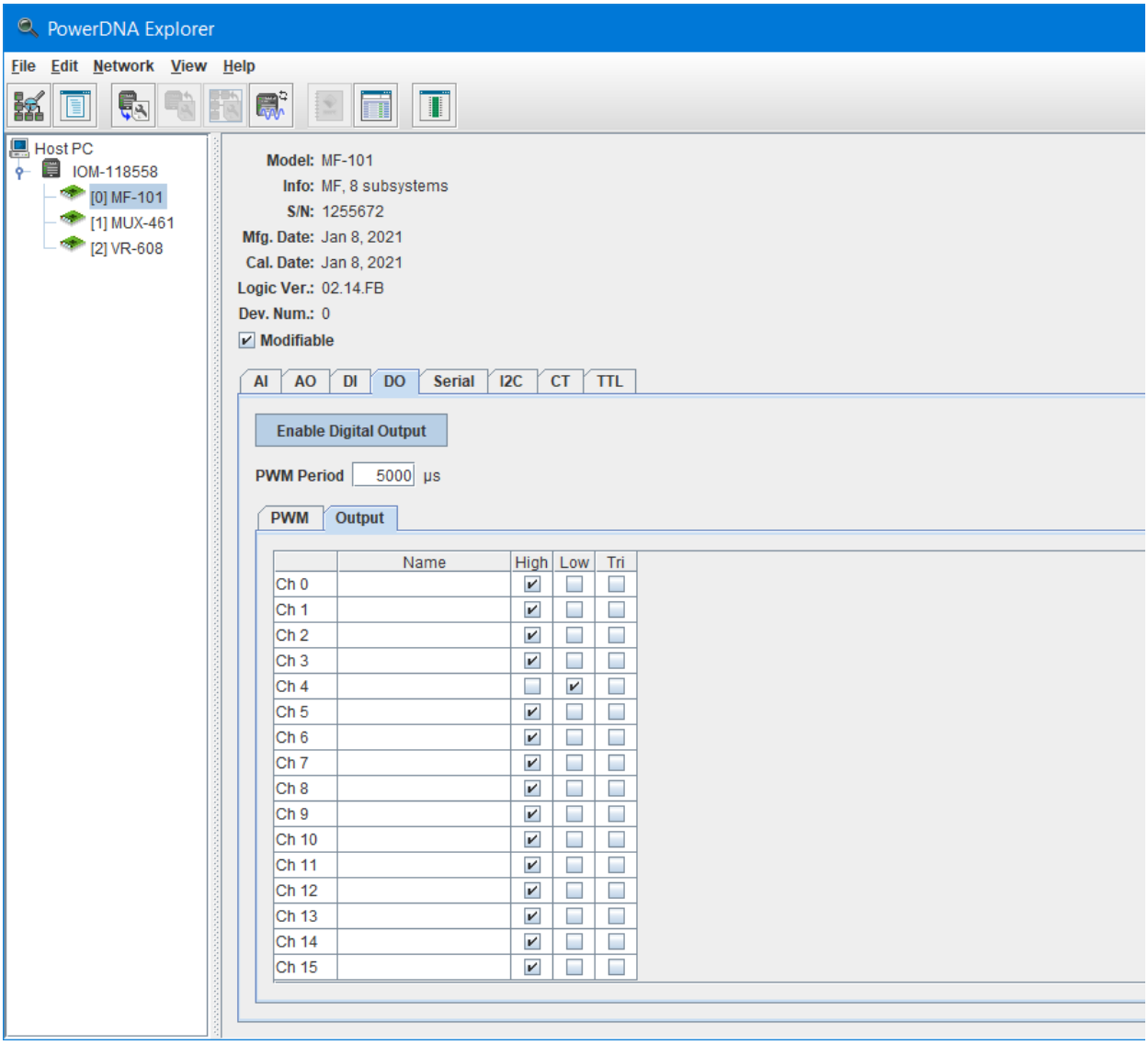


Figure 3-7 PowerDNA Explorer DO Tab, Output Subtab

3.6 RS-232/422/485 Port

To explore the RS-232/422/485 subsystem, open the serial tab and click **Enable Serial**.

3.6.1 Configure Serial Port

The Configuration subtab (**Figure 3-8**) contains the following settings:

- **Mode**: Configures the port mode to RS-232, RS-485 Full Duplex (compatible with RS-422), or RS-485 Half Duplex.
- **Baud**: Sets the baud rate in bits per second (bps). The minimum supported value is 300 bps. RS-232 mode supports rates up to 256 kbps, while RS-422/485 mode supports rates up to 2 Mbps.
- **Parity**: Sets the parity bit to None, Even Parity, or Odd Parity.
- **Data Bits**: Sets the number of data bits transferred with each frame.
- **Stop Bits**: Sets the number of STOP bits.
- **Break Enabled**: Holds the TX line at logic low.
- **Loopback Enabled**: Connects RX and TX internally and disables external signals.
- **Timeout**: Defines the timeout period in milliseconds when no data is seen on the RX line
- **Terminate Messages By String**: A Read stops after this string has been found.

Press the Enter key after typing numerical inputs and click **Store Configuration** to write settings to hardware.

Click the **Start Bus** button to enable the serial port.

NOTE: If you change the port configuration, new settings do not take effect until you stop and restart the bus.

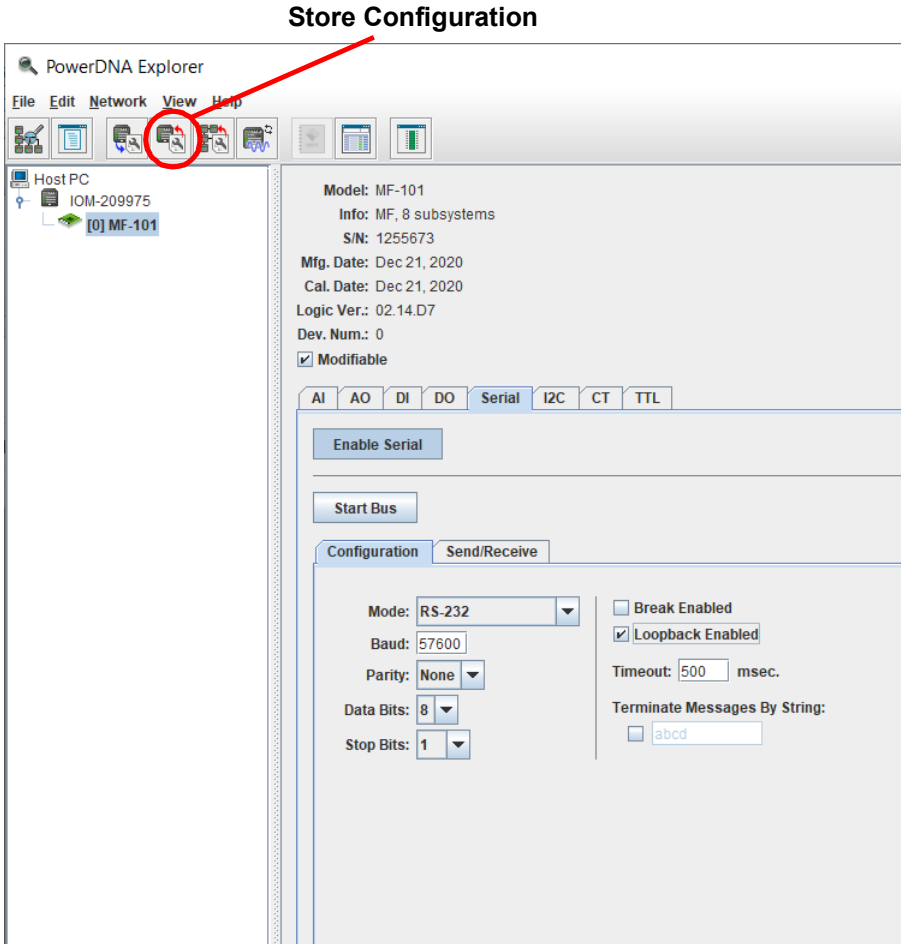


Figure 3-8 PowerDNA Explorer Serial Tab, Configuration Subtab

3.6.2 Send/Receive Data

The Send/Receive subtab (**Figure 3-9**) sends/receives either ASCII or Hex characters, as selected in the "Format" dropdown menu.

- **To Send Data**: Type either an ASCII string or Hex characters (separated by a space) into the text field next to the **Send** button. Click **Send** to write the data to the TX FIFO.
- **To Receive Data**: The "Bytes Requested" field sets the number of bytes to request from the RX FIFO. This value takes effect immediately. Click **Read Input Data** and view the received messages in the display. If the RX FIFO has less data than requested, or if the termination string is encountered, the returned message will be filled in with 0x00. The **Clear** button clears the message display.

Figure 3-9 shows the results of a simple loopback test. In this example, Loopback is enabled, three bytes of data are written to the TX FIFO, and two bytes of data are requested from the RX FIFO per read.

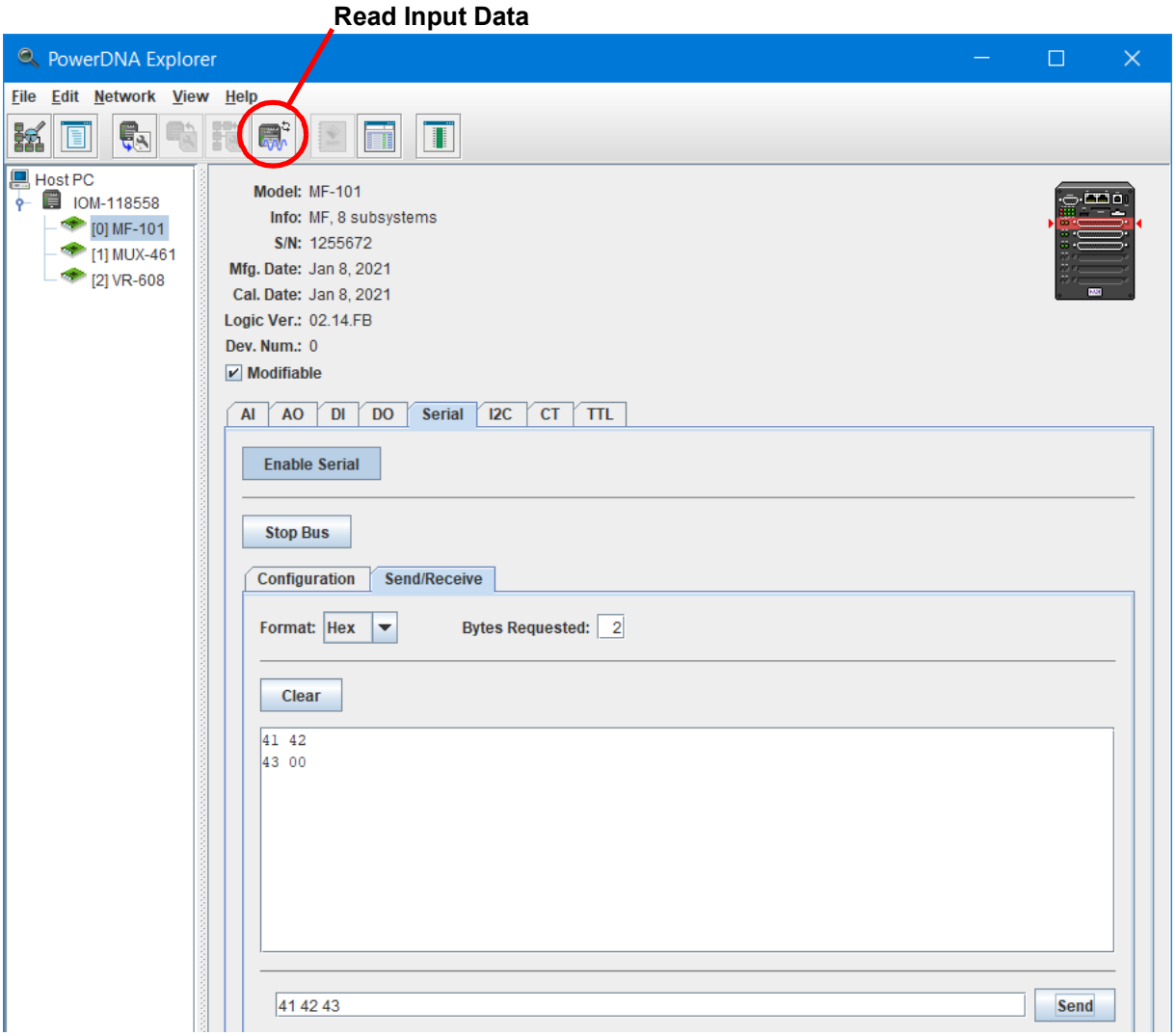


Figure 3-9 PowerDNA Explorer Serial Tab, Send/Receive Subtab

3.7 I²C Port

To explore the I²C subsystem, open the I²C tab and click **Enable Serial**.

3.7.1 Configure I²C Port

The I²C Configuration subtab (**Figure 3-10**) contains the following settings:

- **Clock**: Sets the clock rate to 100 kHz, 400 kHz, or 1 MHz. Both the slave and master modules run at the same clock rate.
- **Enable Master**: Enables the DNx-MF-101 master module.
- **Enable Slave**: Enables the DNx-MF-101 slave module.
  - **Enable BM Mode**: Configures the slave module as a Bus Monitor.
  - **Address**: Sets the slave address in hex format (7-bit default).
  - **10-Bit**: Configures the slave address as a 10-bit value.
  - **Default Data**: This data (specified in hex format) is loaded into the slave's 32-bit TX register and is automatically sent when the slave TX FIFO is empty.

Press the **Enter** key after typing numerical inputs and click **Store Configuration** to write settings to hardware. You can optionally **Reload Configuration** to read back and confirm the configuration.

Click the **Start Bus** button to enable the I²C port. Note that you must stop and restart the bus after changing the configuration.

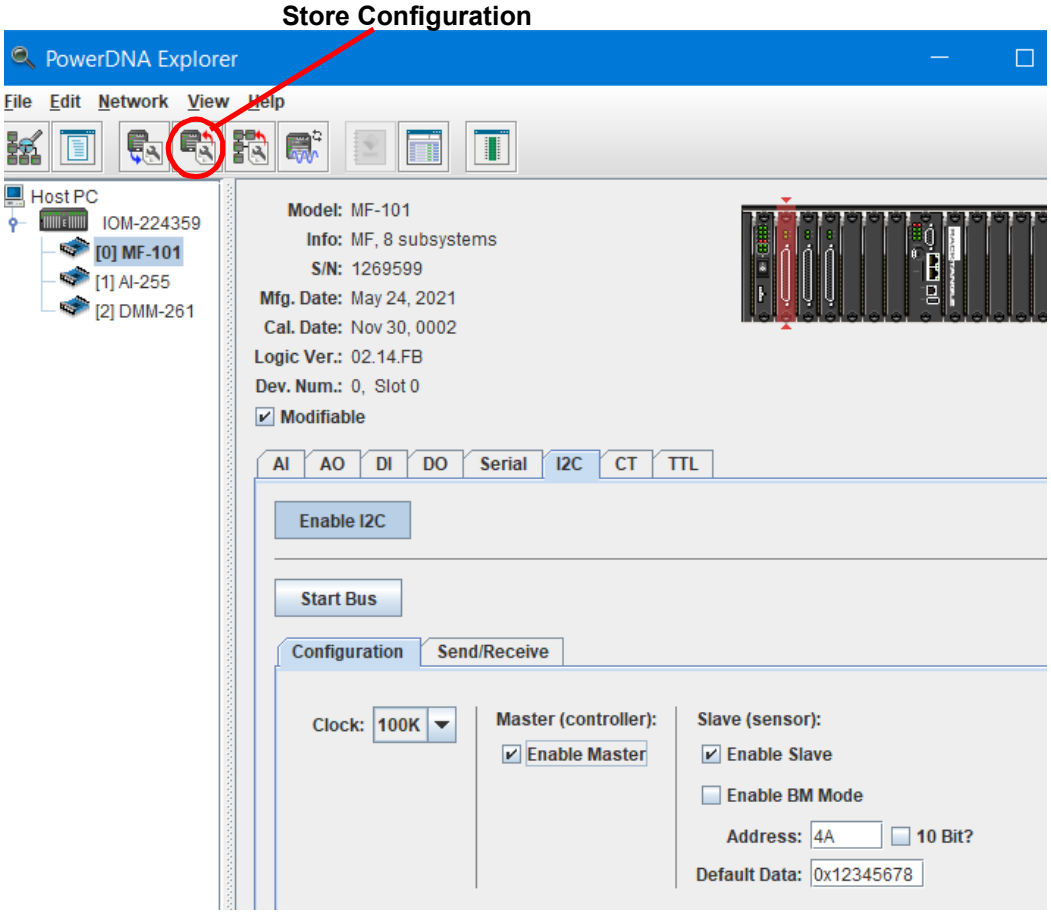


Figure 3-10 PowerDNA Explorer I2C Tab, Configuration Subtab

3.7.2 Read Command Example

The following loop-back test writes to the slave module's TX FIFO and reads back the data using the master module.

1. Enable both master and slave modules as shown in **Figure 3-10**. In this example, the slave address is set to 0x4A and its TX register is loaded with 0x12, 0x34, 0x56, and 0x78.
2. As shown in **Figure 3-11**, use the "Write Slave FIFO" command to write 0xaa and 0xbb to the slave TX FIFO. Click **Send Command**.

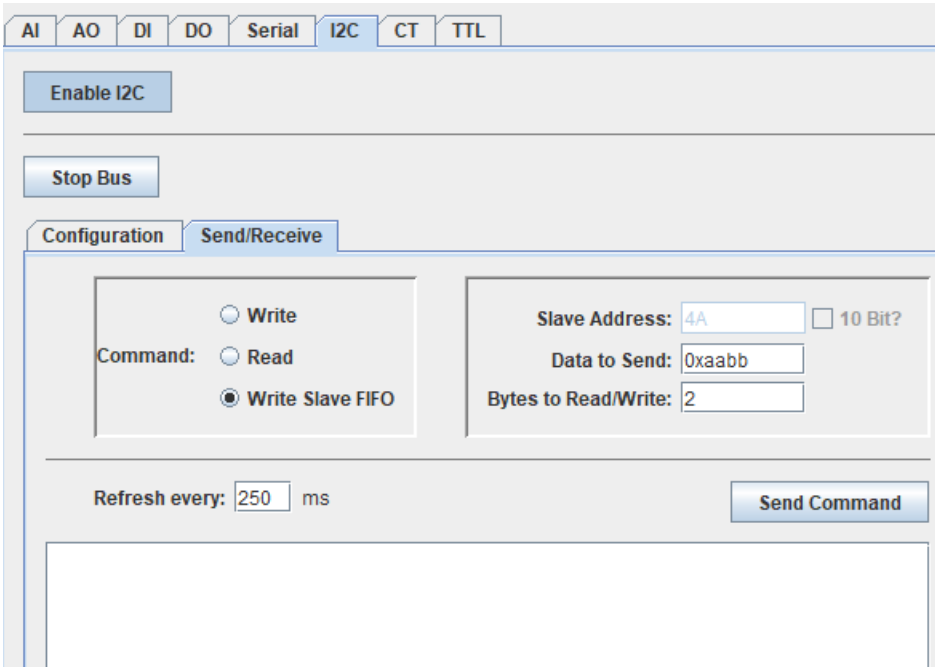


Figure 3-11 Write Slave FIFO Command

3. As shown in **Figure 3-12**, use the “Read” command to request 8 bytes from the slave at address 0x4A. Click **Send Command** and view the received data in the display window. The master receives the slave TX FIFO data, followed by the slave TX register data when the FIFO is empty. The display also shows the commands and data received by the slave (see Section 2.5.3.2). In this case no data was written to the slave.

