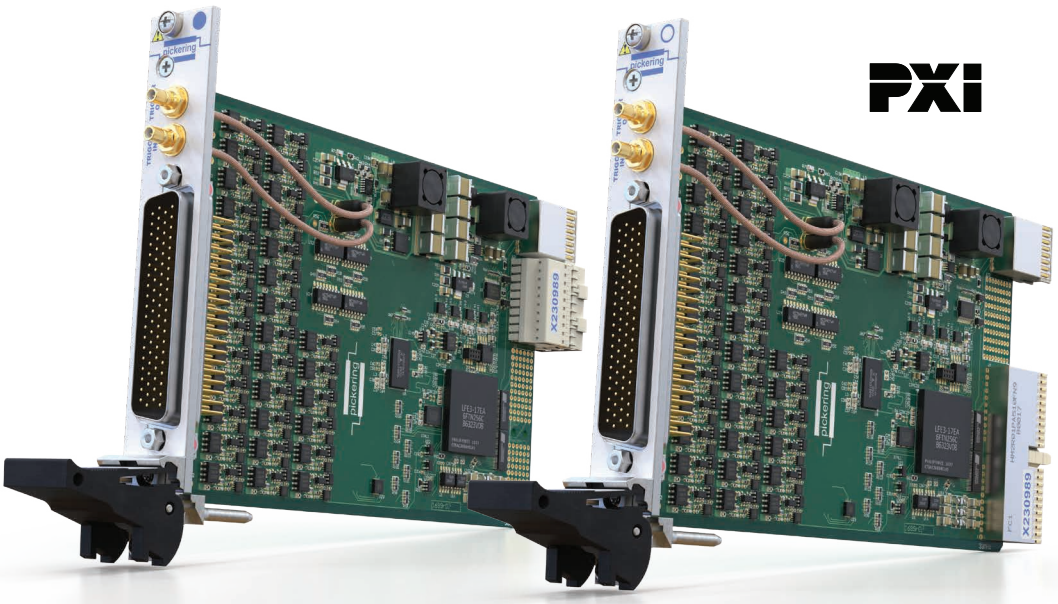
41/43-625 User Manual

PXI/PXIe Multi-Channel Function Generator Module



© Copyright (2026) Pickering Interfaces. All Rights Reserved**.**

No part of this publication may be reproduced, transmitted, transcribed, translated or stored in any form, or by any means without the written permission of Pickering Interfaces.

Technical details contained within this publication are subject to change without notice.



FM38792

Technical Support

For Technical Support please contact Pickering Interfaces either by phone, the website or via e-mail.

Warranty

All products manufactured by Pickering Interfaces are warranted against defective materials and workmanship for a period of three years, excluding programmable power supplies, from the date of delivery to the original purchaser. Any product found to be defective within this period will, at the discretion of Pickering Interfaces be repaired or replaced.

Products serviced and repaired outside of the warranty period are warranted for ninety days.

Extended warranty and service are available. Please contact Pickering Interfaces by phone, the website or via e-mail.

Environmental Policy

Pickering Interfaces operates under an environmental management system similar to ISO 14001.

Pickering Interfaces strives to fulfil all relevant environmental laws and regulations and reduce wastes and releases to the environment. Pickering Interfaces aims to design and operate products in a way that protects the environment and the health and safety of its employees, customers and the public. Pickering Interfaces endeavours to develop and manufacture products that can be produced, distributed, used and recycled, or disposed of, in a safe and environmentally friendly manner.

Worldwide Technical Support and Product Information

Pickering Interfaces Headquarters

Stephenson Road, Clacton-on-Sea, CO15 4NL United Kingdom
Tel: +44 (0)1255-687900
e-mail: sales@pickeringtest.com

USA
Tel: +1 781 897 1710
e-mail: ussales@pickeringtest.com

France
Tel: +33 9 72 58 77 00
e-mail: frsales@pickeringtest.com

Germany
Tel: +49 89 125 953 160
e-mail: desales@pickeringtest.com

Sweden
Tel: +46 340-69 06 69
e-mail: ndsales@pickeringtest.com

Czech Republic
Tel: +420 558 987 613
e-mail: desales@pickeringtest.com

China
Tel: +86 4008 799 765
e-mail: chinasales@pickeringtest.com

Malaysia
Tel: +6012 833 7980
e-mail: aseansales@pickeringtest.com

Product Safety

Safety Symbols

The following safety symbols may be used on the product and throughout the product documentation.

Meaning/Description	Symbol
PROTECTIVE EARTH (GROUND) To identify any terminal which is intended for connection to an external conductor for protection against electric shock in case of a fault, or the terminal of a protective earth (ground) electrode.	
DANGEROUS VOLTAGE To indicate hazards arising from dangerous voltages.	
WARNING/CAUTION An appropriate safety instruction should be followed or caution to a potential hazard exists. Refer to the relevant instructions detailed within the product manual.	
HEAVY If this product is heavy reference should be made to the safety instructions for provisions of lifting and moving.	
STATIC SENSITIVE To indicate that static sensitive devices are present and handling precautions should be followed.	

Contents

Copyright Statement ..... (ii)

Technical Support and Warranty ..... (iii)

Product Safety ..... (iv)

Contents (this page) ..... (v)

Warnings and Cautions ..... (vi)

Section 1

Technical Specification 1.1

Section 2

Technical Description 2.1

- Functional Description 2.1
- Trigger Function 2.2

Section 3

Installation 3.1

- Pre Operation Checks 3.1
- Hardware Installation 3.2
- Software Installation 3.2
- Testing Operation 3.3

Section 4

Programming Guide 4.1

- Programming Options for Pickering PXI/PXIe Modules 4.1
- Programming Architecture 4.2
- APIs 4.8
- General Soft Front Panel (GSFP) 4.64
- Using Pickering Drivers in LabVIEW 4.66

Section 5

Connector Information 5.1

Section 6

Trouble Shooting 6.1

Section 7

Maintenance Information 7.1

- Verification & Adjustment Procedure 7.2

Warnings & Cautions

WARNING - Hazardous Environments	Symbol
This product is not specifically designed for use in hazardous environments, for example in explosive atmospheres. If the product is to be used in hazardous environments we recommend that the user ensures suitable protective measures are taken.	

WARNING - Danger of Electric Shock	Symbol
This module may contain hazardous voltages. Before removing the module from the rack remove all supplies and disconnect user I/O signals. Unused slots in the PXI/PXIe/LXI chassis are populated with blanking plates to prevent access to user I/O signals that may be present. Blanking panels are available to order from Pickering in a variety of slot widths. If the product is not used in this manner for example by using an extender card then additional care must be taken to avoid contact with exposed signals. Not to be used in safety critical circuits, refer to the Pickering Interfaces' terms & conditions of sale.	