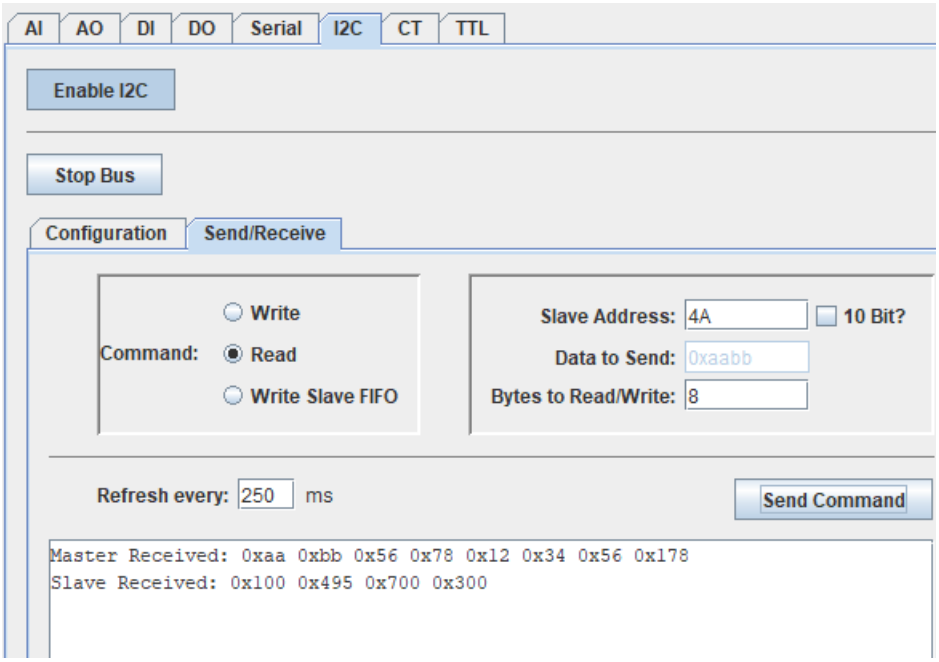


Figure 3-12 Read Command

3.7.3 Write Command Example

The following loop-back test uses the master module to write data to the slave module. The data is then read back from the slave’s RX FIFO.

1. Enable both master and slave modules as shown in **Figure 3-10**. In this example, the slave address is set to 0x4A.

2. As shown in **Figure 3-13**, use the “Write” command to write 0xaa, 0xbb, 0xcc, and 0xdd to the slave at address 0x4A. Click **Send Command** and view the slave RX FIFO data in the display window. The received data includes bus conditions as described in Section 2.5.3.2.

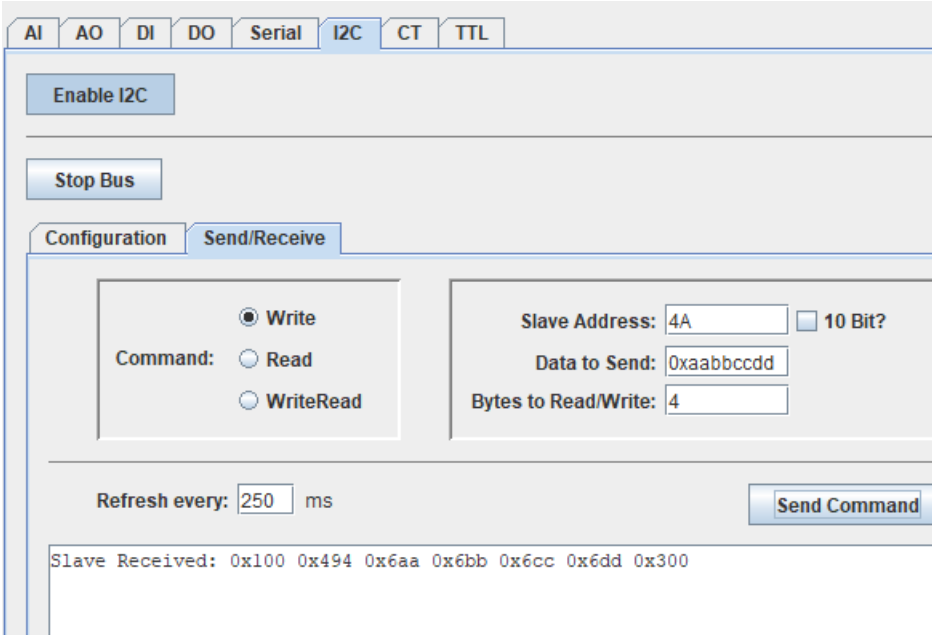


Figure 3-13 Write Command

3.7.4 Read Temperature Sensor

The DNx-TADP-101 and DNA-STP-MF-101 accessories connect an ADT7420 temperature sensor to the I^2C port. The temperature sensor address is 0x48. To read the temperature sensor:

1. As shown in **Figure 3-14**, send a “Write” command to setup the address pointer. Write 0x00 to read from the Tmsb register on the ADT7420.

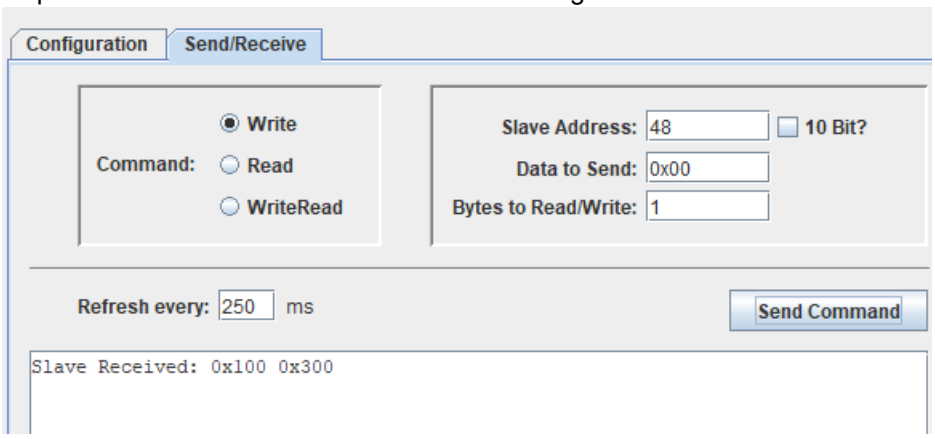


Figure 3-14 Setup Address for Temperature Sensor

2. As shown in **Figure 3-15**, send a “Read” command requesting two bytes. The ADT7420 encodes the temperature in a 13-bit number (discard the three LSBs). See the ADT7420 datasheet and/or the “Test Adapters User Manual” for information about converting the raw data to temperature.

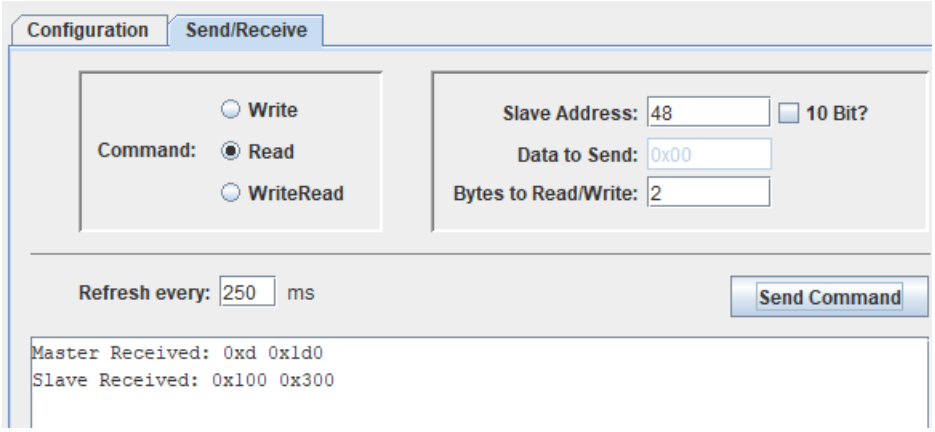


Figure 3-15 Send Command to Read Temperature Sensor

3.8 Counter/ Timer

To explore the counter/timer modules, open the CT tab and click **Enable Counter/Timer**.

3.8.1 Configure Count Mode and Sources

The DNx-MF-101 includes two independent counter/timer modules. Counter/timer configuration and operation depends on the selected mode. PowerDNA Explorer supports the following modes:

- **Quadrature**: counts pulses on the external Input. The count increases or decreases depending on the Gate signal. (Section 3.8.2)
- **Bin Counter**: counts pulses on the external input over a 1 second interval. (Section 3.8.3)
- **PWM Output**: generates a square wave on the Output. (Section 3.8.4)
- **Frequency**: measures the frequency of the external Input over a user-configured time interval. (Section 3.8.5).

You can route the counter's Gate, Input, and Output lines to any FET-based DIO or TTL DIO source listed in the dropdown menu. Click **Store Configuration** to program the source settings in hardware.

NOTE: When using FET-based sources as the Input or Gate, always configure digital input levels on the DI tab (Section 3.4) and click **Read Input Data** to enable the DI ADC.

3.8.2 Quadrature Mode

Quadrature Mode counts pulses on the Input Source. The count increases or decreases depending on the Gate Source. Output Source is unused in this mode.

The CT tab with Quadrature Mode settings is shown in **Figure 3-16**.

Click **Store Configuration** to write settings to hardware.

After you **Start** the counter, data is returned in the **Relative Position** field. This represents the number of pulses on the Input Source in hexadecimal format. Data starts from 0xffffffff and counts up if the value from Gate Source=1. The data counts down if Gate Source=0.

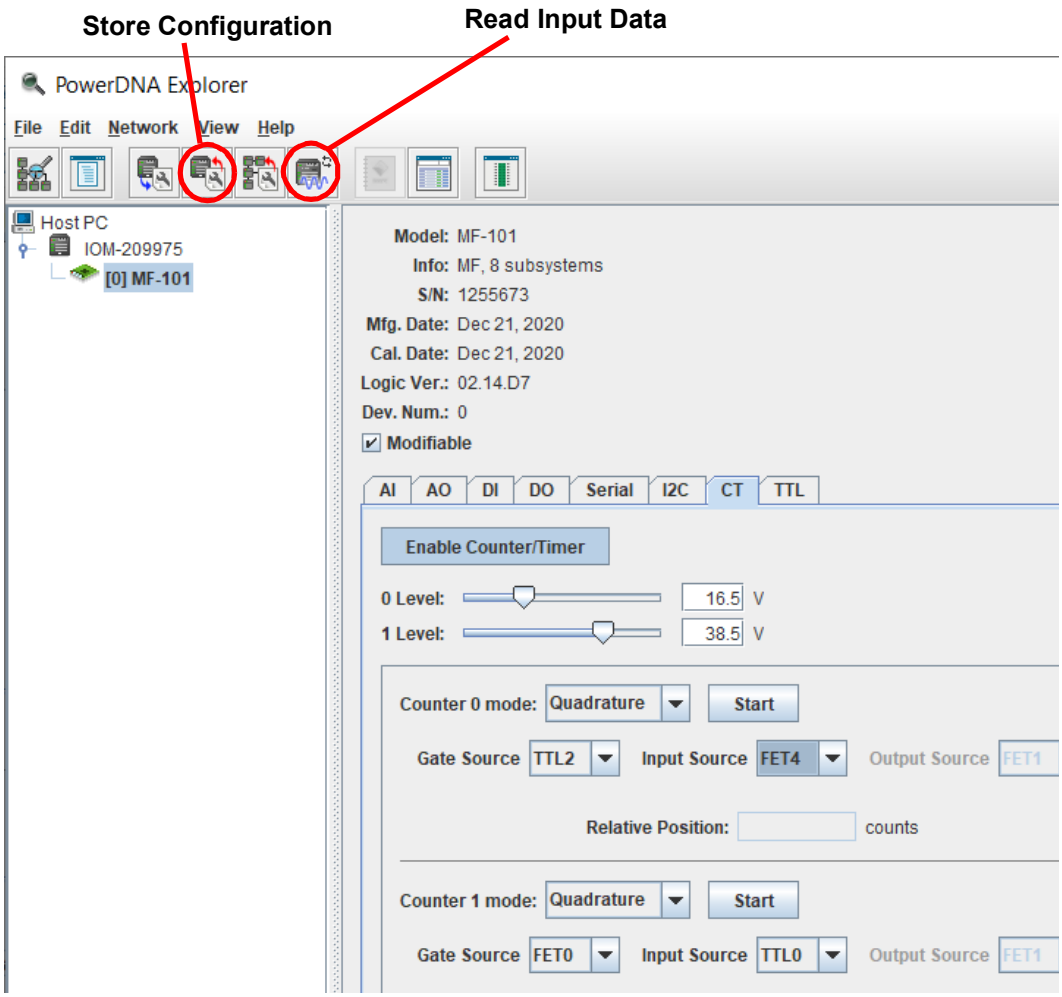


Figure 3-16 PowerDNA Explorer CT Tab, Quadrature Mode

3.8.3 Bin Counter Mode

Bin Counter Mode counts pulses on the Input Source over a 1 second interval. Output Source is unused in this mode.

The CT tab with Bin Counter Mode settings is shown in **Figure 3-17**.

Click **Store Configuration** to write settings to hardware.

After you **Start** the counter, data is returned in the following displays:

- **Counter Value**: number of pulses over 1 second time interval

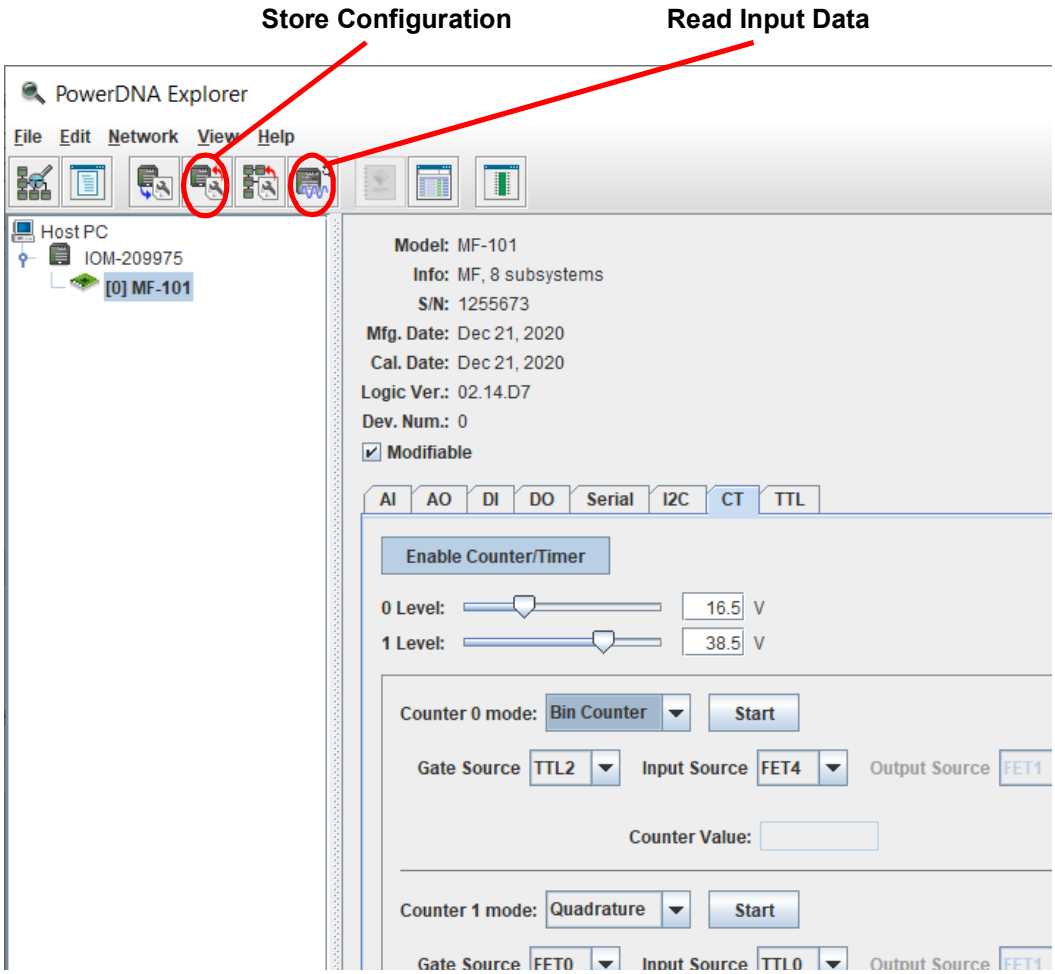


Figure 3-17 PowerDNA Explorer CT Tab, Bin Counter Mode

3.8.4 PWM Output Mode

PWM Output Mode generates a square wave on the Output Source. Gate and Input Sources are unused in this mode.

The CT Tab with PWM Output Mode settings is shown in **Figure 3-18**. It includes the following:

- **Duty Cycle**: Sets the duty cycle of the Output square wave.
- **Output Frequency**: Sets the desired frequency of the Output square wave, between 1 and 10,000 Hz.

Press the **Enter** key after typing numerical inputs and click **Store Configuration** to write settings to hardware.

After you **Start** the counter, PWM is output corresponding to the settings applied.

NOTE: For FET-based digital outputs, it is easier to generate PWM signals directly through the DO subsystem (Section 3.5).

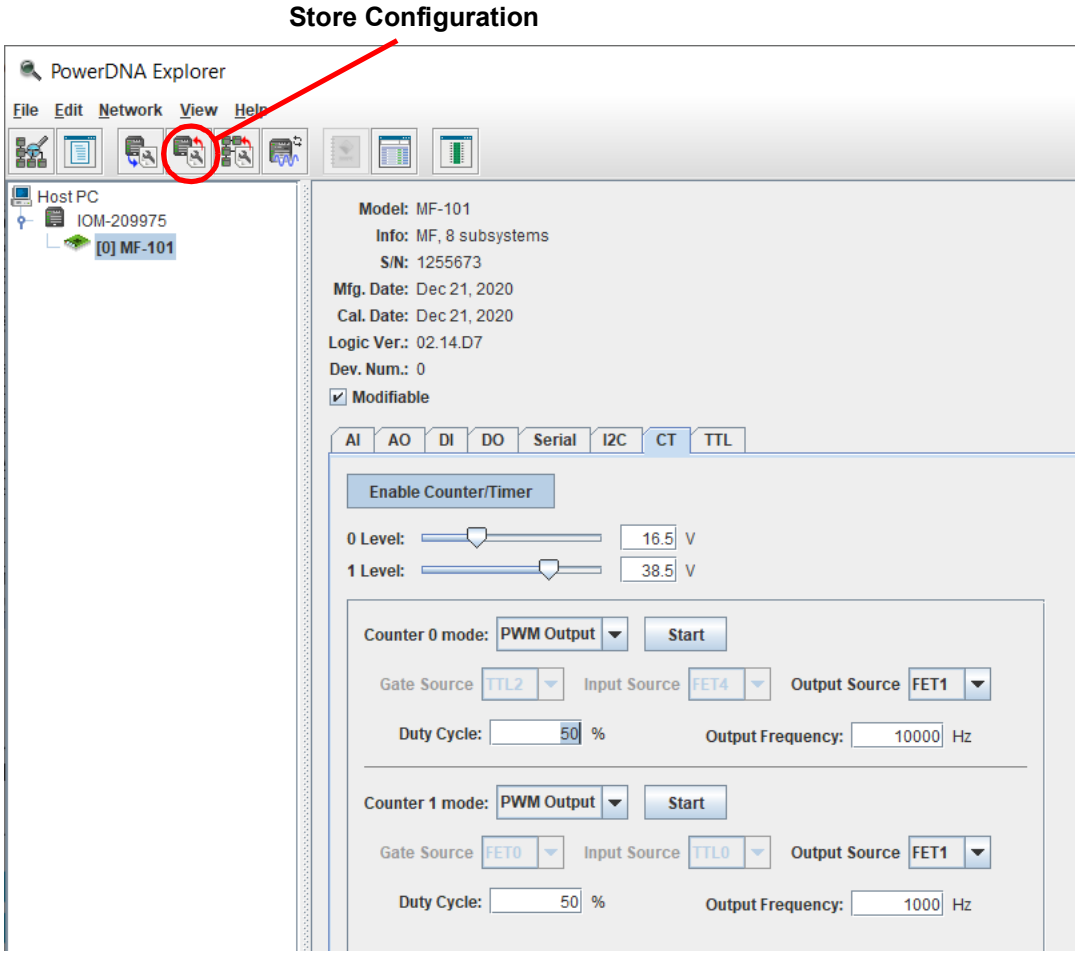


Figure 3-18 PowerDNA Explorer CT Tab, PWM Output Mode

3.8.5 Frequency Mode

Frequency Mode measures the frequency of the Input Source over a user-configured time interval.The Output Source is unused in this mode.

The CT Tab with Frequency Mode settings is shown in **Figure 3-19** It includes the following:

- **Measurement Period**: Time interval for the frequency measurement, between 1 and 32,537,631 microseconds.

Press the Enter key after typing numerical inputs and click **Store Configuration** to write the settings to hardware.

After you **Start** the counter, data is returned in the following displays:

- **Measured Frequency**: measured Input frequency in Hz.

Store Configuration

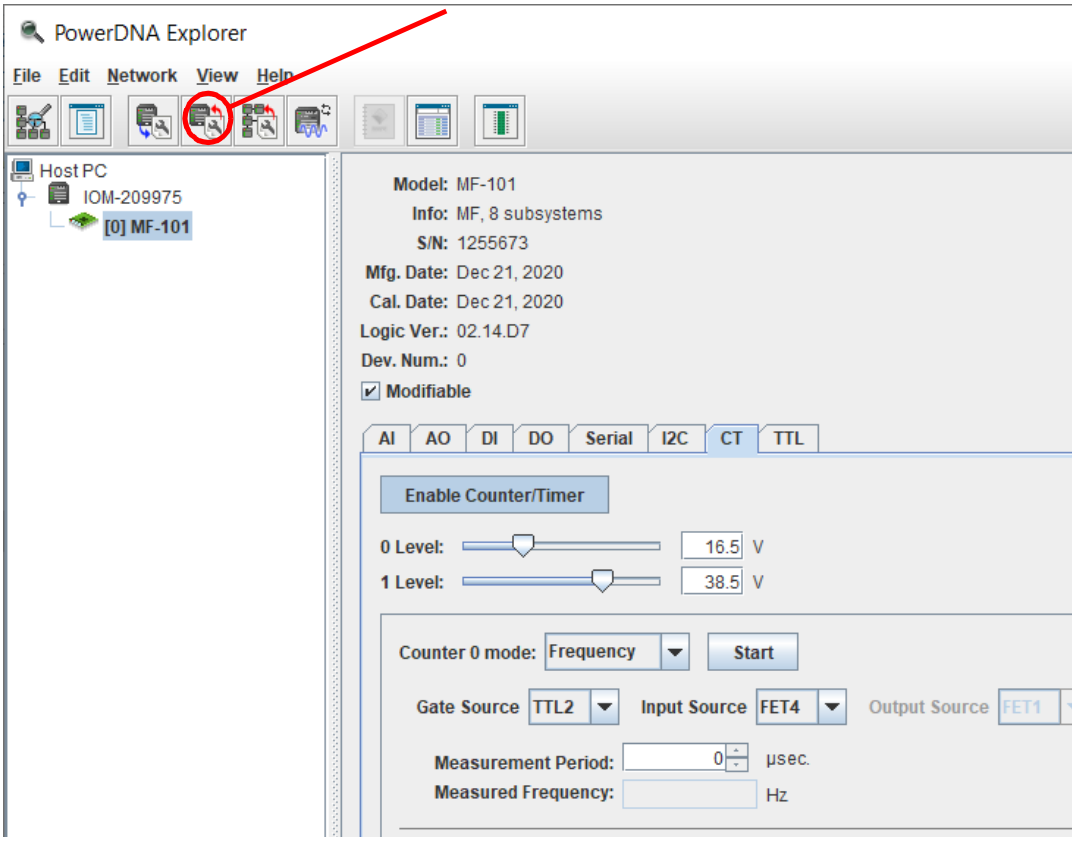


Figure 3-19 PowerDNA Explorer CT Tab, Frequency Mode

3.9 Logic-Level DIO

To explore the TTL-level digital I/O subsystem, open the TTL tab and click **Enable TTL**.

3.9.1 Configure TTL Port

The Configuration subtab (**Figure 3-20**) contains the following columns:

- **TTLX**: read-only display of the channel number.
- **Name**: a name or note that you wish to give to the channel.
- **In/Out**: radio button configures a channel pair as either Input or Output. DIO 0 and 1 are configured together, as are DIO 2 and 3. Click **Store Configuration** to program the new configuration.

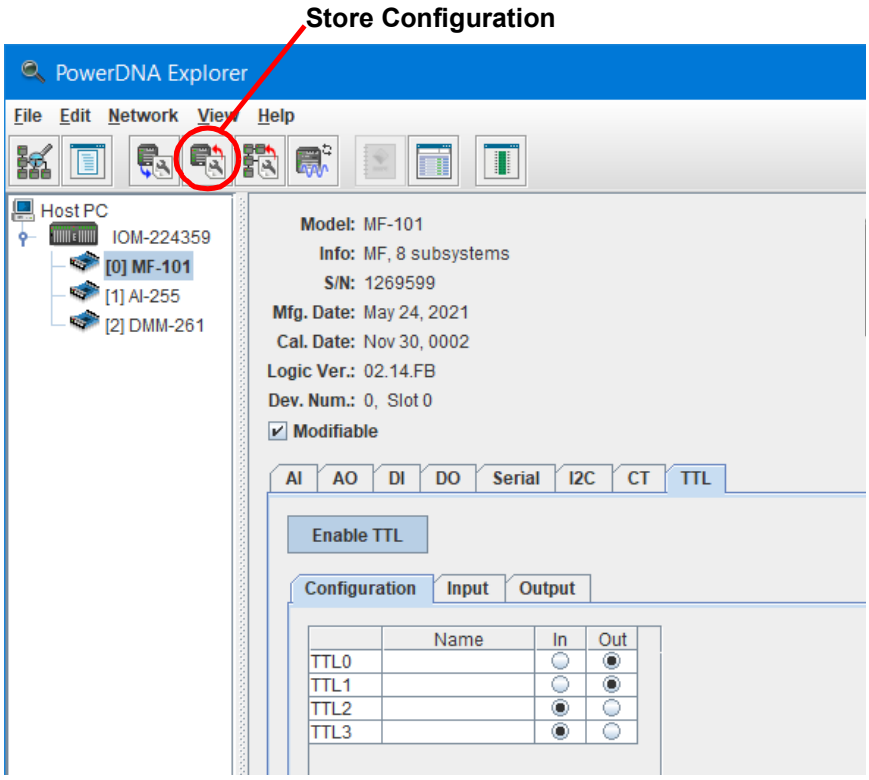


Figure 3-20 PowerDNA Explorer TTL Tab, Configuration Subtab

3.9.2 Read TTL Port

Click the **Read Input Data** button to read the state of all four TTL channels. The TTL Input subtab (**Figure 3-21**) contains the following columns:

- **TTLX**: read-only display of the channel number.
- **Name**: a name or note that you wish to give to the channel.
- **State**: displays the logic state of the channel

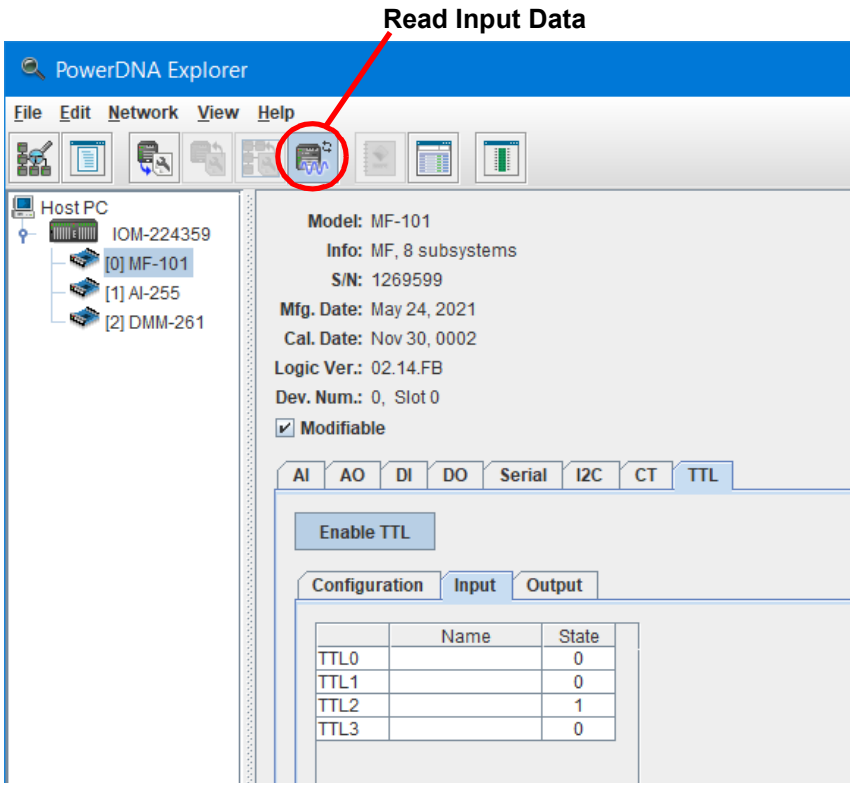


Figure 3-21 PowerDNA Explorer TTL Tab, Input Subtab

3.9.3 Write TTL Data

The TTL Output subtab (**Figure 3-22**) contains the following columns:

- **TTLX**: read-only display of the channel number.
- **Name**: a name or note that you wish to give to the channel.
- **State**: toggles the logic state of the channel. You can only write data on channels configured for Output.

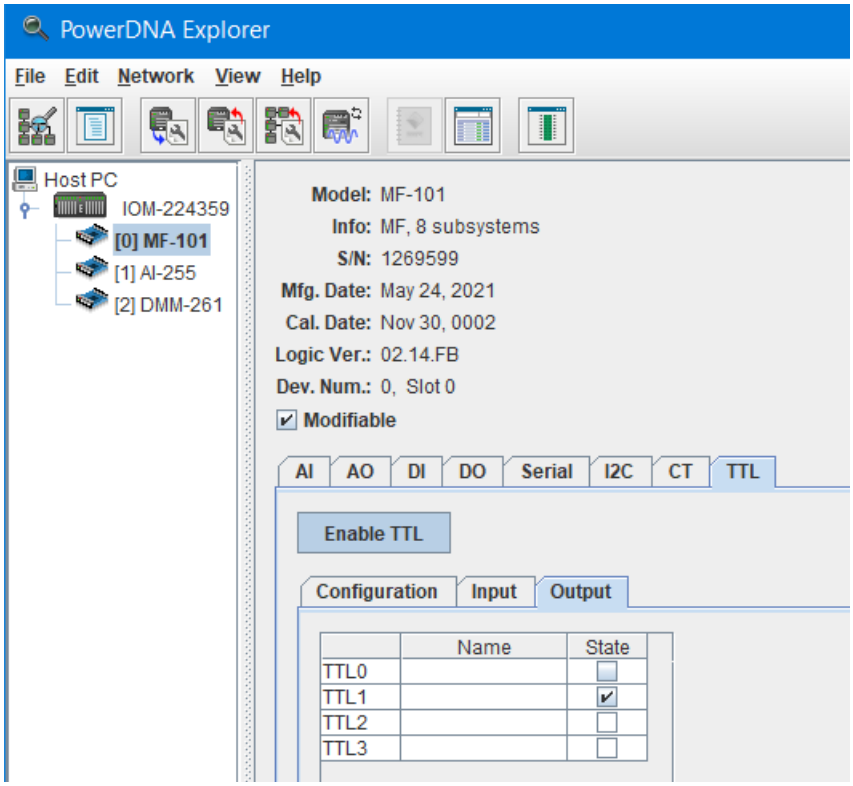


Figure 3-22 PowerDNA Explorer TTL Tab, Output Subtab

Chapter 4 Programming with High-level API

This chapter provides the following information about programming the DNx-MF-101 using the UeiDaq Framework API:

- About the High-level API (Section 4.1)
- Example Code (Section 4.2)
- Create a Session (Section 4.3)
- Assemble the Resource String (Section 4.4)
- Configure the Timing (Section 4.5)
- Start the Session (Section 4.6)
- Analog Input Session (Section 4.7)
- Analog Output Session (Section 4.8)
- Industrial Digital Input Session (Section 4.9)
- Industrial Digital Output Session (Section 4.10)
- TTL Digital Input Session (Section 4.11)
- TTL Digital Output Session (Section 4.12)
- Counter Input Session (Section 4.13)
- Counter Output Session (Section 4.14)
- Diagnostics Session (Section 4.15)
- Serial Port Session (Section 4.16)
- I2C Port Session (Section 4.17)
- Stop the Session (Section 4.18)

4.1 About the High-level API

UeiDaq Framework is object oriented and its objects can be manipulated in the same manner from different development environments, such as C++, Python, MATLAB, LabVIEW, and more. The Framework is supported in Windows 7 and up. It is generally simpler to use compared to the low-level API, and it includes a generic simulation device to assist in software development. Therefore, we recommend that Windows users use the Framework unless unconventional functionality is required. Users programming for a non-Windows operating system should instead use the low-level API (Chapter 5).

For more detail regarding the Framework's architecture, please see the *"UeiDaq Framework User Manual"* located under:

Start » All Programs » UEI

For information on the Framework's classes, structures, and constants, please see the *"UeiDaq Framework Reference Manual"* located under:

Start » All Programs » UEI

4.2 Example Code

UeiDaq Framework is bundled with examples for supported programming languages. The example code is located under:

C:\Program FIles (x86)\UEI\Framework

The examples can be accessed via the Windows Start Menu. For example:

Start » All Programs » UEI » Visual C++ Examples

Unlike the low-level examples, Framework examples are board-agnostic, e.g., the “AnalogInBuffered” example works across all UEI analog input layers which support the Advanced Circular Buffer (ACB) data acquisition mode.

Each high-level example follows the same basic structure listed in the following steps. Subsystem configuration (Step 3) and reading and writing of data (Step 6) are specific to particular subsystems so that information is presented in sections that are tailored to that subsystem.

1. Create a session (Section 4.3).
2. Assemble the resource string (Section 4.4).
3. Configure the session for a particular device and subsystem (Section 4.7 through Section 4.17).
4. Configure the timing (Section 4.5).
5. Start the session (Section 4.6).
6. Read or write data (Section 4.7 through Section 4.17).
7. Stop the session (Section 4.18).

This chapter presents examples using the C++ API, but the concepts are the same no matter which programming language you use. The *“UeiDaq Framework User Manual”* provides additional information about programming in other languages.

4.3 Create a Session

The session object manages all communications with the DNx-MF-101. Therefore, the first step is always to create a new session.

```
//create a session object

CUeiSession mySession;
```

NOTE: If you want to use multiple subsystems on the DNx-MF-101 (for example simultaneous analog input and output), you will need to create a new session for each subsystem. Therefore, example sessions for each subsystem will be given unique names.

4.4 Assemble the Resource String

Each session is dedicated to a specific subsystem within the device. Framework uses a resource string to link the session to the hardware. The resource string syntax is similar to a web URL; it should not have any spaces and is case insensitive.

“<device class>://<IP address>/<device number>/<subsystem><channel list>”

The components of a resource string are as follows:

- *<device class>* - By default, Framework examples open with a generic simulated device. To use the DNx-MF-101, set the device class to `pdna`.
- *<IP address>* - IP address of the IOM.
- *<device number>* - position of the DNx-MF-101 within the chassis, relative to the other I/O boards.
- <subsystem> - one of the following DNx-MF-101 subsystems:
  - `Ai`: analog input session (Section 4.7)
  - `Ao`: analog output session to generate voltage and/or current (Section 4.8)
  - `Di0`: industrial digital input session to configure all 16 lines (Section 4.9)
  - `Diline0`: industrial digital input session to configure selected lines (Section 4.9)
  - `Do0`: industrial digital output session to configure all 16 lines (Section 4.10)
  - `Doline0`: industrial digital output session to configure selected lines (Section 4.10)
  - `Di1`: TTL digital input session (Section 4.11)
  - `Do1`: TTL digital output session (Section 4.12)
  - `Ci`: counter input session to count events or measure pulse width and period (Section 4.13)
  - `Co`: counter output session to generate pulses and pulse trains (Section 4.14)
  - `Diag`: diagnostic session to read from analog output and DIO ADCs (Section 4.15)
  - `Com`: serial port session to send/receive RS-232/422/485 data (Section 4.16)
  - `I2C`: session to send/receive I^2C master/slave data (Section 4.17)
- *<channel list>* - desired lines or ports within the selected subsystem, either as a comma-separated list of numbers or a range. If the subsystem name ends in a number, separate the subsystem and channel list with a forward slash.

Example 1

Here are two valid resource strings for selecting analog input lines 0,1,2,3 on device 1 at IP address 192.168.100.2:

- `"pdna://192.168.100.2/Dev1/Ai0,1,2,3"`
- `"pdna://192.168.100.2/Dev1/Ai0:3"`

Example 2

The following resource string selects TTL digital input port 0 on device 1 at IP address 192.168.100.2:

- `"pdna://192.168.100.2/Dev1/Di1/0"`

Refer to Section 4.7 through Section 4.17 for details on configuring the different types of subsystems.

4.5 Configure the Timing

The UeiDaq Framework supports Point-by-Point data acquisition mode for all DNx-MF-101 functions. AVMap mode is supported for analog inputs. Additional DAQ modes are supported by the low-level API (Chapter 5).

Table 4-1 DAQ Modes Supported by UeiDaq Framework

DAQ Mode	AIn	AOut	DIn	DOut	TTL	CT	Serial	I2C
Point-by-Point	●	●	●	●	●	●	●	●
ACB								
RtDMap								
RtVMap								
ADMap								
AVMap	●							

Point-by-Point mode transfers one sample at a time to or from each configured channel of the I/O board. The delay between samples is controlled by the host application (e.g., by using a Sleep function), thus limiting the data transfer rate to a maximum of 100 Hz. This mode is also known as immediate mode or simple mode.

Point-by-Point mode uses Simple IO timing.

```
//configure timing for Point-by-Point DAQ mode

mySession.ConfigureTimingForSimpleIO();
```

AVMap mode allows acquisition of a variable number of samples per configured analog input channel instead of a single sample.

AVMap mode requires timing to be configured by calling `ConfigureTimingForAsyncVMapIO()`:

```
//configure timing for AVMap DAQ mode
//use an internal clock, 50 Hz date rate into FIFO, digital edge is
//ignored for analog inputs, FIFO watermark of 100 scans, period is not
//used when watermark is used

aiSession.ConfigureTimingForAsyncVMapIO(UeiTimingClockSourceInternal,
             50.0, UeiDigitalEdgeRising, 100, 0);
```

Configure each session in the application with the appropriate timing mode.

NOTE: `ConfigureTimingForMessagingIO()` is only supported by SL-50x and CAN boards. It is NOT available on the DNx-MF-101.

4.6 Start the Session

After a session is configured, you can start the session manually:

```
//Start the session.
mySession.Start();
```

If you don't explicitly start the session, it will start automatically the first time you try to transfer data.

4.7 Analog Input Session

The session may be configured to access the analog input (`Ai`) subsystem.

4.7.1 Add Input Channels

The `CreateAIChannel()` method adds a new channel for each analog input specified in the resource string. Single-ended inputs are numbered 0...15 and differential-ended inputs are numbered 0...7. It is possible to call `CreateAIChannel()` multiple times to add channels with different gains or input modes.

```
//Configure ch[0:2] to read differential inputs 0, 1, and 2.
//Set gain to 1x (-10 V to 10 V range when voltage divider is disabled).
aiSession.CreateAIChannel("pdna://192.168.100.2/Dev1/Ai0:2",
   -10, 10, UeiAIChannelInputModeDifferential);
//Configure ch[7:15] to read the remaining inputs in single-ended mode.
aiSession.CreateAIChannel("pdna://192.168.100.2/Dev1/Ai7:15",
   -10, 10, UeiAIChannelInputModeSingleEnded);
```

The `min` and `max` parameters in `CreateAIChannel()` configure the channel gain. Table 4-2 shows the supported min/max values and their corresponding gain settings. For example, setting [`min`, `max`] to either [-10, 10] or [-80, 80] configures the gain to x1.

Table 4-2 Analog Input Ranges (Volts)

Gain	Without divider	With divider
x1	[-10, 10]	[-80, 80]
x4	[-2.5, 2.5]	[-20, 20]
x8	[-0.625, 0.625]	[-5,5]
x64	[-0.15625, 0.15625]	[-1.25, 1.25]

When reading input channels, saturation or clipping can occur if the gain is too high, making the value appear stuck at the highest or lowest value. Try a lower gain value, or begin with x1. If you accidentally create a channel with unsupported values, the board will be programmed with the closest supported gain.

NOTE: To use the input ranges in the "With divider" column, you must also enable the voltage divider (see Section 4.7.2 below). Setting [`min`, `max`] to [-80, 80], [-20, 20], [-5, 5], or [-1.25, 1.25] only programs the gain; it does not automatically enable the divider.

4.7.2 Enable Voltage Divider

When enabled, the voltage divider reduces the voltage on the channel by a factor of 8. It is also a convenient way to tie unused input pins to ground, as is required on the DNx-MF-101 (see Section 2.8.1.2). The divider is enabled/disabled individually for each channel.

```
//Enable voltage divider on every channel in the session.

for (int ch = 0; ch < aiSession.GetNumberOfChannels(), ch++)
{
   CUeiAIChannel* aichannel =
         dynamic_cast<CUeiAIChannel*>(aiSession.GetChannel(ch));

   aichannel->EnableVoltageDivider(true);
}
```

NOTE: Use the `GetChannel()` method to obtain a pointer to a channel, rather than `CUeiAIChannel* aichannel = aiSession.CreateAIChannel().CreateAIChannel()` returns a pointer to only the first channel in the list.

4.7.3 Add Timestamp

Timestamp the data by adding a `ts` channel as the last channel in the resource string: "`pdna://192.168.100.2/Dev1/Ai0:2,ts`". The units will be in seconds. Note that there are no spaces in a properly formatted resource string.