CAUTION - Handling of Electrostatic-Sensitive Devices	Symbol
Certain semiconductor devices used in this equipment are liable to damage due to static voltage. Observe the following precautions when handling these devices in their unterminated state, or sub-assemblies containing these devices: • Persons removing sub-assemblies from equipment using these devices must be earthed by a wrist strap and a resistor at the point provided on the equipment. • Soldering irons used during the repair operations must be low voltage types with earthed tips and isolated from the mains voltage by a double insulated transformer. • Outer clothing worn must be unable to generate static charges. • Printed Circuit Boards (PCBs) fitted with these devices must be stored and transported in anti-static bags.	

CAUTION - Product Documentation	Symbol
Suitably qualified & trained users should ensure that the accompanying documentation is fully read and understood before attempting to install or operate the product.	

Safety Instructions

Safety Instructions
All cleaning and servicing requires the equipment to be isolated and disconnected from the power source and user I/O signals (refer to the Maintenance Section). • Appropriate manual handling procedures should be followed as dictated by the weight of the individual module or the combined weight of the modules & chassis. • Should a fault occur with the module or chassis, immediately isolate and disconnect the incoming power to the chassis and the user I/O signals. • Ensure the equipment is installed, operated and maintained by trained and authorised personnel. • For suitably equipped products in the event of an emergency press the red "emergency stop" button situated on the front of the unit.

Section 1 - Technical Specification

- Available as PXI or PXIe Modules
- +/-25 V Output Range with 16-bit Resolution
- DC to 50 kHz, Full Amplitude
- 50 kHz to 300 kHz, Reduced Amplitude
- 8, 16, 24 or 32 Channels in One 3 U Slot
- 5.76 mHz Frequency Resolution
- Simple Generation of Repetitive Arbitrary Waveforms
- DC Offset Capability
- Comprehensive Trigger Support
- PXI Versions Supported by PXI or LXI/USB Chassis
- Kernel and VISA Drivers
- 3 Years Warranty



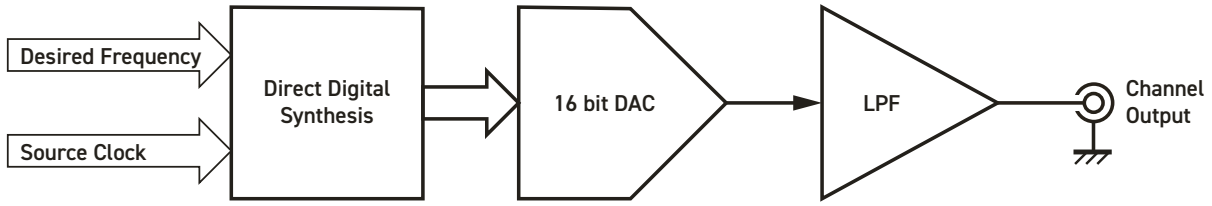
The 41-625 (PXI) and 43-625 (PXIe) are function generators that provide 8, 16, 24 or 32 channels to a single PXI/PXIe slot. It can generate sine waves to 300 kHz with 5.76 mHz frequency resolution referenced to the 10 MHz PXI clock. The output voltage range is up to +/-25 V in a 16-bit resolution. It also supports trigger functions allowing events from other instruments to initiate waveform generation.

The function generator has a very simple method of providing variable output frequencies using Direct Digital Synthesis (DDS). Instantaneous frequency adjustment with continuous phase is done under software control. The output waveform is typically a sine wave but can be programmed to any standard waveform type or customer defined arbitrary.

The DAC is 16 bits with an output dynamic range from -25 V to +25 V with frequencies up to 50 kHz. For frequencies above 50 kHz, amplitude is reduced. The low pass filter (LPF) reduces quantization noise in the output waveform.

Waveforms for the function generator are stored in independent memory blocks, allowing each channel to provide a different signal. Waveforms are easily created externally, using Excel or similar tools, and loaded into the module's memory via the backplane. When channels are set to generate periodic waveforms of the same frequency with continuous trigger, they can be phase aligned via a single command.

The supplied soft front panel allows quick generation of common waveforms and the importing of external waveforms.



DDS (Direct Digital Synthesis) Based Function Generator – Part No. 41/43-625

Section 1 - Technical Specification

Specification - Frequency

Frequency Range:	DC to 50 kHz, full amplitude 50 kHz to 300 kHz, see curve
Frequency Resolution:	5.76 mHz
Phase Coherency:	1° Fout < 1 kHz 10° Fout > 1 kHz
Source Clock Option:	PXI/PCI (33 MHz typical 1 %, PXI version only)* PCIE_REFCLK (100 MHz, PXIe version only)* PXI_CLK10 (10 MHz)* FP_CLK (10 MHz)** On-Board OSC (200 MHz, 50 ppm) †

* Refer to chassis manufacturer's data sheet for clock accuracy specification. Any deviation of the clock source will result in output frequency inaccuracy.

**Only 41/43-625-1XX variants offer the option to input a clock signal via the front panel. Clock accuracy depends on the input.

† Only available in 41/43-625-1XX variants.

Power Requirements - 41-625

+3.3 V	+5 V	+12 V	-12 V
2.24 A	0	0.41 A	0

Power Requirements - 43-625

+3.3 V	+12 V
2.46 A	0.41 A

Specification - Output

Range:	+/- 25 V (0 to 50 kHz) See curve (50 kHz to 300 kHz)
DAC Resolution:	16 bits
Offset Range:	+/-25 V
Accuracy:	+/-0.1 % of Setting +/- 25 mV (DC to 1 kHz, 18 C to 28 C)
Temperature Coefficient:	Offset: 170 µV/C, typical Gain: 0.0033 %/C, typical
Noise:	1 mVrms (DC to 200 kHz, typical)
Output Impedance:	50 Ω
Slew Rate:	3.5 V/µs typical*
Output Current:	10 mA Max
Capacitive Load:	200 pF Max

* Faster ramp rates may cause overshoot or undershoot.

Mechanical Characteristics

41-625 - Single slot 3U PXI (CompactPCI card).
43-625 - Single slot 3U PXIe, compatible with PXIe hybrid slot.

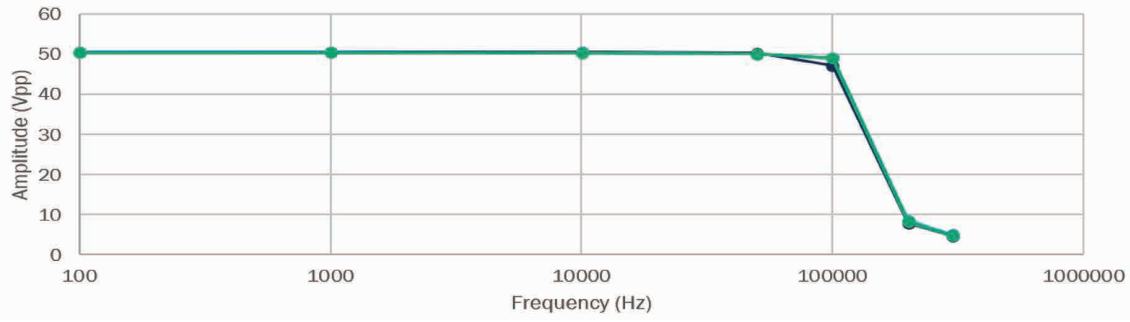
Module weight: <500 g

3D models for all versions in a variety of popular file formats are available on request.