4.7.4 Configure Moving Average

Enabling the moving average can smooth out noise from the sensor input line. The number of samples used for the moving average may be set to 0, 2, 4, 8, 16, 32, 64, 128, or 256. The default window size is 0 (turned off/average every sample). Moving average samples are acquired at the analog input subsystem clock rate (default 2 kHz).

```
//Set moving average window size to 128 samples.

aichannel->SetMovingAverageWindowSize(128);
```

4.7.5 Read Data

Reading data is done using a reader object. An Analog Raw Reader returns the calibrated binary data and an Analog Scaled Reader returns the data converted to volts. The following example code shows how to create a scaled reader object and read input voltages.

```
//Create a reader object and link it to the session's data stream.
CUeiAnalogScaledReader aiReader(aiSession.GetDataStream());
//Buffer must be large enough to contain one sample per channel.
double data[100];
//For point-to-point, read one sample per channel.
aiReader.ReadSingleScan(data);
//For AVMap, read available scans.
aiReader.ReadMultipleScans(100, data);
```

4.8 Analog Output Session

The session may be configured to access the analog output (`Ao`) subsystem.

4.8.1 Configure Output Channels

The two analog outputs on the DNx-MF-101 are independently configurable as either voltage or current outputs.

4.8.1.1 Voltage Output

Use the `CreateAOChannel()` method to add a new voltage output channel to the session. The channel is linked to the output line(s) specified in the resource string.

```
//Configure ch[0] to output voltage on AOut 0 in the -10V to 10V range.
aoSession.CreateAOChannel("pdna://192.168.100.2/Dev1/Ao0", -10, 10);
```

Voltage output ranges (V):

- [-5, 5]
- [-10, 10]

If you accidentally create a channel with unsupported min or max values, the board will be programmed with the closest supported range.

4.8.1.2 Current Output

Use the `CreateAOCurrentChannel()` method to add a new current output channel.

```
//Configure ch[1] to output current on AOut 1 in the 4mA to 20mA range.
aoSession.CreateAOCurrentChannel("pdna://192.168.100.2/Dev1/Ao1",
   4, 20);
```

Current output ranges (mA):

- [0, 20]
- [4, 20]
- [-1, 22]

4.8.2 Write Data

Writing data is done using a writer object. An Analog Raw Writer sends binary data straight to the D/A converter. An Analog Scaled Writer accepts data in units of volts or milliamps, depending on the channel configuration, and automatically converts the scaled data to binary.

The following example code shows how to create a scaled writer object and write a single set of data. Assume both channels are configured for voltage output in the ± 10V range.

```
//Create a writer object and link it to the session's data stream.
CUeiAnalogScaledWriter aoWriter(aoSession.GetDataStream());
//Buffer contains one value per channel.
double data[2] = {-2.5, 7.5};
//Write -2.5V to ch[0] and 7.5V to ch[1]
aoWriter.WriteSingleScan(data);
```

NOTE: The DNx-MF-101 does not support the `CreateAOWaveform()` method. Instead, you must manually generate waveform data and load it into the data buffer.

4.8.3 Read Diagnostic Data

You can read temperature and voltage from the analog output ADCs through a separate Diagnostic session (Section 4.15).

4.9 Industrial Digital Input Session

The session may be configured to access the industrial digital input (`Di0` or `Diline0`) subsystem.

4.9.1 Configure Input Channels

The `CreateDIIndustrialChannel()` method adds FET-based digital input channels, sets their hysteresis thresholds, and programs a debouncer to eliminate glitches and spikes.

NOTE: When configuring DNx-MF-101 channels as both industrial digital inputs and industrial digital outputs, the inputs must be configured before the outputs.

```
CUeiDIIndustrialChannel* CreateDIIndustrialChannel(std::string resource,
   double lowThreshold, double highThreshold, double minPulseWidth);
```

- `resource` – Resource string specifying the port (Section 4.9.1.1) or the line (Section 4.9.1.2)
- `lowThreshold` – Logic level changes from 1 to 0 when the input voltage falls below the low hysteresis threshold.*
- `highThreshold` – Logic level changes from 0 to 1 when the input voltage rises above the high hysteresis threshold.*
- `minPulseWidth` – Debouncer only allows a state change when the input has remained stable at the new level for this number of milliseconds. Use 0.0 to disable the debouncer. The maximum allowable value for `minPulseWidth` width is 327 ms. If a larger value is passed to this method, a value of 327 ms will be used.

*If the signal is in between the low and high thresholds, the detector maintains the previous logic level.

4.9.1.1 Adding a Port

Using `Di0` in the resource string adds the entire digital input port to one channel.

```
//Get pointer to input port (channel index = 0) and configure DIO0:15
//with low threshold=2.0 V, high threshold=3.0 V, and
//debouncing interval=1.0 ms.

CUeiDIIndustrialChannel* diPort = diSession.CreateDIIndustrialChannel
      ("pdna://192.168.100.2/Dev1/Di0",
       2.0, 3.0, 1.0);
```

You can reconfigure individual lines using methods in the `CUeiDIIndustrialChannel` class.

```
//Change DIO7 configuration to low threshold=1.5 V, high threshold=3.5 V,
//and debouncing interval=2.0 ms.

diPort->SetLowThreshold(7, 1.5);
diPort->SetHighThreshold(7, 3.5);
diPort->SetMinimumPulseWidth(7, 2.0);
```

4.9.1.2 Adding Selected Lines

Alternatively, you can configure a subset of lines by specifying `Diline0` in the resource string and appending the desired line numbers. Note that all digital input channels should be initially configured in a single call to `CreateDIIndustrialChannel()`.
Calling `CreateDIIndustrialChannel()` multiple times on the same session will result in only the channels in the final call being added to the session.

```
//Configure DIO2:3 and DIO7:10, initially with the same hysteresis
//thresholds debounce interval.

CUeiDIIndustrialChannel* diLines = diSession.CreateDIIndustrialChannel
("pdna://192.168.100.2/Dev1/Diline0/2:3,7:10", 2.0, 3.0, 1.0);
```

This will create a number of `CUeiDIIndustrialChannel` instances equal to the number of specified digital input lines. Per-channel configuration can then be performed on the channels. The order of channels is the same order in which the channels appeared in the resource string. Note that the `<line>` parameter when setting channel parameters is always 0 when using the "DiLine" session type. The following example sets the low threshold for each of the digital input lines specified in the resource string above.

```
//Set channel index 0 (line 2 in the resource string) low threshold to 0 V
((CUeiDIIndustrialChannel*)diSession.GetChannel(0))->SetLowThreshold(0, 0.0);
// Set channel index 1 (line 3 in the resource string) low threshold to 1 V
((CUeiDIIndustrialChannel*)diSession.GetChannel(1))->SetLowThreshold(0, 1.0);
// Set channel index 2 (line 7 in the resource string) low threshold to 2 V
((CUeiDIIndustrialChannel*)diSession.GetChannel(2))->SetLowThreshold(0, 2.0);
// Set channel index 3 (line 8 in the resource string) low threshold to 3 V
((CUeiDIIndustrialChannel*)diSession.GetChannel(3))->SetLowThreshold(0, 3.0);
// Set channel index 4 (line 9 in the resource string) low threshold to 4 V
((CUeiDIIndustrialChannel*)diSession.GetChannel(4))->SetLowThreshold(0, 4.0);
// Set channel index 5 (line 10 in the resource string) low threshold to 5 V
((CUeiDIIndustrialChannel*)diSession.GetChannel(5))->SetLowThreshold(0, 5.0);
```

4.9.2 Read Data

Reading data is done using a Digital Reader object. This is created using the session's data stream object.
Digital data is stored in a 16-bit integer buffer. The reader reads from all lines in the port, even if `Diline` configured only a subset of lines.

```
//Create a reader object and link it to the session's data stream.

CUeiDigitalReader diReader(diSession.GetDataStream());
```

4.9.2.1 Read DI Port

When reading industrial digital input data from a `Di0` session, use `uInt16` data. A single `uInt16` will be returned with the low/high debounced status mask of all 16 channels.

```
//Read state of DIO0:15

uInt16 data;
diReader.ReadSingleScan(&data);
```

4.9.2.2 Read Specific DI Lines

When reading industrial digital input data from a `Diline0` session, use `uInt16` data. A number of `uInt16` values will be returned that will be equal to the number of configured channels. Only bit 0 of each 16-bit value should be used (0 is low, 1 is high) .

```
//Read state of DIO0:15

uInt16* digitalState = new uInt16[diSession.GetNumberOfChannels()];
diReader.ReadSingleScan(digitalState);
```

NOTE: If you are simultaneously running a digital output session, ensure that the output mask is disabled for the input-only lines. Otherwise, the reader will return the values written to the port.

4.9.3 Read Input Voltages

You can read voltage from the DIO ADCs by creating a separate Diagnostic session (Section 4.15).

4.10 Industrial Digital Output Session

The session may be configured to access the industrial digital output (`Do0` or `Doline0`) subsystem. Because sessions are unidirectional, you will need a dedicated output session even though output and input share the same physical port.

4.10.1 Configure Output Channels

The `CreateDOIndustrialChannel()` method adds FET-based digital output channels and configures PWM on those channels.

NOTE: When configuring DNx-MF-101 channels as both industrial digital inputs and industrial digital outputs, the inputs must be configured before the outputs.

```
CUeiDOIndustrialChannel* CreateDOIndustrialChannel(std::string resource,
   tUeiDOPWMMode pwmMode, uInt32 pwmLengthUs, uInt32 pwmPeriodUs,
   double pwmDutyCycle);
```

- `resource` – Resource string specifying the port (Section 4.10.1.1) or the line (Section 4.10.1.2)
- `pwmMode` – Type of pulse train to output (Section 4.10.1.4)
- `pwmLengthUs` – Total duration of soft start and/or soft stop pulse train in microseconds; ignored in other PWM modes
- `pwmPeriodUs` – Period in microseconds; min 5 µs, max 254200 µs
- `pwmDutyCycle` – Duty cycle between 0.0 and 1.0

4.10.1.1 Add a Port

Using `Do0` in the resource string adds the entire digital output port to one channel.

```
//Configure DIO0:15 for output with no PWM. The last 3 parameters are
ignored when PWM is disabled.

doSession.CreateDOIndustrialChannel("pdna://192.168.100.2/Dev1/Do0",
   UeiDOPWMDisabled, 0, 0, 0);
```

All outputs in the channel are enabled by default. You can selectively enable/ disable outputs with a 16-bit output mask (LSB is DIO0).

NOTE: If you are simultaneously running a digital input session, ensure that the output mask is disabled (i.e. set to 0) for the input-only channels.

```
//Get pointer to output port (channel index = 0)

CUeiDOIndustrialChannel* doPort =
   dynamic_cast<CUeiDOIndustrialChannel*>(doSession.GetChannel(0));

//Enables output on DIO0:7. DIO8:15 are configured as input-only.

doPort->SetOutputMask(0xff);
```

PWM features are configurable on a line-by-line basis.

```
//Configure DIO1 for a soft start; period = 80us and duration = 2000us

doPort->SetPWMMode(1, UeiDOPWMSoftStart);
doPort->SetPWMPeriod(1, 80);
doPort->SetPWMLength(1, 2000);
```

4.10.1.2 Add Selected Lines

Alternatively, you can configure a subset of lines by specifying `Doline0` in the resource string and appending the desired line numbers.

```
//Configure DIO2:3 and DIO4:7 with 25% and 50% duty cycles respectively.

doSession.CreateDOIndustrialChannel("pdna://192.168.100.2/Dev1/Doline0/
   2:3", UeiDOPWMContinuous, 1000, 50, 0.25);
doSession.CreateDOIndustrialChannel("pdna://192.168.100.2/Dev1/Doline0/
   4:7", UeiDOPWMContinuous, 1000, 50, 0.5);
```

This approach creates one channel per line. Unlike a `Do0` line, each `Doline` is reconfigured using a unique channel index as follows:

```
//Get pointer to DIO4. DIO4 is ch3 in the list created above.

CUeiDOIndustrialChannel* dochannel =
dynamic_cast<CUeiDOIndustrialChannel*>(doSession.GetChannel(3));

//Set DIO3 period to 200 us (pass in 0 for the line parameter)

dochannel->SetPWMPeriod(0, 200);
```

However, even if you configured only a subset of lines, the output mask applies to all 16 lines. You can use the same output mask code shown in Section 4.10.1.1. It does not matter which channel calls SetOutputMask().

4.10.1.3 Configure Pull-up/down Resistors

You can connect a DIO line to Vcc and/or Gnd (Figure 2-4).

- `UeiDigitalTerminationNone` – no termination
- `UeiDigitalTerminationPullUp` – enable only pull-up resistor
- `UeiDigitalTerminationPullDown` – enable only pull-down resistor
- `UeiDigitalTerminationPullUpPullDown` – enable both pull-up and pull-down resistor

```
//Connect pull-up resistor between DIO1 and Vcc.
doPort->SetTermination(1, UeiDigitalTerminationPullUp);
```

4.10.1.4 PWM Modes

Choose one of the following options for the `pwmMode` input parameter:

- `UeiDOPWMDisabled` – disable PWM
- `UeiDOPWMSoftStart` – generate a pulse train after writing 1 if its previous state was 0. The PWM duty cycle gradually increases from 0% to `pwmDutyCycle` over `pwmLengthUs`.
- `UeiDOPWMSoftStop` – generate a pulse train after writing 0 if its previous state was 1. The PWM duty cycle gradually decreases from `pwmDutyCycle` to 0% over `pwmLengthUs`.
- `UeiDOPWMSoftBoth` – generate a pulse train for both a low-to-high and high-to-low transition.
- `UeiDOPWMContinuous` – continuously generates a pulse train with `pwmDutyCycle`. When writing to digital outputs, ensure that a 1 is written to any output that is configured for `UeiDOPWMContinuous` mode.
- `UeiDOPWMGated` – generates a pulse train with `pwmDutyCycle` only when a 1 is written to the output.

4.10.1.5 Configure PWM Push/ Pull

You can specify which FETs are switched by the PWM output:

- `UeiDOPWMOutputPush` – switch only high-side FET
- `UeiDOPWMOutputPull` – switch only low-side FET
- `UeiDOPWMOutputPushPull` – switch both FETs
- `UeiDOPWMOutputOff` – no PWM applied to either FET

```
//Enable PWM on only high-side FET of DIO1.
doPort->SetPWMOutputMode(1, UeiDOPWMOutputPush);
```

4.10.2 Write Data

Writing data is done using a a Digital Writer object. Digital data is written as a 16-bit integer. The writer updates all lines in the port, even if `Doline` configured only a subset of lines. FET-based outputs should be enabled using `SetOutputMask()`, else the data for those bits will be ignored.

```
//Create a writer object and link it to the session's data stream.
CUeiDigitalWriter doWriter(doSession.GetDataStream());
//Write a 1 on DIO15:8 and a 0 on DIO7:0.
uInt16 data = 0xff00;
doWriter.WriteSingleScan(&data);
```

4.10.3 Read Output Voltages

You can monitor digital outputs using an analog input session, as described in Section 4.9.3.

4.11 TTL Digital Input Session

The session may be configured to access the TTL digital input (`Di1`) subsystem.

4.11.1 Configure Input Port

The DNx-MF-101 has only one TTL input port, so the resource string should specify port 0 as shown in the code below. The TTL input port includes all four TTL lines and the TRIGIN line. Unlike an industrial digital session, you cannot configure a TTL session to only access a subset of lines.

```
//Configure session to read the TTL input port.
ttliSession.CreateDIChannel("pdna://192.168.100.2/Dev1/Di1/0");
```

4.11.2 Read Data

Reading data is done using a Digital Reader object. Digital data is stored in a 16-bit integer buffer. Bits 0:3 are TTL lines 0:3 and Bit 4 is TRIGIN. The other bits are currently reserved.

```
//Create a reader object and link it to the session's data stream.
CUeiDigitalReader diReader(ttliSession.GetDataStream());
//Read state of all lines in the port. A scan returns a 16-bit integer.
uInt16 data[1];
diReader.ReadSingleScan(data);
```

4.12 TTL Digital Output Session

The session may be configured to access the TTL digital output (`Do1`) subsystem.

4.12.1 Configure Output Port

The DNx-MF-101 has only one TTL output port, so the resource string should specify port 0 as shown in the example below. The TTL output port includes all four TTL lines and the TRIGOUT line. TRIGOUT is always enabled. TTL outputs are enabled or disabled in pairs (TTL0-1 and TTL2-3) using a bitwise mask that can be set by calling `SetOutputMask()`. If you try to enable only one line in a pair, both outputs will be enabled. Failure to enable the TTL lines will result in the data for those bits being ignored.

```
//Configure session to use the TTL output port.
ttloSession.CreateDOChannel("pdna://192.168.100.2/Dev1/Do1/0");
//Obtain pointer to the output channel (only one channel in this case).
CUeiDOChannel* dochannel =
   dynamic_cast<CUeiDOChannel*>(ttloSession.GetChannel(0));
//Enable output on TTL3 and TTL2. TTL1 and TTL0 are set as input-only.
dochannel->SetOutputMask(0xc);
```

4.12.2 Write Data

Writing data is done using a Digital Writer object. Digital data is written as a 16-bit integer: Bits 0:3 are TTL lines 0:3, Bit 4 is TRIGOUT, and the other bits are unused. TTL lines must first be enabled as described in Section 4.12.1.

```
//Create a writer object and link it to the session's data stream.
CUeiDigitalWriter doWriter(ttloSession.GetDataStream());
//Set TRIGOUT=1, TTL3=0, TTL2=0, TTL1=1, and TTL0=0.
uInt16 data = 0x12;
doWriter.WriteSingleScan(&data);
```

4.13 Counter Input Session

The session may be configured to access the counter input (`Ci`) subsystem.

4.13.1 Add Input Channels

The `CreateCIChannel()` method adds counter input channels and sets basic configuration parameters.

```
CUeiCIChannel* CreateCIChannel(std::string resource,
   tUeiCounterSource source, tUeiCounterMode mode,
   tUeiCounterGate gate, Int32 divider, Int32 inverted);
```

- `resource` – Resource string for counter 0 or counter 1
- `source` – Set CLKIN to either the internal 66MHz clock or an external input pin (Section 4.13.2)
- `mode` – Counting mode (Section 4.13.3)
- `gate` – Use either an external or a software gate to enable the counter
- `divider` – Prescaler divides source signal by this factor; default = 1
- `inverted` – TRUE to invert source signal

```
//Configure counter 0 to count events on an external pin.
//An internal gate starts the count immediately.
//Source is divided by 2 and not inverted.

ciSession.CreateCIChannel("pdna://192.168.100.2/Dev1/Ci0",
   UeiCounterSourceInput, UeiCounterModeCountEvents,
   UeiCounterGateInternal, 2, false);
```

4.13.2 Route Counter to DIO Pins

The counter's CLKIN, GATE, and CLKOUT lines can be internally routed to the following pins:

- **fetX** - Industrial DIO pins, e.g. "fet3" for DIO3
- **ttlX** - TTL DIO pins, e.g. "ttl3" for TTL3
- **trigin** - TRIGIN pin (CLKIN or GATE only)
- **trigout** - TRIGOUT pin (CLKOUT only)
- **syncX** - Sync pins 0-3 (CLKIN or GATE only)

The external CLKIN pin is only used when the counter is configured with `source = UeiCounterSourceInput`. Similarly, the GATE pin is only used when the counter is configured with `gate = UeiCounterGateExternal`.

```
//Obtain pointer to the input channel (only one channel in this case).

CUeiCIChannel* counter =
   dynamic_cast<CUeiCIChannel*>(ciSession.GetChannel(0));

//Route CLKIN to DIO5.
//Route GATE to DIO3.
//Route CLKOUT to TRIGOUT and TTL3.

counter->SetSourcePin("fet5");
counter->SetGatePin("fet3");
counter->SetOutputPins("trigout,ttl3");
```

You can set up an optional input debouncer for CLKIN and GATE. The maximum allowable value for the minimum pulse width is 7.94 ms. If a larger value is passed to either of these methods, a value of 7.94 ms will be used.

```
//Allow state change only when inputs have stayed stable for 1.0ms.

counter->SetMinimumSourcePulseWidth(1.0);
counter->SetMinimumGatePulseWidth(1.0);
```

For fetX inputs, you must also create a separate industrial digital input session (Section 4.9). This configures and starts up the A/D converter.

```
//Create new session.

CUeiSession diSession;

//Configure session to read from FET-based digital inputs.
//Low threshold = 2.0V, high threshold = 3.0V, debouncer interval = 1.0ms

diSession.CreateDIIndustrialChannel("pdna://192.168.100.2/Dev1/Di0",
   2.0, 3.0, 1.0);

//Configure timing for Point by Point DAQ mode.

diSession.ConfigureTimingForSimpleIO();
```

You do not need a separate session for TTL-level inputs (ttlX, trigin, trigout, syncX), nor do you need one for outputs. If you are simultaneously running a digital output session and want to read in external inputs, remember to disable the output mask on input-only lines. The counter session automatically overrides digital output session settings on output lines.

NOTE: CLKOUT should always be routed to an external pin, even if the counter is only used for input. CLKOUT remains high during a counter input session.

4.13.3 Counter Input Modes

Choose one of the following options for the `mode` parameter:

- `UeiCounterModeCountEvents` – Count pulses on an external pin, or use as a timer by counting internal clock cycles
- `UeiCounterModeBinCounting` – Count pulses over a user specified time interval (Section 4.13.3.1)
- `UeiCounterModeMeasurePulseWidth` – Count the number of 66 MHz clocks while the input signal is high
- `UeiCounterModeMeasurePeriod` – Count the number of 66 MHz clocks over the specified number of periods (Section 4.13.3.2). The number of clock ticks returned will actually have occurred over the specified number of periods plus 1, e.g., if 10 periods are specified, then the returned number of clock ticks will have occurred over 11 periods.
- `UeiCounterModeTimedPeriodMeasurement` – Measure the average period over a user-specified time interval (Section 4.13.3.1); period is returned as a number of 66 MHz clocks

- `UeiCounterModeQuadratureEncoder` – Quadrature encoder measurement; PWM signal on GATE controls the count direction
- `UeiCounterModeDirectionCounter` – Count up if GATE is high and count down if GATE is low

4.13.3.1 Set Capture Time Interval

The time interval for `UeiCounterModeBinCounting` and `UeiCounterModeTimedPeriodMeasurement` is configured using the session's timing object.

```
//Get pointer to session's timing object.
CUeiTiming* ciTiming = ciSession.GetTiming();
//Set frequency to 0.5 Hz; count is returned every 2.0 sec.
ciTiming->SetScanClockRate(0.5);
```

4.13.3.2 Set Number of Periods

In `UeiCounterModeMeasurePeriod` mode, the counter can be configured to measure the total duration of N+1 periods.

```
//Update the counter when 11 (N+1) periods have elapsed.
counter->SetPeriodCount(10);
```

The counter returns the previous measurement until the specified number of periods have been counted again.

4.13.4 Read Count Data

Reading data is done using a reader object. Digital data is stored in a 32-bit integer buffer.

```
//Create a reader object and link it to the session's data stream.
CUeiCounterReader ciReader(ciSession.GetDataStream());
//Read the current count value.
uInt32 data[1];
ciReader.ReadSingleScan(data);
```

4.14 Counter Output Session

The session may be configured to access the counter output (Co) subsystem.

4.14.1 Add Output Channels

The CreateCOChannel() method adds counter output channels and configures the shape of the output signal.

```
CUeiCOChannel* CreateCOChannel(std::string resource,
   tUeiCounterSource source, tUeiCounterMode mode,
   tUeiCounterGate gate, uInt32 tick1, uInt32 tick2,
   Int32 divider, Int32 inverted);
```

- resource – Resource string for counter 0 or counter 1
- source – Set CLKIN to either the internal 66MHz clock or an external input pin (Section 4.13.2)
- mode – Counting mode (Section 4.14.3)
- gate – Use either an external or a software gate to enable the counter
- tick1 – Number of counts for which output is low
- tick2 – Number of counts for which output is high
- divider – Prescaler divides source signal by this factor; default = 1
- inverted – TRUE to invert source signal

```
//Configure counter 0 to output pulse train (period=6ms, duty cycle=75%).
//Count ticks of an undivided, uninverted 66MHz source clock.
//An internal gate starts the output immediately.

coSession.CreateCOChannel("pdna://192.168.100.2/Dev1/Co0",
   UeiCounterSourceClock, UeiCounterModeGeneratePulseTrain,
   UeiCounterGateInternal, 100000, 300000,
   1, false)
```

4.14.2 Route Counter to DIO Pins

Refer to Section 4.13.2 and the methods in the CUeiCOChannel class.

4.14.3 Counter Output Modes

Choose one of the following options for the mode parameter:

- UeiCounterModeGeneratePulse – Generate a single pulse
- UeiCounterModeGeneratePulseTrain – Generate a continuous pulse train
- UeiCounterModePulseWidthModulation – Generate a pulse width modulated waveform (same as GeneratePulseTrain)

4.14.4 Write Output Parameters

You can write new tick1 and tick2 values to the counter using a writer object. This is used to change the PWM period and/or duty cycle after the session has already been started.

```
//Create a writer object and link it to the session's data stream.
CUeiCounterWriter coWriter(coSession.GetDataStream());
//Buffer must be large enough to contain two 32-bit integers per channel.
uInt32 data[2]={20000, 5000};
//Set tick1 = 20000 (low duration)
//Set tick2 = 5000 (high duration)
coWriter.WriteSingleScan(data);
```

4.15 Diagnostics Session

The session may be configured to read diagnostic data from the Analog Output and Industrial DIO subsystems.

4.15.1 Add Input Channels

The `CreateDiagnosticChannel()` method adds the diagnostic channels specified in the resource string. The `Diag` subsystem supports the channel numbers listed in Table 4-3 plus a time stamp channel (Section 4.7.3).

```
//Configure session to read all AOut1 diagnostics.

diagSession.CreateDiagnosticChannel("pdna://192.168.100.2/Dev1/
Diag4:7");
```

Table 4-3 Diagnostic Channel Numbers

Channel #	Description
0	DAC temperature on AOut0
1	Voltage on AOut0
2	Voltage on AGnd0
3	DAC supply voltage on AOut0
4	DAC temperature on AOut1
5	Voltage on AOut1
6	Voltage on AGnd1
7	DAC supply voltage on AOut1
8	Voltage on DIO0
9	Voltage on DIO1
10	Voltage on DIO2
11	Voltage on DIO3
12	Voltage on DIO4
13	Voltage on DIO5
14	Voltage on DIO6
15	Voltage on DIO7
16	Voltage on DIO8
17	Voltage on DIO9
18	Voltage on DIO10
19	Voltage on DIO11
20	Voltage on DIO12
21	Voltage on DIO13
22	Voltage on DIO14

Table 4-3 Diagnostic Channel Numbers

Channel #	Description
23	Voltage on DIO15

To help keep track of the different channels in a session, you can retrieve an abbreviated description of each channel with `GetAliasName()`. The following example code returns the string 'temp_aout1' when used with the channel list created above.

```
//Retrieve name of first channel in the CreateDiagnosticChannel() list.
diagSession.GetChannel(0)->GetAliasName();
```

4.15.2 Read Data

Read diagnostic data the same way as you would in an analog input session. An Analog Raw Reader object returns the calibrated binary data, while an Analog Scaled Reader returns the data converted to °C or Volts. The following example code reads scaled temperature and voltage from a session with 4 channels.

```
//Create a reader object and link it to the session's data stream.
CUeiAnalogScaledReader diagReader(diagSession.GetDataStream());
//Buffer must be large enough to contain one sample per channel.
double data[4];
//Read one sample per channel.
diagReader.ReadSingleScan(data);
```

4.16 Serial Port Session

The session may be configured to access the RS-232/422/485 (`Com`) subsystem.

4.16.1 Configure the Port

The `CreateSerialPort()` method links the session to the serial port (Port 0), configures basic port settings, and returns a pointer to the port.

```
//Configure session for RS-232 serial communications @57600 bps.
//Each UART frame has 8 data bits, no parity bit, and 1 stop bit.
//No termination string is set.

CUeiSerialPort* port = serialSession.CreateSerialPort(
                                    “pdna://192.168.100.2/Dev1/Com0”,
                                    UeiSerialModeRS232,
                                    UeiSerialBitsPerSecond57600,
                                    UeiSerialDataBits8,
                                    UeiSerialParityNone,
                                    UeiSerialStopBits1,
                                    “”);
```

You can configure additional DNx-MF-101 serial port settings by calling the `CUeiSerialPort` methods summarized in **Table 4-4**. For example:

```
//Connect RX and TX signals internally and disable external signals.

port->EnableLoopback(TRUE);
```

Table 4-4 High-level API for Serial Port Configuration

Function	Description
`SetMode`	Set port to RS-232, RS-422, or RS-485 mode.
`SetSpeed`	Select a predefined baud rate or enable a custom rate.
`SetCustomSpeed`	Set a custom baud rate in bits per seconds.
`SetDataBits`	Set the number of data bits transferred per character. Each character is always stored as a byte in the FIFO.
`SetParity`	Set the type of parity bit.
`SetStopBits`	Set the number of stop bits.
`EnableLoopback`	Connect RX and TX signals internally and disable external signals.
`EnableErrorReporting`	Send a break, i.e. hold TX line at logic low. No errors are currently reported.
`EnableRxTerminationResistor`	Enable RS-485 termination resistor (91 Ω) between RX+ and RX-.
`EnableTxTerminationResistor`	Enable RS-485 termination resistor (91 Ω) between TX+ and TX-.
`SetCharDelay`	Set the delay between each character in microseconds.
`SetMinorFrameMode`	Set how characters are grouped into minor frames (Section 4.16.1.2).
`SetMinorFrameLength`	Set the number of characters in a minor frame (only used for fixed length frame mode).

Table 4-4 High-level API for Serial Port Configuration

`SetMinorFrameDelay`	Set the delay between minor frames in microseconds.
`SetMajorFramePeriod`	Set the repeat period for a major frame in microseconds.
`SetTermination`	Set the termination string used to define the end of a message (max 128 characters). A READ command stops when the termination string has been found. **NOTE:** Setting the termination string is currently only supported in low-level API. Framework support is under development.
`EnableHDEchoSuppression`	Stop RS-422 receiver from reading the transmitted characters.
`SetFlowControl`	Enable RS-232 hardware flow control. **NOTE:** Setting the watermark level is currently only supported in low-level API. Framework support is under development.

Refer to the `CUeiSerialPort` class definition and/or the "UeiDaq Framework Reference Manual" for more information about these functions and their accepted input parameters.

4.16.1.1 Configure Custom Baud Rate

The following example shows how to program a custom port speed. See **Table 1-6** for the maximum supported speeds.

```
//Set baud rate to 15000 bits per second.

port->SetSpeed(SerialBitsPerSecondCustom);
port->SetCustomSpeed(15000);
```

4.16.1.2 Configure Minor Frames

The DNx-MF-101 supports three possible ways of defining a minor frame:

1. **Fixed Length** – each minor frame is a fixed number of characters. For example:.

```
//Insert a 1000us delay after every 20 characters.

port->SetMinorFrameMode(UeiSerialMinorFrameModeFixedLength);
port->SetMinorFrameLength(20);
port->SetMinorFrameDelay(1000);
```

2. **Zero Character** – the end of a minor frame is indicated by an ASCII NUL character (0x00). The zero character is transmitted when it's the last character in a WRITE command.
3. **Variable Length** – the size of each minor frame is indicated by an extra character preceding the data characters. For example, if the write buffer contains `writeData={3, 0xa, 0xb, 0xc, 2, 0xd, 0xe}`, the following sequence will be transmitted: *0xa, 0xb, 0xc, delay, 0xd, 0xe*

4.16.1.3 Configure Flow Control

The DNx-MF-101 only supports hardware flow control mode. Data transmission stops when CTS is low, and RTS goes low when the RX FIFO reaches the RX watermark level.

```
//RX watermark is default 512 characters. Enable hardware flow control.
port->SetFlowControl(UeiSerialFlowControlRtsCts);
```

If the RX FIFO overflows when RTS Autoflow is enabled, the receiver stops receiving data until a hard reset is performed.

4.16.2 Read Data

Reading data from the RX FIFO is done using a reader object. The following sample code requests 10 bytes from the RX FIFO and returns the number of bytes actually read.

```
//Create a reader object and link it to the session's data stream.
CUeiSerialReader serialReader(serialSession.GetDataStream());
//Data buffer must be large enough to contain the number of bytes read.
char readData[10];
//Read up to 10 bytes from the RX FIFO.
serialReader.Read(10, readData, &numBytesRead);
```

The number of returned bytes may be less than the number of requested bytes if the RX FIFO is short on data or if the termination string has been found. The termination string can span across multiple READ commands. If one READ command returns the beginning of the termination string, the next command will watch for the remainder of the string.

4.16.3 Write Data

Writing data to the TX FIFO is done using a writer object. The following example commands a write of two bytes and returns the number of bytes actually written.

```
//Create a writer object and link it to the session's data stream.
CUeiSerialWriter serialWriter(serialSession.GetDataStream());
//Load two bytes of data into buffer.
char writeData[2] = {0x53, 0x54};
//Write 0x53 and 0x54 to TX FIFO.
//If numBytesWritten==2, both bytes fit into the TX FIFO.
serialWriter.Write(2, writeData, &numBytesWritten);
```

4.17 I2C Port Session

The session may be configured to access the `I2C` subsystem.

4.17.1 Configure the Master Module

The `CreateI2CMasterPort()` method links the session to the I^2C master module, configures the port speed, and returns a pointer to the master port. The DNx-MF-101 has only one port, so the resource string should specify Port 0.

```
//Create I2C master port and set clock rate to 100kHz.
//Use 5V TTL levels and disable CRC checking.

CUeiI2CMasterPort* masterport = i2cSession.CreateI2CMasterPort(
                                    "pdna://192.168.100.2/Dev1/I2C0",
                                    UeiI2CBitsPerSecond100K,
                                    UeiI2CTTLLevel5V,
                                    false);
```

The DNx-MF-101 does not support 3.3V TTL levels or secure shell mode; therefore, the last two parameters are ignored.

You can configure additional DNx-MF-101 master port settings by calling the `CUeiI2CMasterPort` methods summarized below in **Table 4-5**.

Table 4-5 High-level API for Master Port Configuration

Function	Description
`SetSpeed`	Select a predefined port speed (100kHz, 400kHz, 1MHz) or enable a custom speed.
`SetCustomSpeed`	Set a custom port speed (2kHz -100kHz).
`SetLoopbackMode`	Enable loopback between master and slave modules.
`SetByteToByteDelay`	Set the delay between bytes sent by master, i.e. the time between the falling edge of the ACK clock to the rising edge of the next clock. This delay is programmed with 1µs resolution up to a max of 490µs.
`SetMaxClockStretchingDelay`	Set the maximum time that any slave on the bus can stretch the clock. If a slave is still holding SCL low after this delay has elapsed, the master stops sending data and returns the bus to an idle state. This delay is a 16-bit value programmed with 1µs resolution.

4.17.1.1 Configure Custom Clock Rate

The following example programs the port to a custom clock rate. The DNx-MF-101 supports rates between 2 kHz to 100 kHz. Both master and slave modules share the same speed.

```
//Set clock rate to 25kHz.

masterport->SetSpeed(I2CBitsPerSecondCustom);
masterport->SetCustomSpeed(25000);
```

4.17.1.2 Configure Loopback

You can connect the master and slave modules while still generating external signals on the I^2C bus.

```
//Enable loopback between master and slave modules.

masterport->SetLoopbackMode(UeiI2CLoopbackRelay);
```

FPGA loopback mode is currently not supported on the DNx-MF-101.

4.17.2 Configure the Slave Module

The `CreateI2CSlavePort()` method links the session to the I^2C slave module, configures the slave address, and returns a pointer to the slave port. The DNx-MF-101 has only one port, so the resource string should specify Port 0.

```
//Create I2C slave port, use 7-bit address, and set address to 0x12.
//Only 5V TTL levels are supported.

CUeiI2CSlavePort* slaveport = i2cSession.CreateI2SlavePort(
                                    "pdna://192.168.100.2/Dev1/I2C0",
                                    UeiI2CTTLLevel5V,
                                    UeiI2CSlaveAddress7bit,
                                    0x12;
```

You can configure additional DNx-MF-101 slave port settings by calling the `CUeiI2CSlavePort` methods summarized below in **Table 4-6**.

Table 4-6 High-level API for Slave Port Configuration

Function	Description
`SetSlaveAddressWidth`	Configure slave to use either a 7-bit or 10-bit address.
`SetSlaveAddress`	Set the slave address.
`EnableBusMonitor`	Use slave as a Bus Monitor (Section 4.17.2.1).
`EnableBusMonitorAck`	Allow acknowledge generation (ACK) in Bus Monitor mode.
`SetClockStretchingDelay`	When clock stretching is enabled, slave extends the ACK time by this number of 15ns clocks. (Section 4.17.2.2).
`EnableAddressClockStretching`	Allow slave to stretch the clock when evaluating the address from the master.
`EnableTransmitClockStretching`	Allow slave to stretch the clock when sending data to the master.
`EnableReceiveClockStretching`	Allow slave to stretch the clock when processing data from the master.
`SetMaxWordsPerAck`	Set the max number of words the slave will receive from the master before issuing a NACK.
`SetSlaveRegisterData`	Load data into the slave's 32-bit TX register. (Section 4.17.2.3)

4.17.2.1 Configure Bus Monitoring

The slave port can be configured to store all data on the I²C bus, either with or without responding to the master.

```
//Enable bus monitoring mode.
slaveport->EnableBusMonitor(TRUE);
//Disable ACK generation.
slaveport->EnableBusMonitorAck(FALSE);
```

4.17.2.2 Configure Clock Stretching

The slave port can delay the next byte by holding the SCL line LOW during transactions. You can selectively enable clock stretching for address, transmit, or receive cycles. The delay is programmed as a 12-bit number in units of 15 nanosecond clocks (max delay time = 62µs).

```
//Set clock stretching delay to 45ns.
slaveport->SetClockStretchingDelay(3);
//Hold SCL low for 45ns between receiving the address and issuing an ACK.
slaveport->EnableAddressClockStretching(TRUE);
//Hold SCL low for 45ns between receiving the master's ACK and sending
//   the next byte.
slaveport->EnableTransmitClockStretching(TRUE);
//Hold SCL low for 45ns between receiving a byte and issuing an ACK.
slaveport->EnableReceiveClockStretching(TRUE);
```

4.17.2.3 Load Slave TX Register

You can load up to 4 bytes of data into the slave's data padding register, which is transmitted on repeat whenever the slave's FIFO is empty.

```
//Load slave TX register with 0x12345678.
slaveport->SetSlaveRegisterData(0,0x12);
slaveport->SetSlaveRegisterData(1,0x34);
slaveport->SetSlaveRegisterData(2,0x56);
slaveport->SetSlaveRegisterData(3,0x78);
```

See Section 2.5.3.1 for an example of the transmitted data.

NOTE: `UeiI2CSlaveDataModeRegister` transmit mode is not supported at this time. You cannot bypass the FIFO and send data directly from the TX register.

4.17.3 Read Data

Reading data from the I²C Port is done using a reader object. First, create a reader and set its channel parameter to Port 0:

```
//Create a reader object and link it to the session's data stream.
CUeiI2CReader i2cReader(i2cSession.GetDataStream(), 0);
```

You can use the same reader to read from master and slave RX FIFOs, as described below.

4.17.3.1 Slave RX Data

The following example code requests ten data words from the slave RX FIFO and returns the number of words actually read.

```
//Read up to 10 data elements from the slave RX FIFO.

tUeiI2CSlaveMessage slaveRxData[10];
i2cReader.ReadSlave(10, slaveRxData, &numElementsRead);
```

The `ReadSlave()` command parses the received 12-bit word (Section 2.5.3.2) and stores it in a `tUeiI2CSlaveMessage` data structure:

```
typedef struct _tUeiI2CSlaveMessage{
   tUeiI2CBusCode busCode; //4-bit bus condition
   uInt8 data;             //8-bit data
} tUeiI2CSlaveMessage;
```

4.17.3.2 Master RX Data

The master RX FIFO is empty until the master requests data from the slave. To obtain data, the master must first send an I^2C READ command to the slave using a writer object:

```
//Create a writer object to write commands.

CUeiI2CWriter i2cWriter(i2cSession.GetDataStream(), 0);

//Build I2C READ command.
//10 bytes are requested from the slave at address 0x2A.

tUeiI2CMasterCommand params;
params.type = UeiI2CCommandRead;
params.slaveAddress = 0x2A;
params.numReadElements = 10;

//Write command to I2C bus.

i2cWriter.WriteMasterCommand(&params);
```

After the transaction is complete, you can call `ReadMaster()` to retrieve the data from the master RX FIFO:

```
//Read up to 10 data elements from the master RX FIFO.

tUeiI2CMasterMessage masterRxData[10];
i2cReader.ReadMaster(10, masterRxData, &numElementsRead);
```

`ReadMaster()` parses the received 9-bit word (Section 2.5.2.3) and stores it in a `tUeiI2CMasterMessage` data structure:

```
typedef struct _tUeiI2CMasterMessage{
   uIint8 stopBit;         //stop bit marks the last word in the READ
   uInt8 data;             //8-bit data
} tUeiI2CMasterMessage;
```

4.17.4 Write Data

Writing data to the I^2C Port is done using a writer object. First, create a writer and set its channel parameter to Port 0:

```
//Create a writer object and link it to the session's data stream.

CUeiI2CWriter i2cWriter(i2cSession.GetDataStream(), 0);
```

You can use the same writer to write to both master and slave TX FIFOs, as described below.