Connectors

41-625 - PXI bus via 32-bit P1/J1 backplane connector.
43-625 - PXIe bus via XJ3 and XJ4 backplane connectors.

Output signals via male 78-pin D-type connector, trigger connections via two SMB coaxial connectors. Please refer to the user manual for pin out information



41/43-625 - Full Scale Amplitude (50 Vpp) vs Frequency for 4 Different Channels

Parametric Performance

Number of Channels:	8, 16, 24 or 32
Spurious Free Dynamic Range (SFDR): (Test Condition: Continuous Sine, 30 Vpp, No DC offset, into 100 kΩ load)	-65 dBc (typical) (Fout <= 1 kHz) -60 dBc (typical) (1 kHz < Fout <= 10 kHz) -30 dBc (typical) (10 kHz < Fout <= 100 kHz) -20 dBc (typical) (100 kHz < Fout <= 300 kHz)
Total Harmonic Distortion (THD): (Test Condition: Continuous Sine, 30 Vpp, No DC offset, into 100 kΩ load)	-70 dBc (typical) (Fout <= 1 kHz) -60 dBc (typical) (1 kHz < Fout <= 10 kHz) -30 dBc (typical) (10 kHz < Fout <= 100 kHz) -20 dBc (typical) (100 kHz < Fout <= 300 kHz)
Waveform Type:	Continuous Sine Complex Waveform (Loaded into Sample Buffer) Pulse (10 µs Minimum) Triangle (Ramp Symmetry = 50 %) Saw (0 % (falling) or 100 % (rising)) DC (+/-25 V)
AC Response Flatness: (Test Condition: Continuous Sine, 20 Vpp, No DC offset, into 10 kΩ load)	+/-0.3 dB (typical) (Fout <= 50 kHz) +/-2 dB (typical) (50 kHz < Fout <= 100 kHz)
DAC Sample Buffer:	2048 Sa/Channel
DAC Conversion Rate:	1.562 MSPS (PXI_CLK10)

PXI & CompactPCI Compliance - 41-625

The module is compliant with the PXI Specification 2.2. Local Bus, Trigger Bus & Star Trigger are not implemented. Uses a 33 MHz 32-bit backplane interface.

PXIe Compliance - 43-625

The module is compliant with the PXIe Specification 1.0. Local Bus, Trigger Bus & Star Trigger are not implemented.

Safety & CE Compliance

All modules are fully CE compliant and meet applicable EU directives:
Low-voltage safety EN61010-1:2010,
EMC Immunity EN61326-1:2013,
Emissions EN55011:2009+A1:2010.

Operating/Storage Conditions

Operating Temperature:	0 °C to +55 °C
Humidity:	Up to 90 % non-condensing
Altitude:	5000 m
Storage/Transport Temperature:	-20 °C to +75 °C
Humidity:	Up to 90 % non-condensing
Altitude:	15000 m

Section 1 - Technical Specification

Product Order Codes

PXI Function Generator Module (Clock Resource: PXI/PCI, PXI_CLK10)	
32 Channel	41-625-001
24 Channel	41-625-002
16 Channel	41-625-003
8 Channel	41-625-004
PXIe Function Generator Module (Clock Resource: PCIE_REFCLK, PXI_CLK10)	
32 Channel	43-625-001
24 Channel	43-625-002
16 Channel	43-625-003
8 Channel	43-625-004
PXI Function Generator Module (Clock Resource: PXI/PCI, PXI_CLK10, On-Board OSC and FP_CLK)	
32 Channel	41-625-101
24 Channel	41-625-102
16 Channel	41-625-103
8 Channel	41-625-104
PXIe Function Generator Module (Clock Resource: PCIE_REFCLK, PXI_CLK10, On-Board OSC and FP_CLK)	
32 Channel	43-625-101
24 Channel	43-625-102
16 Channel	43-625-103
8 Channel	43-625-104
Accessories	
Calibration/Verification Cable	40-975-625-0.3M

Product Customization

Pickering modules are designed and manufactured on our own flexible manufacturing lines, giving complete product control and enabling simple customization to meet very specific requirements.

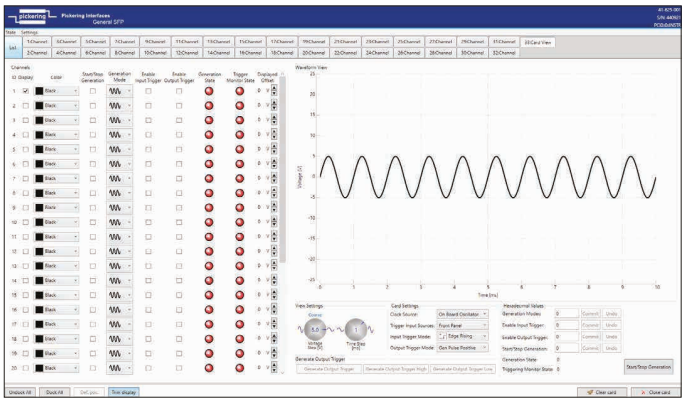
All customized products are given a unique part number, fully documented and may be ordered at any time in the future. Please contact your local sales office to discuss.

Support Products

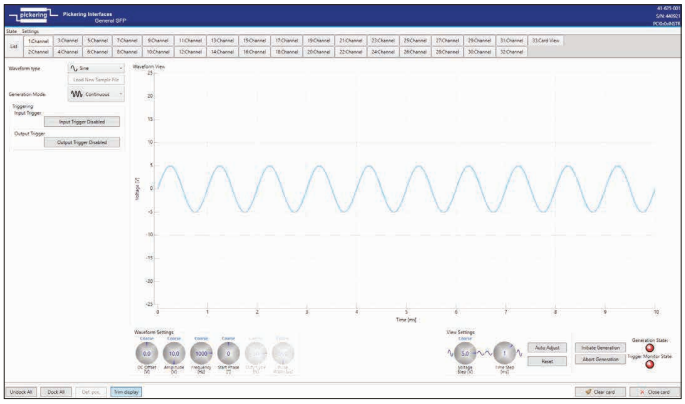
Mating Connectors & Cabling

For connection accessories for the 41/43-625 module please refer to the 90-006D 78-pin D-type Connector data sheet where a complete list and documentation can be found for accessories.

Soft Front Panel



Soft Front Panel for the Function Generator - Card View



Soft Front Panel for the Function Generator - Channel View



43-625 Function Generator Module in PXIe Format



Pickering can supply mating connectors and cable assemblies to enable easy integration of the 41/43-625 module

Section 2 - Technical Description

Functional Description

A functional block diagram is provided in Figure 2.1. The Multi-Channel Function Generator module is power by +3.3V, +5V and +12V (PXI) or +12V and +3.3V (PXIe) from the host chassis backplane. The onboard voltage regulator converts the power from the backplane to supply the operational amplifier in each analog channel. The 10MHz PXI reference clock is sourced from the trigger connector XJ1. Under the control of the PXI clock, the PCI bridge (U1) sends digital patterns to the DAC of each channel for the generation of arbitrary waveforms via SPI interface. The operational amplifiers are for adjusting the amplitude and offset of the waveforms from the DAC and the output of the modulated waveform via the front panel 78-way D-type connector. To deliver an accurate output, each module is pre-calibrated and the memory U3 is for retaining the calibration data. The trigger in and trigger out signal can be accessed through the SMB connectors on the front panel.