4.17.4.1 Slave TX Data

The following example code commands a write of two bytes to the slave TX FIFO. and returns the number of bytes actually written. The slave transmits this data when replying to a master's READ command.

```
//Load two bytes of data into buffer (only lower 8 bits are used).

uInt16 slaveTxData[2] = {0xaa, 0xbb};

//Write 0xaa and 0xbb to slave TX FIFO.
//If numBytesWritten==2, both bytes fit into the TX FIFO.

i2cWriter.WriteSlaveData(2, slaveTxData, &numElementsWritten);
```

When the slave TX FIFO is empty, data is sent from the slave TX register (see Section 4.17.2.3).

4.17.4.2 Master TX Data

The master writes data to the slave by sending an I^2C WRITE command, as shown in the following example code:

```
//Build I2C WRITE command.
//2 bytes are written to the slave at address 0x2A.

tUeiI2CMasterCommand params;
params.type = UeiI2CCommandWrite;
params.slaveAddress = 0x2A;
params.numWriteElements = 2;
params.data[0] = 0xaa;
params.data[1] = 0xbb;

//Write command to I2C bus.

i2cWriter.WriteMasterCommand(&params);
```

Up to 255 bytes of data may be loaded into the `tUeiI2CMasterCommand` structure. You can send multiple WRITE commands in order to write up to 1024 words into the master TX FIFO.

4.18 Stop the Session

The session will automatically stop and clean itself up when the session object goes out of scope or when it is destroyed. To manually stop the session:

```
//Stop the session.

mySession.Stop();
```

To reuse the object with a different set of channels or parameters, you can manually clean up the session as follows:

```
//clean up session and free resources
mySession.CleanUp();
```

Chapter 5 Programming with Low-level API

This chapter provides the following information about programming the DNx-MF-101 using low-level API:

- About the Low-level API (Section 5.1)
- Example Code (Section 5.2)
- Data Acquisition Modes (Section 5.3)
- Point-by-Point API (Section 5.4)
- Async Events API (Section 5.5)
- RtDMap API (Section 5.6)
- RtVMap API (Analog IO) (Section 5.7)
- RtVMap API (Serial) (Section 5.8)
- AVMap API (Section 5.9)

5.1 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 or real-time operating systems.

When programming in Windows OS, we recommend that you use the UeiDaq high-level Framework API (see Chapter 4). The Framework simplifies the low-level API, making programming easier and faster while still providing access to the majority of low-level API features. Additionally the Framework supports a variety of programming languages and the use of scientific software packages such as LabVIEW and MATLAB.

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

- On Linux: *<PowerDNA-x.y.z>/docs*
- On Windows: *C:\Program Files (x86)\UEI\PowerDNA\Documentation*

NOTE: The DNx-MF-101 is supported in PowerDNA version 5.0.0.29+. If you're unsure if your version supports the board please contact Technical Support at uei.support@ametek.com.

5.2 Example Code

Application developers are encouraged to explore the self-documented source code examples to get started programming UEI products. The example code is located in the following directories:

- On Linux: *<PowerDNA-x.y.z>/src/DAQLib_Samples*
- On Windows: *C:\Program Files (x86)\UEI\PowerDNA\SDK\Examples*

The I/O board number is embedded in the name of the example code. For example, the Sample101 folder contains example code specific to the DNx-MF-101. The example code should run out of the box after inputting the IOM's IP address and the board's Device Number (DEVN).

5.3 Data Acquisition Modes

Table 5-1 lists the data acquisition (DAQ) modes available for transferring data between the DNx-MF-101 and the low-level user application.

Table 5-1 DAQ Modes Supported by the Low-level API

DAQ Mode	AIn	AOut	DIn	DOut	TTL	CT	Serial	I2C
Point-by-Point	●	●	●	●	●	●	●	●
ACB								
RtDMap	●	●	●	●	●	●		
RtVMap	●	●					●	
ADMap								
AVMap	●							

- **Point-by-Point:** Transfers one data point at a time to/from each configured channel of a single I/O board. Timing is controlled by the user application, which limits the transfer rate to 100 Hz. Point-by-Point mode is also known as immediate mode or simple mode.
- **Real-Time Data Map (RtDMap):** Transfers a packet containing one data point for each channel in the user-defined map. The newest data is transferred and old data is discarded. RtDMap is designed for closed-loop (control) applications and may include channels across multiple I/O boards.
- **Real-Time Variable Map (RtVMap):** Transfers a packet containing a variable number of data points per channel. RtVMap buffers the data and transfers the oldest data first. RtVMap is designed for closed-loop (control) applications and may include channels across multiple I/O boards.
- **Asynchronous Variable Map (AVMap):** Transfers a packet containing a variable number of data points per channel. AVMap buffers the data and transfers the oldest data first. AVMap is designed for closed-loop (control) applications and may include channels across multiple I/O boards. With AVMap, a hardware condition, e.g., a timer countdown, triggers data delivery.

ACB and ADMap are currently not supported on the DNx-MF-101.

Please refer to “FAQ - Data Acquisition Modes” for an overview and comparison of all the different acquisition modes offered by UEI. The *“PowerDNx Protocol Manual”* includes more detailed information about the protocols. Both of these documents are located in the directories listed in Section 5.1.

NOTE: Multiple subsystems (AIn, AOut, etc.) may be used together as long as they share the same DAQ mode. It is not possible to mix and match multiple DAQ modes on a single IO board, e.g., Point-by-Point serial messaging alongside VMap analog I/O.

5.3.1 Async Events Mode

The DNx-MF-101 supports asynchronous event handling. This event-driven mode runs in a separate thread alongside the selected DAQ mode. The firmware sends an event packet when a specific event occurs. Events on the DNx-MF-101 include:

- DIn periodic status
- DIn pin change of state

You can call any of the DAQ mode functions upon receiving the event.

5.4 Point-by-Point API

This section summarizes the low-level API used to configure, read from, and write to the DNx-MF-101 in Point-by-Point DAQ mode. The functions and parameters are described in detail in the *"PowerDNA API Reference Manual"*. Please see *Sample101* for a comprehensive example which includes typical initialization, error handling, and usage of these functions. The example splits the I/O subsystems into separate cases, making it easy to copy-paste different subsystems into a true multifunction application.

The information in this section is intended as a supplement to the example code and the API reference manual.

5.4.1 Analog I/O

Table 5-2 lists the low-level API for the DNx-MF-101 analog I/O subsystem. See *Sample101AnalogIn.c* and *Sample101AnalogOut.c* for example code.

Table 5-2 Low-level Analog I/O API

	Function	Description
Analog Input	`DqAdv101AIRead`	Return continuously sampled data from input channel.
	`DqAdv101AISetConfig`	Enable/disable voltage divider on input channel and configure moving average.
Analog Output	`DqAdv101AOWrite`	Write either floating point or raw values to output channel.
	`DqAdv101AOSetConfig`	Select voltage or current output mode and set range.
	`DqAdv101AOWriteWForm`	Load waveform data into output channel FIFO.
	`DqAdv101AOEnableWForm`	Enable/disable a waveform on output channel.
	`DqAdv101AOReadAdc`	Read back voltage and temperature from diagnostic ADCs.

5.4.2 Digital I/O

Table 5-3 lists the low-level API for the DNx-MF-101 digital I/O subsystems. See *Sample101DigitalIn.c*, *Sample101DigitalOut.c*, and *Sample101TTL.c* for example code.

Table 5-3 Low-level Digital I/O API

	Function	Description
Industrial DIn	`DqAdv101DIRead`	Read the current and debounced states on DIO lines.
	`DqAdv101DIReadAdc`	Read voltage on DIO lines.
	`DqAdv101DISetDebouncer`	Set debouncing interval for digital inputs.
	`DqAdv101DISetLevels`	Set low and high voltage levels for digital inputs.
	`DqAdv101DISetMovingAverage`	Set number of samples used to calculate moving average for every digital input ADC channel.
Industrial DOut	`DqAdv101DORead`	Read back the last state written to digital outputs.
	`DqAdv101DOSetPWM`	Configure pulse width modulation on digital outputs.
	`DqAdv101DOSetTermination`	Configure pull up/down resistors.
	`DqAdv101DOWrite`	Set digital output state to 0, 1, or turned off.
TTL DIO	`DqAdv101TTLRead`	Read the state of all TTL lines and TRIGIN.
	`DqAdv101TTLSetConfig`	Enable output on TTL lines in pairs.
	`DqAdv101TTLWrite`	Set state of TTL outputs and TRIGOUT.

5.4.3 Counters

Table 5-4 lists the low-level API for the DNx-MF-101 digital counter subsystem. See *Sample101CT.c* for example code.

Table 5-4 Low-level Counter API

	Function	Description
Counters	DqAdv101CTSetSource	Connect digital I/O pins to CLKIN and GATE.
	DqAdv101CTSetOutput	Connect one or more digital outputs to CLKOUT.
	DqAdv101CTStartCounter	Start counter if not using auto-start mode.
	DqAdv101CTClearCounter	Reset counter to the initial value in the load register.
	DqAdv101CTRead	Read data from a counter.
	DqAdv101CTWrite	Change CLKOUT signal by writing to CR0 and CR1.
	DqAdv101CTConfigCounter	Configure advanced counter settings.
	DqAdv101CTCfgForGeneralCounting	Configure counter as a general event counter or timer.
	DqAdv101CTCfgForBinCounter	Configure counter to count the number of events in a specific time interval.
	DqAdv101CTCfgForPeriodMeasurment	Configure counter to measure how long CLKIN is high and how long CLKIN is low over N periods.
	DqAdv101CTCfgForHalfPeriod	Configure counter to measure pulse width of CLKIN.
	DqAdv101CTCfgForTPPM	Configure counter to measure the average period of CLKIN over the user-defined time interval.
	DqAdv101CTCfgForQuadrature	Configure counter as a quadrature decoder; GATE pin defines direction of counting.
	DqAdv101CTCfgForPWM	Configure counter for PWM output.
	DqAdv101CTCfgForPWMTrain	Configure counter to output a set number of PWM pulses.

5.4.3.1 Configuration Settings

Each counter can be independently configured using either `DqAdv101CTConfigCounter()` or one of the `DqAdv101CTCfg___()` functions. `DqAdv101CTConfigCounter()` is the lowest level configuration function. However, since not all parameter combinations are supported in all modes, it is easier to use a `DqAdv101CTCfg___()` function when possible. `DqAdv101CTCfg___()` automatically selects the best counting mode for the application and only exposes relevant parameters. Table 5-5 lists the counter configuration parameters.

Table 5-5 Counter Configuration Parameters

Parameter	Description
startmode	Auto-start or start on DqAdv101CTStartCounter()
sampwidth	PWM sample width
ps	Prescaler value for clock division
pc	Period count register; used when measuring multiple periods
cr0	Compare register 0, CLKOUT is low between lr and cr0
cr1	Compare register 1, CLKOUT is high between cr0 and cr1

Table 5-5 Counter Configuration Parameters (Cont.)

Parameter	Description
tbr	Timebase register; used for timed measurements
dbg	Input debouncing gate register; GATE to be stable
dbc	Input debouncing clock register; defines time for CLKIN to be stable
iie	Invert CLKIN pin
gie	Invert GATE pin
oie	Invert CLKOUT pin
mode	Counting mode (Section 5.4.3.2)
trs	Use GATE as trigger
enc	Auto-clear counter after end_mode and await next trigger
gated	Use GATE to enable/disable counter, if GATE is not already being used as a trigger
re	Restart counter after end_mode condition is met
end_mode	Count termination condition (Section 5.4.3.3)
lr	Load register; sets initial value of the counter

5.4.3.2 Counting Modes

The following modes are selectable in `DqAdv101CTConfigCounter()`:

- **Basic timer** - counts the number of 66MHz clock cycles (or cycles of 66MHz divided by the prescaler). The output stays low as the counter counts from `lr` up to `cr0` and then stays high until it reaches `cr1`. The counter may be used as a Bin Counter or generate a One-Shot Output by selecting an appropriate end mode (Section 5.4.3.3).
- **External event counter** - similar to the Basic Timer, except the clock source is the debounced CLKIN signal rather than the 66MHz clock.
- **Timed Pulse Period Measurement** - counts the total number of rising CLKIN edges over the `tbr` time interval, as well as the total number of 66MHz clock cycles between the first and last rising edge. The average period can be computed from these two measurements.
- **Half-period capture** - counts the number of 66MHz clock cycles over which CLKIN is high. The pulse width can then be calculated.
- **N-period capture** - counts the number of 66MHz clock cycles for both the positive and negative parts of CLKIN until `pc`-1 number of periods have elapsed. The average period can then be calculated.
- **Quadrature Decoder** - counts the number of rising CLKIN edges, counting up if GATE=1 and down if GATE=0.

All modes except the Quadrature Decoder support an optional hardware trigger.

5.4.3.3 End Modes

The following count termination conditions are available:

- Count register reaches CR0 value
- Count register reaches CR1 value
- Count register reaches 0xFFFFFFFF
- Period count register reaches 0
- Timebase register reaches 0

- GATE goes from high to low

5.4.4 Serial Port

Table 5-6 lists the low-level API for the DNx-MF-101 RS-232/422/485 port subsystem. See *Sample101Serial.c* for example code.

Table 5-6 Low-level Serial Port API

	Function	Description
RS-232/422/485	`DqAdv101SerialSetConfig`	Set configuration properties for the serial port.
	`DqAdv101SerialClearFIFO`	Clear the input and/or output FIFOs.
	`DqAdv101SerialEnable`	Enable or disable serial port.
	`DqAdv101SerialReadRxFIFO`	Read data from the RX FIFO.
	`DqAdv101SerialReadRxFIFOEx`	Read data, timestamps, and status bits from the RX FIFO.
	`DqAdv101SerialWriteTxFIFO`	Write data to the TX FIFO.
	`DqAdv101SerialSendBreak`	Transmit a break of a specified duration.
	`DqAdv101SerialFlowControl`	Configure RS-232 RTS/CTS hardware flow control.

5.4.4.1 Configuring the Serial Port

The `DqAdv101SerialSetConfig()` function takes in port settings through an `MF101SERIALCFG` structure and populates a configuration card.

Supported `MF101SERIALCFG` structure members are listed in Table 5-7. Some parameters require a single value and some accept a logically grouped combination of constants. Refer to the *"PowerDNA API Reference Manual"* for a complete description of each parameter.

Table 5-7 Serial Port Configuration Parameters

Parameter	Description	Flag
flags	OR in flags to change associated parameters	n/a
baud_rate	desired baud rate	DQ_MF101_SERIAL_CFG_BAUD
mode	RS-232, 422, or 485	DQ_MF101_SERIAL_CFG_CHAN
loopback	=1 enable internal loopback	
stop_bits	number of stop bits	
parity	type of parity bit	
width	number of bits in each character	
break_en	=1 sets serial output to logical 0	
term_fs_tx_rx	=1 enables RS-485 termination resistors	
char_delay_src	delay between each character sent to FIFO	DQ_MF101_SERIAL_CFG_CHAR_DELAY
char_delay_us	clock source for char_delay_us	
frame_delay_mode	defines minor frame for frame_delay_us	DQ_MF101_SERIAL_CFG_FRAME_DELAY
frame_delay_length	number of characters in minor frame; only for FIXEDLEN delay mode	
frame_delay_src	clock source for frame_delay_us	
frame_delay_us	delay between minor frames	
frame_delay_repeat_us	repeat time between major frames	
term_buf	termination string	DQ_MF101_SERIAL_CFG_TERM_STRING
term_length	length of term_buf to use	
timeout	number of clocks without receiving data before timeout	DQ_MF101_SERIAL_CFG_TIMEOUT
timeout_clock	units for timeout	
tx_watermark	reserved	DQ_MF101_SERIAL_CFG_TX_WM
rx_watermark	RX FIFO watermark for data flow control	DQ_MF101_SERIAL_CFG_RX_WM
suppress_hd_echo	=1 suppresses echo in RS-422 mode	DQ_MF101_SERIAL_CFG_EXT
add_ts_on_idle	=1 adds timestamp to RX FIFO during idle state	

Note that `<flags>` defines what other parts of the configuration structure are valid. For example, the `DQ_MF101_SERIAL_CFG_CHAN` flag is required to switch the `mode`. If `DQ_MF101_SERIAL_CFG_CHAN` is not included in `<flags>`, then the parameter values associated with the flag are ignored and remain unchanged.

By using this strategy, configuration calls can be additive, so each following call adds or changes a parameter to the configuration card. Any untouched parameters are enabled with default values. To reset the entire configuration back to the default state, call `DqAdv101SerialSetConfig()` with only the `DQ_MF101_SERIAL_CFG_CLEAR` bit set in `<flags>`.

The settings on the configuration card take effect when `DqAdv101SerialEnable()` is called.

5.4.5 I²C Port

Table 5-8 lists the low-level API for the DNx-MF-101 I²C port subsystem. See *Sample101I2C.c* for example code.

Table 5-8 Low-level I²C Port API

	Function	Description
I²C	`DqAdv101I2CSetConfig`	Configure the I²C master and slave.
	`DqAdv101I2CFlush`	Clear the I²C port FIFO(s).
	`DqAdv101GetStatus`	Return the status of I²C subsystem.
	`DqAdv101I2CEnable`	Enable or disable the I²C port.
	`DqAdv101I2CBuildCmdData`	Build master command.
	`DqAdv101I2CCalcCustomTiming`	Calculate timing parameters for custom clock rate.
	`DqAdv101I2CMasterReadRxFIFO`	Read data from the master RX FIFO.
	`DqAdv101I2CMasterWriteTxFIFO`	Write I²C commands to the master TX FIFO (command mode).
	`DqAdv101I2CMasterWriteTxPhyFIFO`	Write low-level instructions to the master TX FIFO (raw mode).
	`DqAdv101I2CSlaveReadRxFIFO`	Read data from the slave RX FIFO.
	`DqAdv101I2CSlaveWriteTxFIFO`	Write data to the slave TX FIFO.

5.4.5.1 Configuring the I²C Port

The `DqAdv101I2CSetConfig()` function takes in master and slave settings through an `MF101I2CCFG` structure and populates a configuration card. The settings on the configuration card take effect when `DqAdv101I2CEnable()` is called.

`MF101I2CCFG` structure members are listed in Table 5-9. Some parameters require a single value and some accept a logically grouped combination of constants. All members are uint32, although some values are at most 12 or 16 bits. Refer to the "PowerDNA API Reference Manual" for a complete description of each parameter.

Table 5-9 I²C Configuration Parameters

Parameter	Description	Flag
`flags`	OR in flags to change associated parameters	n/a
`clock`	clock frequency	`DQ_MF101_I2C_CFG_CLOCK`
`master_cfg`	select active master settings	`DQ_MF101_I2C_CFG_MASTER_VALID`
`master_idle_delay`	delay before acquiring bus in MM mode, per 15ns	
`master_byte_delay`	delay between bytes sent by master (µs)	
`master_max_sync_delay`	max time that a slave can delay the clock (µs)	
`master_to_cfg`	max timeout before releasing bus (µs)	
`slave_cfg`	select active slave settings	`DQ_MF101_I2C_CFG_SLAVE_VALID`
`slave_sync_dly`	time slave delays clock during ACK, per 15ns	
`slave_ack_dly`	time to wait for the ACK clock, per 15ns	
`slave_max_ack`	max # of RX words before issuing NACK	
`slave_addr`	set 7 or 10-bit slave address	`DQ_MF101_I2C_CFG_SDATA_ADDR`
`slave_data`	data to send when slave TX FIFO is empty	

Note that `<flags>` defines what other parts of the configuration structure are valid. For example, the `DQ_MF101_I2C_CFG_MASTER_VALID` flag is required to change `master_byte_delay`. If `DQ_MF101_I2C_CFG_MASTER_VALID` is not included in `<flags>`, then the master configuration parameters are ignored and remain unchanged.

By using this strategy, configuration calls can be additive, so each following call adds or changes a parameter to the configuration card. Untouched parameters are configured to default values. To reset the entire configuration back to the default state, call `DqAdv101I2CSetConfig()` with only the `DQ_MF101_I2C_CFG_CLEAR` bit set in `<flags>`.

Custom Clock Settings

`DqAdv101I2CSetConfig()` also accepts an optional `MCTPARAM` structure for configuring custom clock rates. The current implementation limits custom clock rates to between 2 kHz and 100 kHz. These rates are intended for custom, slower devices or for systems with a large bus capacitance. `MCTPARAM` may be set to 0 if using typical 100 kHz, 400 kHz, or 1 MHz rates.