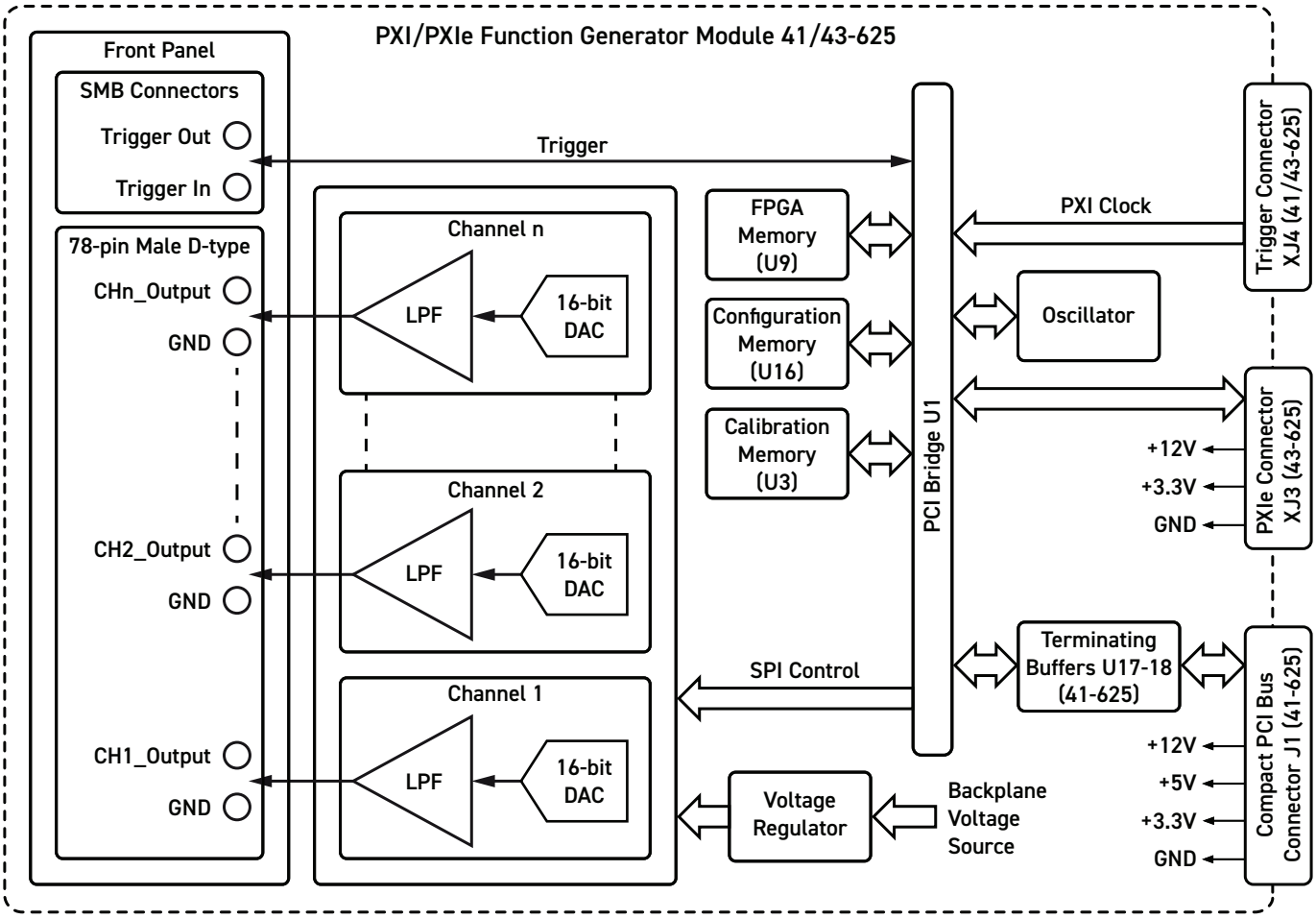


Figure 2.1 - Multi-Channel Function Generator Module 41/43-625 : Functional Block Diagram

Trigger Function

When multiple channels within one card need to have a tight phase relationship, they should all be triggered from the same trigger input such as the software trigger (see section 4 for details), or the front panel SMB Trigger Input. When triggered simultaneously, all channels will start at a phase value (default is phase =0, it also can be configured through *SetStartPhase* API Function) and have the specified phase relationship per the datasheet. All channels within one card will share the same clock source, so channel to channel frequency drift is not an issue. There are two clock source options (see section 4 for clock source selection), PXI_CLK10 or the on-board oscillator.

When multiple channels across multiple cards need to have a tight phase relationship, they should all be triggered from the same trigger input such as the front panel SMB Trigger. One of the cards will generate the master trigger and output the trigger pulse on its front panel SMB Trigger out. All cards in the system, including the master card, should have their Trigger Input SMB connected to the master card's Trigger Output SMB. When the master card issues the trigger, such as from a software trigger, all cards in the system will see the trigger pulse on their front panel SMB at approximately the same time.

To ensure that all channels in the multiple card system do not drift in frequency, they should all share a common clock source. This will need to be the PXI backplane clock, PXI_CLK10.

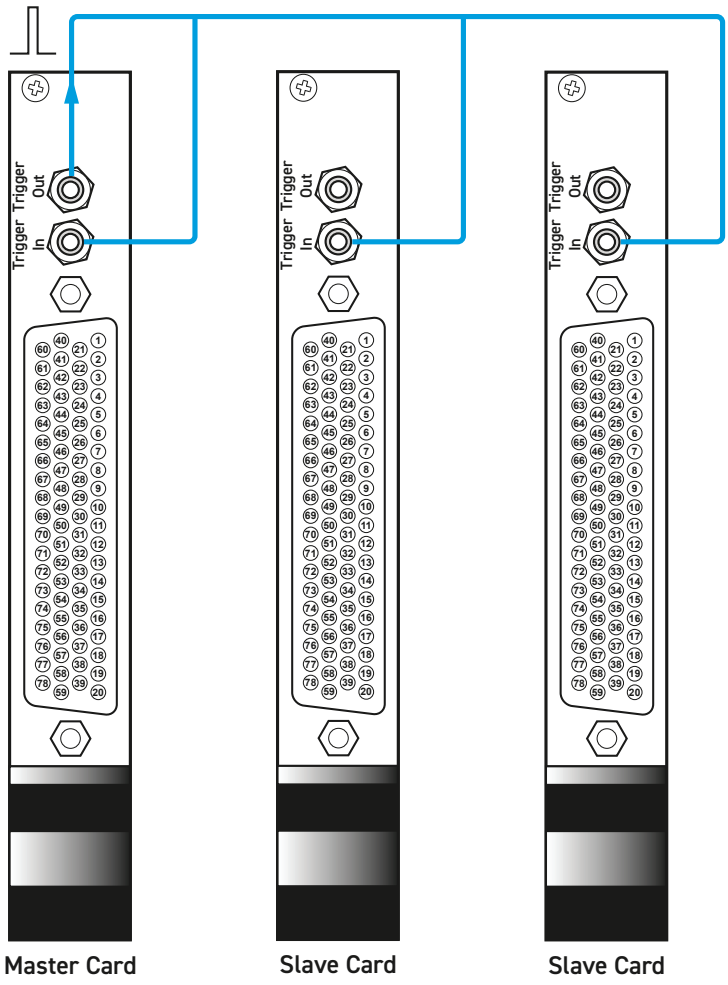


Figure 2.2 An example of Using the Trigger Function Between Multiple Cards

Section 3 - Installation

Refer to the Warnings and Cautions at the front of this manual

Modular products require installation in a suitable PXI/PXIe/LXI chassis. The module is designed for indoor use only.

Pre-operation Checks (Unpacking)

1. Check the module for transport damage and report any damage immediately to Pickering Interfaces. Do not attempt to install the product if any damage is evident.
2. Position the chassis relative to any other equipment the module(s) will connect with. Ensure the chassis is not connected to the electrical supply.
3. Ensure that the designated area for the chassis containing the module is of flat and solid construction to withstand and support the combined weight of the module(s) and chassis.

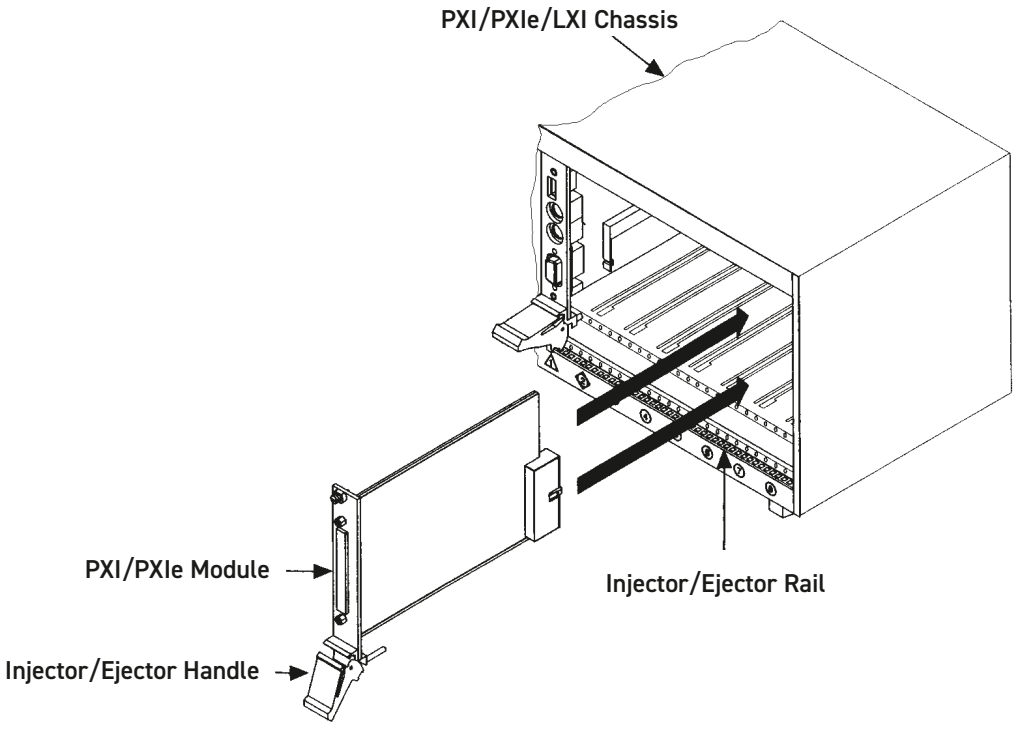


Figure 3.1 - Installing the Module into a PXI/PXIe/cPCI/LXI Chassis

Hardware Installation Electrical Connection (Chassis)

The chassis must be electrically installed in accordance with local regional electrical codes by a qualified engineer.

The chassis must be grounded.

Electrostatic discharge can damage the components on the module. To avoid such damage in handling the board refer to page vi.

Ensure that there is adequate ventilation.

The module should be installed in accordance with the following procedure:

1. Ensure that the system is turned OFF but still connected to mains so that it remains grounded.
2. Choose an appropriate slot in the rack.
3. Remove the blanking plate for the chosen slot.
4. Ensure that the injector/ejector handle is in its downward position. Align the module with the card guides on the top and bottom of the slot.
   Do not raise the injector/ejector handle whilst inserting the module as the module will not insert properly.
5. Hold the handle whilst slowly sliding the module into the card guides until the handle catches on the injector/ejector rail (refer to Figure 3.1).
6. Raise the injector/ejector handle until the module firmly seats into the backplane. The front panel of the module should be flush with the front panel of the chassis.
7. Secure the front panel of the module using both screws to the chassis front panel mounting rails.
8. In a system employing a remote control interface to connect an external controller to a PXI chassis or to link multiple chassis, power-up the system as follows:
   a. For a system comprising an external controller and one chassis, power up the chassis before powering up the external controller.
   b. For a system comprising more than one chassis, turn ON the last chassis in the system followed by the penultimate, etc, and finally turn ON the external controller or chassis containing the system controller.
9. For Pickering Interfaces modular LXI installation there is no requirement to use any particular power up sequence.

Software Installation

First install the appropriate Pickering PXI switch card drivers by running the installer program Setup.exe (provided in a compressed zip file), either from the CD-ROM supplied, or by downloading the latest version from our website: pickeringtest.com - the recommended method. Setup is accompanied by a ReadMe file containing additional installation information. A single installation covers all Pickering switching and simulation PCI, PXI and PXIe products.

When installation completes, the installed drivers' ReadMe file is offered for display. It can also be displayed later using a shortcut on the Programs>>Pickering menu.

If you are not a LabVIEW user you should choose the "full" version, and once that has been installed run the LabVIEW Runtime Engine installer via the shortcut on the Programs>>Pickering menu. In the absence of LabVIEW the Runtime Engine is required to support the Pickering Test Panels application.

Testing Operation

After installation of the hardware and Pickering PXI Driver software package start the General Soft Front Panel (GSFP) using the desktop icon or via the Start menu (***Start Menu-> Pickering Interfaces Ltd -> General Soft Front Panel***).



Figure 3.2 - General Soft Front Panel Icon

A selector panel will appear, listing all installed Pickering PCI, PXI/PXIe or LXI switch cards and resistor cards. Click on the card you wish to control, and a graphical control panel is presented allowing operation of the card. Panels can be opened simultaneously for all the installed cards.