To set up a custom clock rate, initiate an `MCTPARAM` structure and use the `DqAdv101I2CCalcCustomTiming()` helper function to fill out the structure with the proper timing parameters. Then, in the `MF101I2CCFG` structure, use `DQ_MF101_I2C_CFG_CLOCK_CUST` for `<clock>` and make sure the `DQ_MF101_I2C_CFG_CLOCK` bit is included in `<flags>`.

```
//Enable custom clock rate.

MF101I2CCFG i2c_cfg = {0};
i2c_cfg.flags = DQ_MF101_I2C_CFG_CLOCK;
i2c_cfg.clock = DQ_MF101_I2C_CFG_CLOCK_CUST;

//Generate timing parameters for 50kHz (=100kHz base clock divided by 2).

MCTPARAM i2c_timing = {0};
DqAdv101I2CCalcCustomTiming(hd, 2, &i2c_timing);

//Write configuration to device.

DqAdv101I2CSetConfig(hd, devn, &i2c_cfg, &i2c_timing);
```

5.4.5.2 Command vs. Raw Mode

There are two ways the master can control the I^2C bus - command and raw modes.

In command mode, users use `DqAdv101I2CBuildCmdData()` to construct uint32 words from a list of pre-defined commands (Section 2.5.2.1). The built command words are then written to the outgoing FIFO using `DqAdv101I2CMasterWriteTxFIFO()`.

In raw mode, users use `DqAdv101I2CMasterWriteTxPhyFIFO()` to write atomic commands to the master TX FIFO. PHY stands for physical — the lowest accessible level of the I^2C state machine implemented on the FPGA. Available raw mode commands are listed in Table 5-10.

Table 5-10 Raw Mode Commands

Raw Mode Command	Description
`I2C_MRAW_PHY_TX1(B)`	Set SDA to 1 for one clock (use B=0)
`I2C_MRAW_PHY_TX0(B)`	Set SDA to 0 for one clock (use B=0)
`I2C_MRAW_PHY_RX(B)`	Set SDA to 1 for one clock, return SDA at the falling edge of the clock (use B=0)
`I2C_MRAW_PHY_START(B)`	START condition on the bus (use B=0)
`I2C_MRAW_PHY_STOP(B)`	STOP condition on the bus (use B=0)
`I2C_MRAW_PHY_RELEASE(B)`	Release bus without creating START or STOP conditions (use B=0)
`I2C_MRAW_DLY_2NUS(B)`	Delay for 2^B µS (B=0…31)
`I2C_MRAW_DLY_NX8US(B)`	Delay for B*8 µS (B=0…255)
`I2C_MRAW_BYTE_SEND(B)`	Transmit data byte B to the bus (B=0…255)
`I2C_MRAW_BYTE_RECEIVE(B)`	Read byte of data from the bus and save to master RX FIFO (use B=0)
`I2C_MRAW_ACK_WAIT(B)`	Wait for ACK; NACK ends sequence (use B=0)
`I2C_MRAW_SEQ_END(B)`	Ends sequence; bus is idle until this command initiates the write (use B=0)

Refer to *Sample101I2C.c* for examples of START+WRITE and START+READ sequences implemented in raw mode. An example START+WRITE+ReSTART+READ sequence is shown below.

Example:

START+WRITE+ReSTART+READ sequence in raw mode

```
I2C_MRAW_PHY_START(0);      // start
I2C_MRAW_BYTE_SEND(slave_addr<<1);   // 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(((slave_addr<<1)|1); // address + read
I2C_MRAW_ACK_WAIT(0);       // slave ACK
I2C_MRAW_BYTE_RECEIVE(0);   // store byte in master RX FIFO
I2C_MRAW_PHY_TX1(0);        // master NACK
I2C_MRAW_PHY_STOP(0);       // stop
I2C_MRAW_SEQ_END(0);        // write sequence to the bus
```

To switch from command mode to raw mode, edit the `MF101I2CCFG` configuration structure to include the `DQ_MF101_I2C_MCFG_RAWMODE` bit in `<master_cfg>`. The `DQ_MF101_I2C_CFG_MASTER_VALID` bit should also be set in `<flags>`. Please note that clock stretching and timeout parameters in `DqAdv101I2CSetConfig()` remain in full effect.

5.5 Async Events API

Most asynchronous event-handling functions are board-agnostic and described in the *"PowerDNA API Reference Manual"*. There is only one function specific to the DNx-MF-101. Please see *SampleAsync101* for an example of how to configure and retrieve event packets.

Table 5-11 Low-level Asynchronous Events API

	Function	Description
Async	`DqAdv101ConfigEvents`	Configure the board to send status data upon one of the following events: • DIO pin changes state • Periodically at a user-defined rate

5.6 RtDMap API

Real Time Data Map (RtDMap) mode uses the same API as Point-by-Point mode for channel configuration (Section 5.4); however, generic DMap functions are used for reading data. The DMap API is documented in the *"PowerDNA API Reference Manual"*.

Refer to *SampleRTDMap101* for an example of how to set up a Data Map on the DNx-MF-101. Table 5-12 lists the DNx-MF-101 channels that can be added to the DMap.

Table 5-12 DMap Channels

Subsystem	Channels	Notes
DQ_SS0IN	DQ_LNCL_TIMESTAMP	Read timestamp.
DQ_MF101_SS_AI	0...15 for single-ended channels	Read analog inputs; See DqAdv101AIRead() for channel gain configuration details.
	0...7 for differential channels	
DQ_MF101_SS_AO	0...1	Write to analog outputs.
DQ_MF101_SS_DI	DQ_MF101_DMAP_DI_FET_STATE	Read FET-based DIO port (16 bits).
	DQ_MF101_DMAP_DI_FET_DEB	Read debounced FET-based DIO port.
	DQ_MF101_DMAP_DI_TTL	Read TTL DIO port; Bits 0:3 are TTL0:3 and Bit 4 is TRIGIN
	DQ_MF101_DMAP_DI_CT_0	Read counter 0; See Section 5.4.3 and *Sample101CT.c* for configuration details.
	DQ_MF101_DMAP_DI_CT_1	Read counter 1; See Section 5.4.3 and *Sample101CT.c* for configuration details.
DQ_MF101_SS_DO	DQ_MF101_DMAP_DO_FET	Set state of FET-based digital outputs.
	DQ_MF101_DMAP_DO_TTL	Set state of TTL digital outputs.
DQ_MF101_SS_GUARDIAN	DQ_MF101_DMAP_GUARD_DI_ADC_CHAN	Read voltage on FET-based DIO; OR in the desired channel number (0...15).

A basic overview of DMap usage is provided in Section 5.6.1. More information on RtDMap is available in the *"PowerDNx Protocol Manual"*.

5.6.1 DMap Tutorial

As shown in *SampleRtDMap101*, a DMap program is structured as follows:

DMap Configuration:

1. Create a DMap.
2. Configure input/output channels.
3. Add input/output channels to the DMap.
4. Configure DNx-MF-101 scan rates.
5. Start the DMap.

DMap Operation:

6. Schedule output data to write upon next refresh.
7. Refresh the DMap.

8. Read retrieved data from input channels (returned in reply to refresh).

Close Out DMap:

9. Stop and close the DMap.

5.6.1.1 DMap Configuration

1. To create a new DMap, call `DqRtDmapInit()`. One copy of the DMap is stored on the IOM and another is stored on the host. During operation (Step 8), the IOM will update its version of the map at the rate specified during initialization.

```
//Create and initialize a DMap with a 1000 Hz refresh rate.

DqRtDmapInit(hd, &dmapid, 1000);
```

2. Configure I/O channels using the Point-by-Point API. This tutorial will focus on analog I/O; additional subsystems are covered in the example code.

```
//Optionally configure moving average

DqAdv101AISetConfig(hd, DEVN, 0, DQ_MF101_AI_MAV_1);

//Configure 16 single-ended input channels for range +-10V.
//Set up an input channel list.

for(ch=0; ch<16; ch++){
   input_cl[ch] = ch | DQ_LNCL_GAIN(DQ_MF101_AI_GAIN_1);
}

//Configure 2 analog output channels for range +-5V.
//Set up an output channel list.

for(ch=0; ch<2; ch++){
   DqAdv101AOSetConfig(hd, DEVN, ch, DQ_MF101_AO_RANGE_PN_5V);
    output_cl[ch] = ch;
}
```

3. Add the channels to the DMap with their corresponding subsystem names (Table 5-12).

```
//Add analog input channels to the DMap.

DqRtDmapAddChannel(hd, dmapid, DEVN, DQ_MF101_SS_AI, &input_cl, 16);

//Add analog output channels to the DMap.

DqRtDmapAddChannel(hd, dmapid, DEVN, DQ_MF101_SS_AO, &output_cl, 2);
```

4. By default, all boards in the DMap are clocked at the DMap refresh rate (set in Step 1). You can override this setting and specify a different sampling rate for the DNx-MF-101:

```
//Set the device scan rate to 100 Hz.

DqRtDmapSetSamplingRate(hd, dmapid, DEVN, 100);
```

5. Start the DMap with the configuration and channels requested above.

```
//Start the DMap.

DqRtDmapStart(hd, dmapid);
```

5.6.1.2 DMap Operation

6. `DqRtDmapWriteRawData()` or `DqRtDmapWriteScaledData()` writes output channel values to the host map. The DMap can hold one data point per channel. However, data is not actually transferred to the IOM until the `DqRtDmapRefresh()` call in Step 7.

```
//Copy AO data to output packet (-2.5V to AOut0 and +7.5V to AOut1).

double fdata[2] = {-2.5, 7.5};
DqRtDmapWriteScaledData(hd, dmapid, DEVN, fdata, 2);
```

7. Calling `DqRtDmapRefresh()` sends the output data from the host to the IOM. On the reply, the IOM transfers one data point per configured input channel to the host.

```
//Send output data and receive input data.

DqRtDmapRefresh(hd, dmapid);
```

8. Input data can be read from the host's version of the map using `DqRtDmapReadRawData()` or `DqRtDmapReadScaledData()`.

```
//Read analog input voltage from DMap.

DqRtDmapReadScaledData(hd, dmapid, DEVN, fdata, 16);
```

5.6.1.3 Close Out DMap

9. Stop and clean up the DMap with the calls:

```
DqRtDmapStop(hd, dmapid);

DqRtDmapClose(hd, dmapid);
```

5.7 RtVMap API (Analog IO)

VMap uses the same API as Point-by-Point mode for channel configuration (Section 5.4); however, generic VMap functions are used for reading data. The VMap API is documented in the *"PowerDNA API Reference Manual"*.

Refer to *SampleVMap101* for an example of how to set up and run a Variable Map (VMap) for analog input and output on the DNx-MF-101. Table 5-13 lists all of the DNx-MF-101 channels that can be added to the VMap.

Table 5-13 VMap Channels

Subsystem	Channels	Notes
`DQ_MF101_SS_AI`	0...15 for single-ended channels	Read analog inputs with timestamping; See `DqAdv101AIRead()` for channel gain configuration details.
	0...7 for differential channels	
`DQ_MF101_SS_AO`	0...1	Write to analog outputs.

A basic overview of VMap usage is provided in Section 5.7.1. More detailed information on RtVMap can be found in the *"PowerDNx Protocol Manual"*.

5.7.1 VMap Tutorial

As shown in *SampleVMap101*, a VMap program is structured as follows:

VMap Configuration:

1. Create a VMap.
2. Configure input/output channels.
3. Add input/output channels to the VMap.
4. Configure DNx-MF-101 scan rates.
5. Start the VMap.

VMap Operation:

6. Schedule output data to write upon next refresh.
7. Schedule input data to read upon next refresh.
8. Refresh the VMap.
9. Read retrieved data from input channels (returned in reply to refresh).

Close Out VMap:

10. Stop and close the VMap.

5.7.1.1 VMap Configuration

1. To create a new VMap, call `DqRtVmapInit()`. One copy of the VMap is stored on the IOM and another is stored on the host. During operation (Step 8), the IOM will update its version of the map at the rate specified during initialization.

```
//Create and initialize a VMap with a 1000 Hz refresh rate.
DqRtVmapInit(hd, &vmapid, 1000);
```

2. Configure analog I/O channels and set a VMap flag for each channel (required in Step 3):.

```
//Configure 16 single-ended input channels for range +-10V.
//Set up flag array for retrieving FIFO state.

for(ch=0; ch<16; ch++){
   in_cl[ch] = ch | DQ_LNCL_GAIN(DQ_MF101_AI_GAIN_1);
    in_flags[ch] = DQ_VMAP_FIFO_STATUS;
}

// Optionally configure voltage divider and moving averages

DqAdv101AISetConfig(hd, DEVN, AI_DIVIDER_MASK, AI_MOVING_AVERAGES);

//Configure 2 analog output channels for range +-5V.
//Set up flag array for retrieving FIFO state.

for(ch=0; ch<2; ch++){
   DqAdv101AOSetConfig(hd, DEVN, ch, DQ_MF101_AO_RANGE_PN_5V);
    out_cl[ch] = ch;
}
```

3. Add the channels to the VMap with their corresponding subsystem names (Table 5-13). The `DqRtVmapSetChannelList()` function identifies the number of physical channels on the DNx-MF-101.

```
//Add analog I/O channels to the VMap.

DqRtVmapAddChannel(hd, vmapid, DEVN, DQ_MF101_SS_AI, in_cl, in_flags, 1);
DqRtVmapAddChannel(hd, vmapid, DEVN, DQ_MF101_SS_AO, out_cl, out_flags,
   1);

//Specify number of physical channels per subsystem.

DqRtVmapSetChannelList(hd, vmapid, DEVN, DQ_MF101_SS_AI, in_cl, 16);
DqRtVmapSetChannelList(hd, vmapid, DEVN, DQ_MF101_SS_AO, out_cl, 2);
```

4. The DNx-MF-101 board is clocked according to the rates set by `DqRtVmapSetScanRate()`. Since there are 16 configured channels plus 1 automatically added timestamp channel, the board's Input FIFO fills at `IN_SCANRATE`*17. `OUT_SCANRATE` defines the overall rate at which the board's Output FIFO empties; you can fill the FIFO with a chunk of Channel 0 data followed by a chunk of Channel 1 data, or the two channels can be interleaved.

```
//Set the device scan rate.

DqRtVmapSetScanRate(hd, vmapid, DEVN, DQ_MF101_SS_AI, IN_SCANRATE);
DqRtVmapSetScanRate(hd, vmapid, DEVN, DQ_MF101_SS_AO, OUT_SCANRATE);
```

5. Start the VMap with the configuration and channels requested above.

```
//Start the VMap.

DqRtVmapStart(hd, vmapid);
```

5.7.1.2 VMap Operation

6. `DqRtVmapAddOutputData()` writes raw Analog Output values to the host's version of the map. If passing raw data directly into `DqRtVmapAddOutputData()`, you must logical OR the raw data with `DQ_MF101_CLO_AO_CHAN(ch)`, where `ch` is the channel number. `DqAdvScaleToRawValue()` does this operation automatically. The `DqNtohl()` helper function reverses the byte order from little endian (used with Intel-based host computers) to big endian (network data representation). This conversion is not handled automatically because VMap packets can contain data from many different layer types.

```
//Prepare to send 100 data points per output channel.

for (i=0; i<100; i++){
   for(ch=0; ch<2; ch++){
      DqAdvScaletoRawValue(hd, DEVN, out_cl[ch], out_fdata[i*2+ch],
          &out_bdata[i*2+ch]);
       out_bdata[i*2+ch] = DqHtonl(hd, out_bdata[i*2+ch]);
   }
}

//Copy data to the output packet.
//(the AO subsystem was added after AI, so its VMap index = 1)

DqRtVmapAddOutputData(hd, vmapid, 1, 200 * sizeof(uint32),
   &updates_accepted, (uint8*)out_bdata);
```

Note that data is not actually transferred to the IOM until the `DqRtVmapRefresh()` call in Step 8.

7. Use `DqRtVmapRqInputDataSz()` to schedule a request for data from the IOM. You can request a variable number of data points per channel. Note that data is not actually received until the `DqRtVmapRefresh()` call in Step 8.

```
//Request 1000 data points per input channel, including timestamp.
//(the AI subsystem was configured first, so its VMap index = 0)

DqRtVmapRqInputDataSz(hd, vmapid, 0, 17000*sizeof(uint32),
   &in_act_size, NULL);
```

8. The VMap request has been prepared, so the command can be sent with `DqRtVmapRefresh()`. During the refresh,

- The host transfers Analog Output data to the board's Output FIFO in an Ethernet packet.
- Analog Input data is transferred from the board's Input FIFO to the host in one Ethernet packet.

```
//Send output data and receive input data.

DqRtVmapRefresh(hd, vmapid, 0);
```

If a FIFO overflow error occurs, try reducing `IN_SCANRATE`, increasing `OUT_SCANRATE`, or increasing the `DqRtVmapRefresh()` rate.

9. Input data can be read from the host's version of the map using `DqRtVmapGetInputData()`. On Intel-based host computers, the received data will need to be converted from big endian to little endian.

```
//Read analog input and timestamp data from VMap.
//(the AI subsystem was configured first, so its VMap index = 0)

DqRtVmapGetInputData(hd, vmapid, 0, 17000* sizeof(uint32),
    &in_data_size, &in_avl_size, (uint8*)in_bdata)

//Reverse byte order from Network to Host representation.

for (i = 0; i < (in_data_size / (int)sizeof(uint32)); i++) {
    recv_data = DqNtohl(hd, in_bdata[i]);
}
```

5.7.1.3 Close Out VMap

10. Stop and clean up the VMap with the calls:

```
DqRtVmapStop(hd, vmapid);

DqRtVmapClose(hd, vmapid);
```

5.8 RtVMap API (Serial)

VMap uses the same API as Point-by-Point mode for channel configuration (Section 5.4); however, generic VMap functions are used for reading data. The VMap API is documented in the *"PowerDNA API Reference Manual"*.

Refer to *SampleVMap101Serial* for an example of how to set up and run a Variable Map (VMap) for serial communication on the DNx-MF-101. Table 5-14 lists the DNx-MF-101 channels that can be added to the VMap.

Table 5-14 VMap Subsystems and Channels for Serial Communication

Subsystem	Channels	Notes
DQ_MF101_ VMAP_SS_CHAN_IN	DQ_MF101_VMAP_CHAN_IN_SERIAL	Read serial input data.
DQ_MF101_ VMAP_SS_CHAN_OUT	DQ_MF101_VMAP_CHAN_OUT_SERIAL	Write serial output data.

A basic overview of VMap usage for serial communication is provided in Section 5.8.1. More detailed information on RtVMap can be found in the *"PowerDNx Protocol Manual"*.

5.8.1 VMap Tutorial (Serial)

As shown in *SampleVMap101Serial*, a VMap program for serial communication is structured as follows:

VMap Configuration:

1. Prepare configuration and set up channel list.
2. Create a VMap.
3. Add input/output channels to the VMap.
4. Start the VMap.

VMap Operation:

5. Prepare to write and read data upon next refresh.
6. Refresh the VMap.
7. Read input data from host's version of the VMap.

Close Out VMap:

8. Stop and close the VMap.

5.8.1.1 VMap Configuration

1. Prepare for serial communication by setting configuration properties, enabling the serial port, and setting up the channel list and flags.

```
//Set the configuration properties and enable the serial port.
//Note that SetSerialConfiguration() calls DqAdv101SerialSetConfig()
//and DqAdv101SerialEnable()

SetSerialConfiguration(hd, DEVN);

// Set up the channel list and flags

cl_in[0] = DQ_MF101_VMAP_CHAN_IN_SERIAL;
cl_out[0] = DQ_MF101_VMAP_CHAN_OUT_SERIAL;
flags_in[0] = DQ_VMAP_FIFO_STATUS;
flags_out[0] = DQ_VMAP_FIFO_STATUS;
```

2. Create a new VMap, by calling `DqRtVmapInit()`. One copy of the VMap is stored on the IOM and another is stored on the host. The refresh rate parameter is ignored for serial communication.

```
//Create and initialize a VMap

DqRtVmapInit(hd, &vmapid, 0);
```

3. Add the channels to the VMap with their corresponding subsystem names (Table 5-13), channel lists, and flags.

```
//Add serial I/O channels to the VMap.

DqRtVmapAddChannel(hd, vmapid, DEVN, DQ_MF101_VMAP_SS_CHAN_IN,
                                     cl_in, flags_in, 1);
DqRtVmapAddChannel(hd, vmapid, DEVN, DQ_MF101_VMAP_SS_CHAN_OUT,
                                     cl_out, flags_out, 1);
```

4. Start the VMap with the configuration and channels requested above.

```
//Start the VMap.

DqRtVmapStart(hd, vmapid);
```

5.8.1.2 VMap Operation

Execute the following steps in the VMap Operation section until there is a terminating condition.