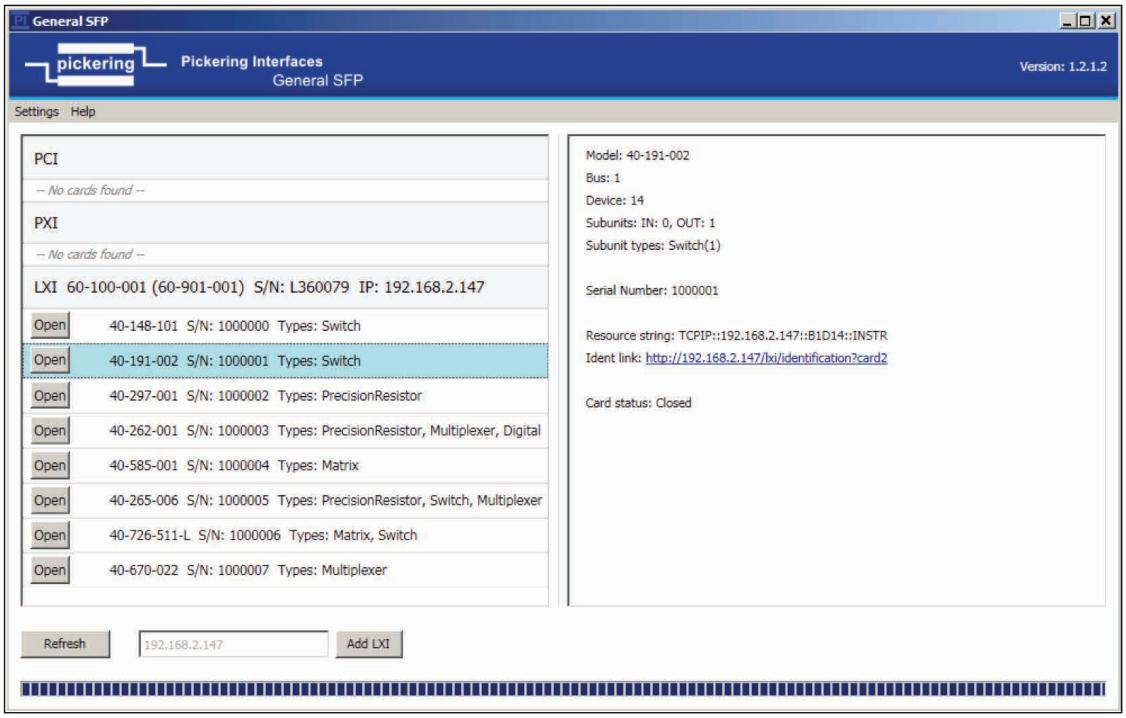


Figure 3.3 - General Soft Front Panel Selector

More details how to use the GSFP can be found in the manual, via menu ***Help\Manual***.

Section 4 - Programming

Programming options for Pickering Interfaces PXI/PXIe Modules

For information on the installation and use of drivers and the programming of Pickering's products in various software environments, please refer to the Software User Manual.

This is available as a download from:
pickeringtest.com/support/software-drivers-and-downloads

Also, it can be accessed from the DVD supplied with the product:
<DVD_DRIVE>\downloads\Documentation\Software-Manual.pdf

Section 4 - Programming

Programming Architecture

Parameters

The Pickering driver utilizes parameter described in the below table.

Parameter	Units	Range	Notes
Amplitude	Volts	See specification notes	Peak-to-Peak voltage value
DC Offset	Volts	See specification notes	Average of the minimum and maximum waveform values
Frequency	Herz	See specification notes	Number of cycles per second
Start Phase	Degrees	<0:360>	Horizontal offset of the waveform
Duty Cycle	Percentage	<0:100>	Percentage of the cycle for which the signal is at its high state (PWM Waveform only)
Pulse Width	Microseconds	<0:1/F_{min}>	Minimum frequency (F_{min}) defined in specification notes

Waveform Types:

- Sine – 50% symmetry,
- Square – 50% symmetry,
- Triangle – 50% symmetry,
- Ramp Up – 100% symmetry,
- Ramp Down – 0% symmetry,
- DC – output assumes DC Offset value,
- Pulse – symmetry defined via pulse width,
- PWM – symmetry defined via duty cycle,
- Arbitrary – user-defined waveform.

Due to the nature of the product, signal parameters may change when changing the waveform type. Therefore, the specific scenarios are handled as follows:

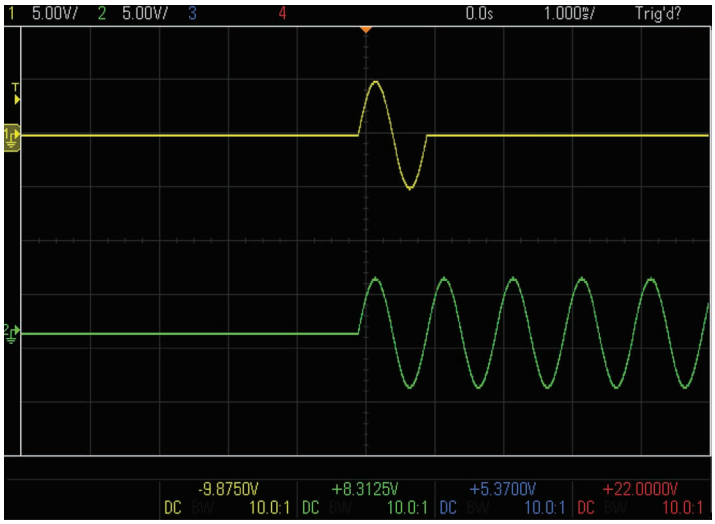
- Changing the waveform type from DC sets the DC Offset to 0.0V,
- Changing the waveform type from Arbitrary sets the Amplitude to 10V and DC Offset to 0V.

It is recommended to reconfigure the channel when changing the waveform type.

Generation Modes and Features

41/43-625 supports the following generation modes:

- One-Shot – the output generates a full cycle of the configured waveform,
- Continuous – the output generates the configured waveform continuously, until abort generation signal is issued.



One-Shot and Continuous Generation modes

Get Generation State feature provides information about the current state of the channel output (0 –> the channel is not generating the output signal, 1 –> the channel is generating the output signal).

Triggering:

The Pickering driver provides a means of synchronization among modules in the form of triggering. Modules can simultaneously generate waveforms when the external signal is received. The feature is configurable via a set of software commands.

Input Trigger:

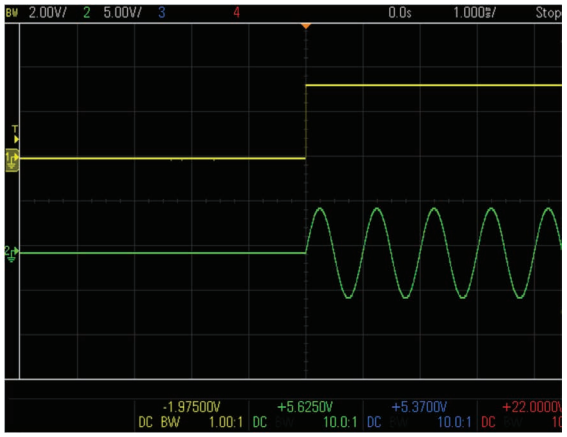
The following input trigger sources are supported:

- Front Panel.

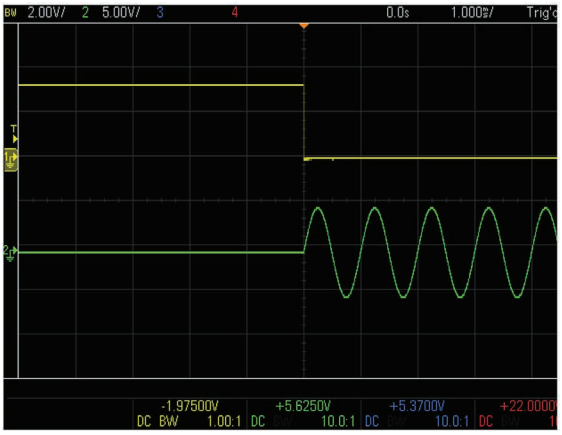
The following input trigger signals are supported:

- Rising Edge,
- Falling Edge,
- Level High,
- Level Low.

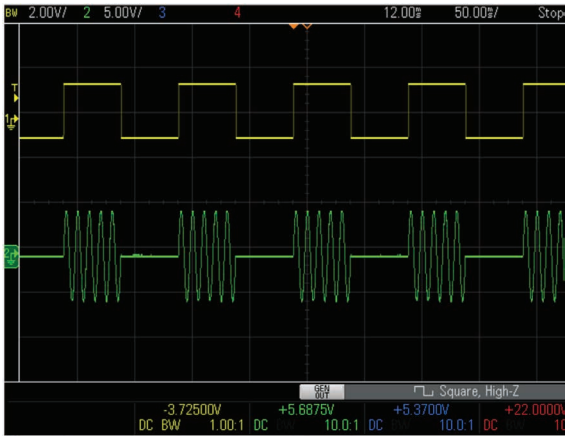
The *Input Trigger Enable* property defines whether the channel responds to the configured input trigger signal. If it is enabled (set to 0x1), the channel starts generation once the trigger is received. Otherwise, the channel can be initiated only via a software command.



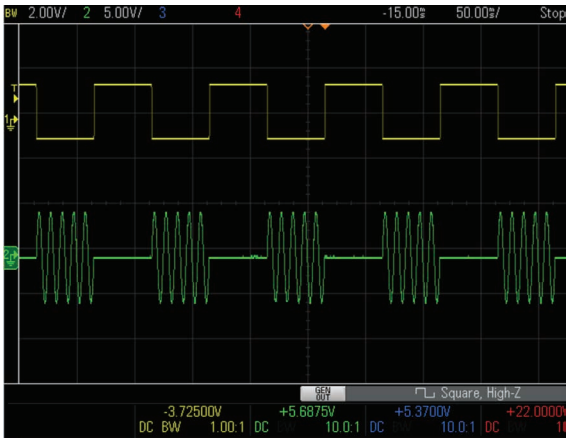
Rising Edge



Falling Edge



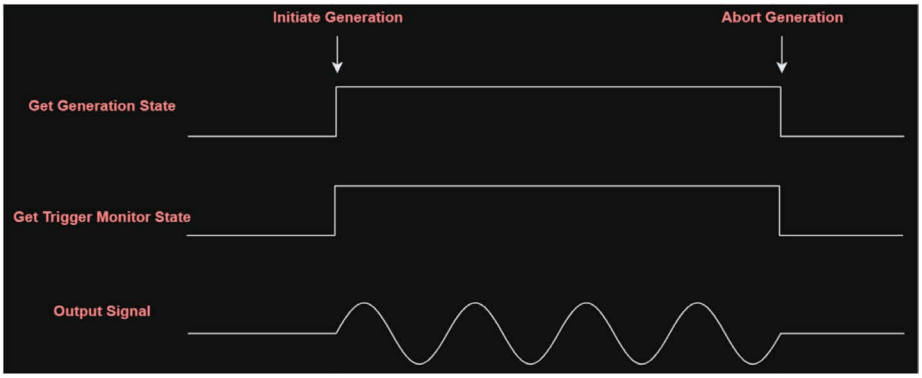
Level High



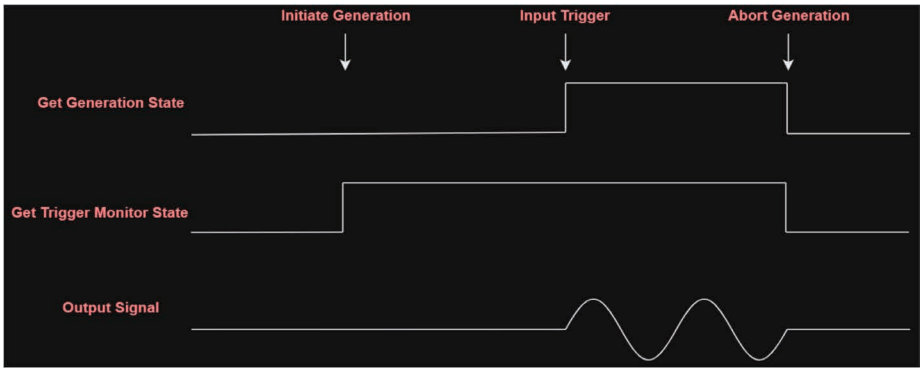
Level Low

Note: *Initiate Generation* command moves channels with *Input Trigger Enable* property enabled into the mode where the incoming input trigger signal triggers generation. Then, the generation starts after the input trigger is received. If the *Input Trigger Enable* property is disabled, the Initiate Generation command will trigger generation straightaway.

Get Trigger Monitor State feature indicates whether the channel is actively listening for the configured trigger signal (0 –> the configured trigger signal will not trigger the generation, 1 –> the configured trigger signal will trigger the generation).



Input Trigger Enable ->0



Input Trigger Enable ->1

Additional Note:

Some operations supported in the APIs require channels in the configuration state i.e. the output signal is not being generated and the generation cannot be triggered. Get Trigger Monitor State indicates whether these operations are obtainable. (0 -> operation can be executed, 1 -> operation cannot be executed, error will be returned).