5. Prepare to write and read serial data at the next refresh of the VMap.

```
//Prepare output data.

len = sprintf((char*)(&out_data[0]), "output string example");

// Write bytes to be sent at next refresh

DqRtVmapWriteOutput(hd, vmapid, DEVN, cl_out[0], len, out_data);

// Request the max number of bytes to receive at next refresh

DqRtVmapRequestInput(hd, vmapid, DEVN, cl_in[0], MAX_RX_MESSAGES);
```

6. Refresh the VMap.

```
// Write output data to each TX port FIFO and Read each RX port FIFO

DqRtVmapRefresh(hd, vmapid, 0);
```

7. Read input data from the host's version of the map using `DqRtVmapReadInput()`.

```
// Read data received during the last refresh
// To treat the data as a string, add a NULL character to the end of
// in_data

DqRtVmapReadInput(hd, vmapid, DEVN, cl_in[0], MAX_RX_MESSAGES,
                                    &rx_data_size, in_data);
```

5.8.1.3 Close Out VMap

8. Stop and clean up the VMap with the calls:

```
DqRtVmapStop(hd, vmapid);

DqRtVmapClose(hd, vmapid);
```

5.9 AVMap API

Asynchronous Variable Map (AVMap) uses the same API as Point-by-Point mode for channel configuration (Section 5.4); however, generic AVMap functions are used for reading data.

Refer to *SampleAVMap101* for an example of how to set up and run an AVMap on the DNx-MF-101. The example program also provides more detail on declaring and initializing the variables used in the following tutorial. Table 5-15 lists the DNx-MF-101 channels that can be added to the AVMap.

Table 5-15 AVMap Channels

Subsystem	Channels	Notes
`DQ_MF101_SS_AI (DQ_SS0IN)`	ch | DQ_LNCL_GAIN (DQ_MF101_AIGAIN_1)	Channel ORed with the gain bits (bits 8-11) For differential channels, OR in DQ_LNCL_DIFF

5.9.1 AVMap Tutorial

This section provides a basic overview of AVMap usage. As shown in *SampleAVMap101*, an AVMap program is structured as follows:

AVMap Configuration:

1. Create a VMap.
2. Configure input channels, voltage divider, and moving averages.
3. Add input channels to the VMap.
4. Set the channel list and scan rates.
5. Start the VMap.

AVMap Operation:

6. Schedule input data to read upon next refresh.
7. Refresh the VMap and get data

Close Out AVMap:

8. Stop and close the VMap.

5.9.1.1 AVMap Configuration

1. To create a new AVMap, call `DqRtVmapInit()`.

```
//Create the VMap

DqRtVmapInit(hd, &vmapid, XMAPRATE);
```

2. Configure input channels and optionally configure voltage divider and moving averages.

```
//Configure input channels

for (ch = 0; ch < AI_CHANNELS; ch++) {
    //Build AI channel list. For differential bitwise OR in DQ_LNCL_DIFF.
    in_cl[ch] = ch | DQ_LNCL_GAIN(AI_GAIN) /* | DQ_LNCL_DIFF */;
    in_flags[ch] = DQ_VMAP_FIFO_STATUS;
}

//Optionally configure voltage divider and moving averages

DqAdv101AISetConfig(hd, DEVN, AI_DIVIDER_MASK, AI_MOVING_AVERAGES);
```

3. Add the channels to the VMap with their corresponding subsystem names (Table 5-15).

```
//Add channels to the VMap

DqRtVmapAddChannel(hd, vmapid, DEVN, DQ_MF101_SS_AI, in_cl,
     in_flags, 1);
```

4. Set the channel list and scan rate.

```
//Set channel list for each device in the VMap and setup scan rate

DqRtVmapSetChannelList(hd, vmapid, DEVN,
                       DQ_MF101_SS_AI, in_cl, AI_CHANNELS);
DqRtVmapSetScanRate(hd, vmapid, DEVN,
                       DQ_MF101_SS_AI, in_cl, IN_SCANRATE);
```

5. Start the AVMap.

```
//Start the AVMap. Only now the transfer list is transmitted to the IOM

DqRtAXMapStart(hd, vmapid, XMAPMODE, XMAPRATE, XMAPWMRK, 0);
```

5.9.1.2 AVMap Operation

6. Setup to read data out of the VMAP. Note that data is not actually transferred to the IOM until the `DqRtVmapRefresh()` call.

```
// Setup request for data that will occur on next DqRtVmapRefresh call

DqRtAXMapSlotAllocate(hd, TRUE, vmapid, 0);
DqRtVmapRqInputDataSz(hd,
                      vmapid, vmap_in_ch, rq_size, &in_act_size, NULL);

// Update data from the layer

DqRtVmapRefresh(hd, vmapid, 0);
DqRtAXMapEnable(hd, TRUE);
DqCmdTrig(hd);
```

7. *Loop through the remaining steps in AVMap Operation*.

```
//Refresh Inputs
//Note that DqRtAVmapRefreshInputsExt() can return DQ_WAIT_ENDED, indicating
//that no packet was sent from the IOM to the host within the timeout.

DqRtAVmapRefreshInputsExt(hd, vmapid, &pkttype, &counter, &wm_timestamp, NULL)

// Get data from the last DqRtVmapRefresh call

DqRtVmapGetInputData(hd, vmapid, 0, rq_size, &in_data_size,
     &in_avl_size, (uint8*)in_bdata);

// Iterate through each received sample of each scan

scans_rcvd = scans_rcvd + ((in_data_size / (int)sizeof(uint32)) /
                                                num_input_channels);

for (i = 0; i < (in_data_size / (int)sizeof(uint32)); i++) {
    // Extract single sample from buffer,
    //convert data to host endian order
    recv_data = DqNtohl(hd, in_bdata[i]);

    // Check if this is a timestamp
    if (recv_data & DQ_MF101_CLI_TIMESTAMP) {
        timestamp = (double)((recv_data & 0x7fffffff) * (1.0 /
                     ((BUS_FREQUENCY) / (DQ_LN_10us_TIMESTAMP + 1))));
        fprintf(fo, "%.6f\n", timestamp);
    } else {
        // Verify data is from analog input subsystem
        recv_ss = DQ_MF101_CLI_SS(recv_data);
        switch (recv_ss) {
        case DQ_MF101_CLI_SS_AIN:
            // Extract channel and data from sample
            recv_ch = DQ_MF101_CLI_AI_CHAN(recv_data);
            recv_data = DQ_MF101_CLI_AI_DATA(recv_data);
            // Convert to scaled value and write to file
            Chk4Err(DqAdvRawToScaleValue(hd, DEVN,
                            in_cl[recv_ch], recv_data,
                            &in_fdata), goto finish_up);
            fprintf(fo, "%.6f,", in_fdata);
            break;
        default:
            break;
        }
    }
}
```

5.9.1.3 Close Out AVMap

8. Stop and clean up the AVMap with the calls:

```
DqRtAXMapEnable(hd, FALSE);

DqRtVmapStop(hd, vmapid);

DqRtVmapClose(hd, vmapid);
```

Appendix A

Accessories

A.1 MF-101 STP Board and Cable

The DNA-MF-CBL-STP accessory kit is designed to simplify field wiring to the DNx-MF-101. The DNA-MF-CBL-STP kit contains two pieces: the DNA-CBL-MF-1M cable and the DNA-STP-MF-101 screw terminal panel, which may also be ordered separately if desired.

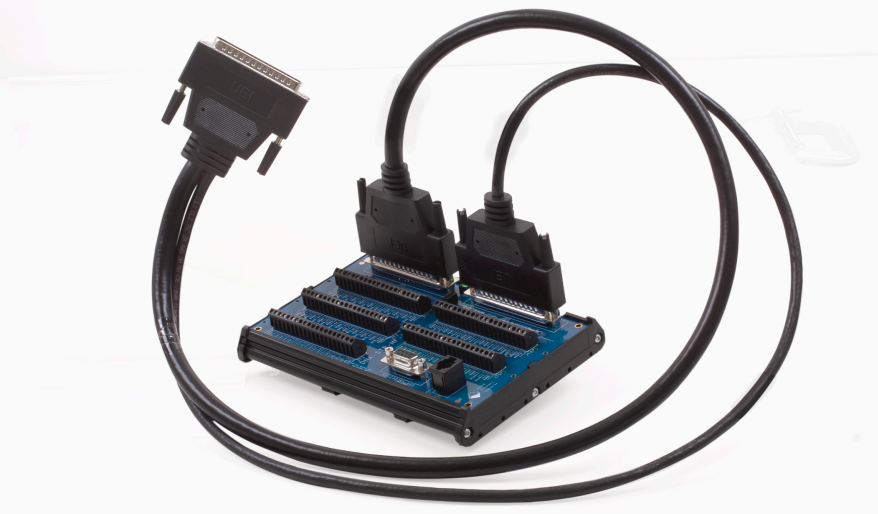


Figure A-1 Photo of DNA-STP-MF-101 screw terminal board with DNA-CBL-MF-1M cable

DNA-CBL-MF-1M

This round, heavy-shielded cable attaches to the DNx-MF-101 using a 62-pin male D-sub connector. The cable splits out analog and digital signals into a 37-pin female D-sub and a 62-pin female D-sub respectively, which plug directly into the DNA-STP-MF-101 screw terminal board. Splitting the analog and digital signals into separate cables ensures good noise performance on the analog signals even when the digital I/O section might be switching high frequencies or currents. The cable utilizes twisted pairs to further reduce noise and crosstalk.

The “1M” cable is 1 meter (3.28 ft) long and weighs 0.46 lbs (0.21 kg).

DNA-STP-MF-101

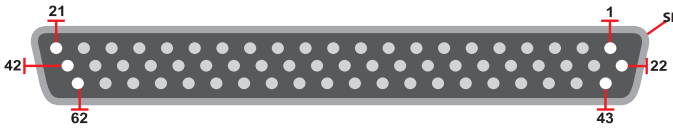
The STP-MF-101 is a screw terminal panel which connects to the DNx-MF-101 using the DNA-CBL-MF-1M splitter cable. The pinout is shown in Figure A-2. Features include:

- All signals brought out to five 20-position terminal blocks
- DB-9 female connector for RS-232/422/485
- RJ-11 jack for I^2C
- Jumper block for connecting TTL inputs to either +5V or DGnd
- On-board ADT7420 temperature sensor for cold junction compensation. The sensor connects to the I2C port using a jumper.
- An on-board LED indicates the presence of the +5V supply (pin 24).

The STP-MF-101 board measures 7 x 4.25 inches (17.8 x 10.8 cm) and weighs 0.30 lbs (0.14 kg).

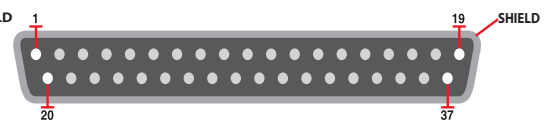
DB-62 (male)
62-pin connector

62	DV 12-15 [32]	42	DGnd [31]	21	DIO-15 [53]
61	DV 12-15 [32]	41	DGnd [31]	20	DIO-14 [10]
60	DIO-11 [51]	40	DGnd [31]	19	DIO-13 [54]
59	DV 8-11 [30]	39	DGnd [31]	18	DIO-12 [11]
58	DIO-09 [52]	38	DGnd [29]	17	DIO-10 [8]
57	DV 8-11 [30]	37	DGnd [29]	16	DIO-08 [9]
56	DV 4-7 [28]	36	DGnd [27]	15	DIO-07 [49]
55	DV 4-7 [28]	35	DGnd [27]	14	DIO-06 [6]
54	DIO-03 [47]	34	DGnd [27]	13	DIO-05 [50]
53	DV 0-3 [26]	33	DGnd [27]	12	DIO-04 [7]
52	DIO-02 [4]	32	DGnd [25]	11	DIO-01 [48]
51	DV 0-3 [26]	31	DGnd [25]	10	DIO-00 [5]
50	Trig Out [45]	30	+5V-TTL [24]	9	+5V-TTL [24]
49	Gnd [34]	29	Gnd [33]	8	TTL 3 [56]
48	Trig In [46]	28	Gnd [33]	7	TTL 2 [13]
47	Gnd [34]	27	Gnd [34]	6	TTL 1 [55]
46	I2C SDA [3]	26	Gnd [34]	5	TTL 0 [12]
45	Gnd [23]	25	Gnd [23]	4	CTS232/RX485- [43]
44	I2C SCL [2]	24	Gnd [23]	3	RX232/RX485+ [44]
43	Gnd [23]	23	Gnd [23]	2	RTS232/TX485+ [1]
		22	Gnd [23]	1	TX232/TX485- [22]



DB-37 (male)
37-pin connector

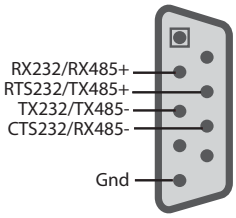
37	AIn 15/7- [21]	19	n/c
36	AIn 14/7+ [42]	18	AGnd [60]
35	AIn 13/6- [20]	17	AGnd [60]
34	AIn 12/6+ [41]	16	AGnd [60]
33	AIn 11/5- [61]	15	AGnd [60]
32	AIn 10/5+ [62]	14	AGnd [60]
31	AIn 9/4- [19]	13	AGnd [60]
30	AIn 8/4+ [40]	12	AGnd [60]
29	AIn 7/3- [18]	11	AGnd [60]
28	AIn 6/3+ [39]	10	AGnd [59]
27	AIn 5/2- [17]	9	AGnd [59]
26	AIn 4/2+ [38]	8	AGnd [59]
25	AIn 3/1- [57]	7	AGnd [59]
24	AIn 2/1+ [58]	6	AGnd [59]
23	AIn 1/0- [16]	5	AGnd [59]
22	AIn 0/0+ [37]	4	AGnd [59]
21	AGnd 1 [14]	3	AGnd [59]
20	A~Gnd 0 [15]	2	AOut 1 [35]
		1	AOut 0 [36]



20-position terminal blocks

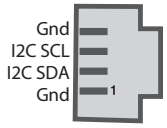
JT1	JT2	JT3	JT4	JT5
DV 12-15 [32]	DV 4-7 [28]	+5V-TTL [24]	AOut 0 [36]	AGnd [60]
DV 12-15 [32]	DV 4-7 [28]	+5V-TTL [24]	AGnd 0 [15]	AIn 8/4+ [40]
DIO-15 [53]	DIO-07 [49]	Gnd [25]	AOut 1 [35]	AIn 9/4- [19]
DGnd [33]	DGnd [29]	Gnd [25]	AGnd 1 [14]	AGnd [60]
DIO-14 [10]	DIO-06 [6]	Trig Out [45]	AGnd [59]	AGnd [60]
DGnd [31]	DGnd [29]	Trig In [46]	AIn 0/0+ [37]	AIn 10/5+[62]
DIO-13 [54]	DIO-05 [50]	TTL 3 [56]	AIn 1/0- [16]	AIn 11/5- [61]
DGnd [33]	DGnd [29]	TTL 2 [13]	AGnd [59]	AGnd [60]
DIO-12 [11]	DIO-04 [7]	TTL 1 [55]	AGnd [59]	AGnd [60]
DGnd [31]	DGnd [27]	TTL 0 [12]	AIn 2/1+ [58]	AIn 12/6+ [41]
DV 8-11 [30]	DV 0-3 [26]	Gnd [23]	AIn 3/1- [57]	AIn 13/6- [20]
DV 8-11 [30]	DV 0-3 [26]	I2C SDA [3]	AGnd [59]	AGnd [60]
DIO-11 [51]	DIO-03 [47]	Gnd [23]	AGnd [59]	AGnd [60]
DGnd [29]	DGnd [27]	I2C SCL [2]	AIn 4/2+ [38]	AIn 14/7+ [42]
DIO-10 [8]	DIO-02 [4]	Gnd [25]	AIn 5/2- [17]	AIn 15/7- [21]
DGnd [29]	DGnd [27]	CTS232/RX485- [43]	AGnd [59]	AGnd [60]
DIO-9 [52]	DIO-01 [48]	RX232/RX485+ [44]	AGnd [59]	AGnd [60]
DGnd [29]	DGnd [25]	RTS232/TX485+ [1]	AIn 6/3+[39]	AGnd [60]
DIO-8 [9]	DIO-00 [5]	TX232/TX485- [22]	AIn 7/3- [18]	AGnd [60]
DGnd [27]	DGnd [25]	Gnd [25]	AGnd [59]	AGnd [60]

DB-9 (female)



*unlabeled pins are n/c

RJ-11 jack



Jumpers

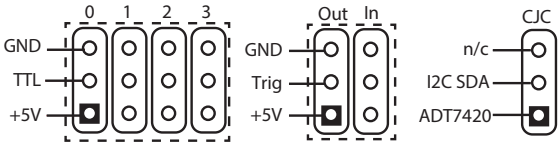


Figure A-2 DNA-STP-MF-101 Pinout

A.2 General Purpose STP Board and Cable

The DNx-MF-101 is also compatible with UEI's general purpose 62-pin cable and screw terminal board. This may be an attractive alternative when space is at a premium and/or your application is not switching high frequency and/or high power digital signals.

DNA-CBL-62

The DNA-CBL-62 is a 62-conductor round shielded cable with 62-pin male D-sub connectors on both ends. It is made with round, heavy-shielded cable; 2.5 ft (75 cm) long, weight of 9.49 ounces or 269 grams; up to 10ft (305cm) and 20ft (610cm).

DNA-STP-62

The STP-62 is a Screw Terminal Panel with three 20-position terminal blocks (JT1, JT2, and JT3) plus one 3-position terminal block (J2). The dimensions of the STP-62 board are 4w x 3.8d x1.2h inch or 10.2 x 9.7 x 3 cm (with standoffs). The weight of the STP-62 board is 3.89 ounces or 110 grams.

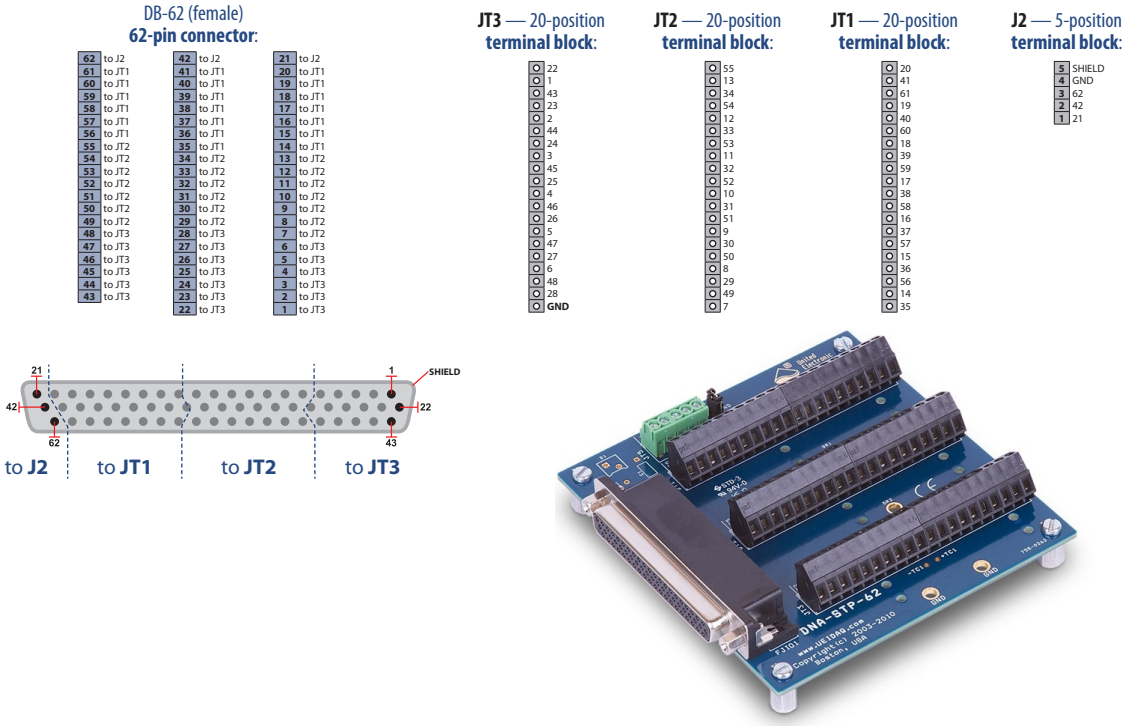


Figure A-3 Pinout and Photo of DNA-STP-62 Screw Terminal Panel

A.3 Test Adapter

The DNx-TADP-101 facilitates testing of DNx-MF-101 hardware and software independent of field wiring. The test adapter plugs into the DB-62 connector on the DNx-MF-101 and internally loops back analog inputs to outputs, industrial digital inputs to outputs, TTL inputs to outputs, and serial receiver to transmitter. A built-in ADT7420 temperature sensor is used to verify I^2C port functionality.