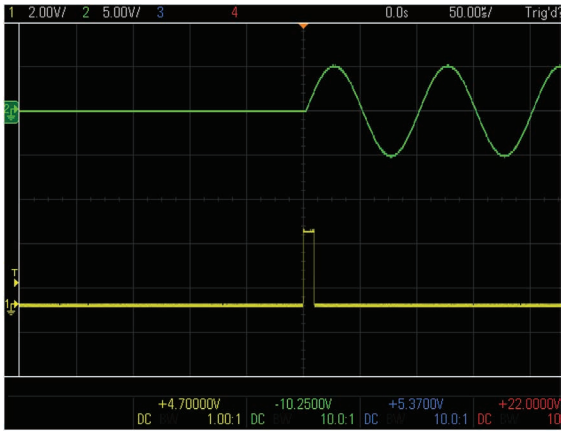
Output Trigger

The following output trigger signals are supported:

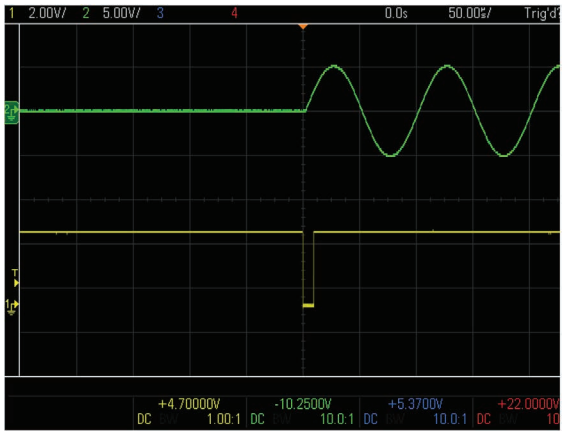
- *Generation Positive Pulse* – TRIG OUT pin generates positive pulse (pin set for 10us) when any of the enabled channels start generation,
- *Generation Negative Pulse* – TRIG OUT pin generates negative pulse (pin reset for 10us) when any of the enabled channels start generation,
- *Software Positive Pulse* – TRIG OUT pin generates positive pulse (pin set for 10us) when the software command is received,
- *Software Negative Pulse* – TRIG OUT pin generates negative pulse (pin reset for 10us) when the software command is received,
- *Generation Level High* – TRIG OUT pin is set when any of the enabled channels is generating a signal,
- *Generation Level Low* – TRIG OUT pin is reset when any of the enabled channels is generating a signal,
- *Software Level* – TRIG OUT pin state defined via the software command.

The *Output Trigger Enable* property defines whether the channel generates the output trigger signal. If it is enabled (set to 0x1), the configured output trigger signal is generated when the channel starts generation. Otherwise, the channel has no impact on the output trigger line.

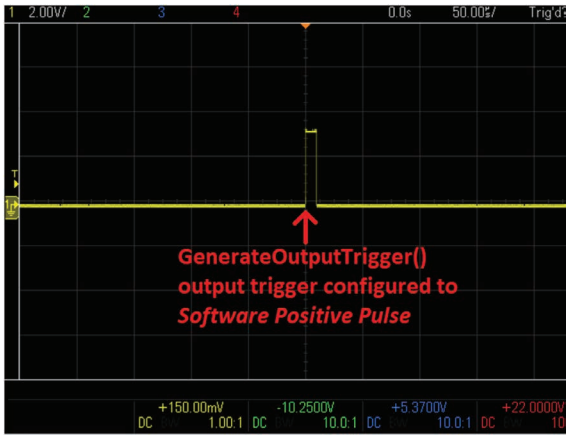
If the trigger output is set to a software mode, the front panel trigger output line is exclusively controlled via software command. The configured output trigger is generated once the software command is received. In this mode, the *Output Trigger Enable* property has no effect on the front panel output line.



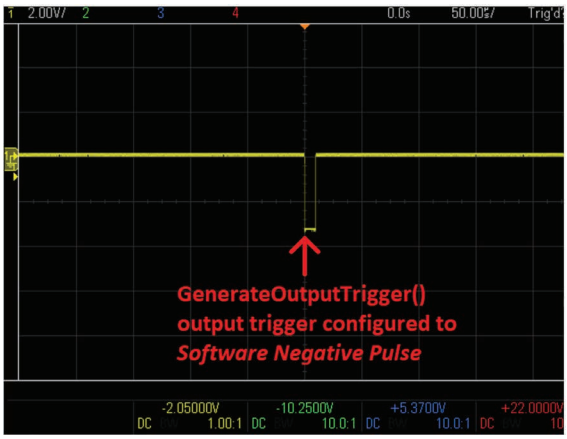
Generation Positive Pulse



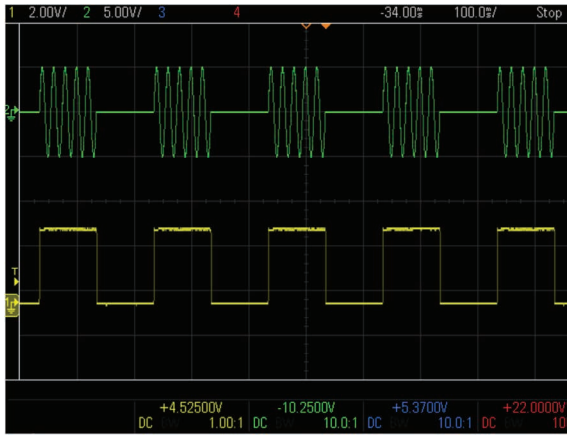
Generation Negative Pulse



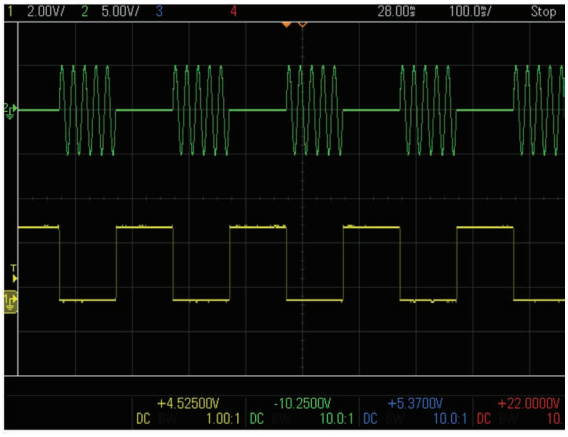
Software Positive Pulse



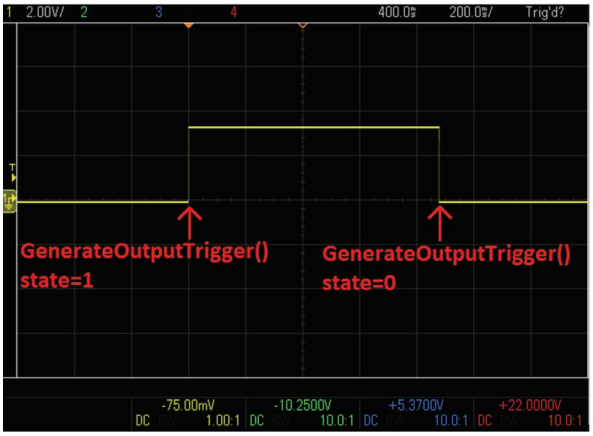
Software Negative Pulse



Generation High Level



Generation Low Level



Software Level (state 1->0)

APIs

The Pickering driver provides typical function generator functionalities in the form of the following APIs:

Pickering Direct I/O driver API for 41/43-625 modules - PILFG:

Parameter	Units
Set Amplitude	PILFG_SetAmplitude
Get Amplitude	PILFG_GetAmplitude
Set DC Offset	PILFG_SetDcOffset
Get DC Offset	PILFG_GetDcOffset
Set Frequency	PILFG_SetFrequency
Get Frequency	PILFG_GetFrequency
Set Start Phase	PILFG_SetStartPhase
Get Start Phase	PILFG_GetStartPhase
Set Duty Cycle	PILFG_SetDutyCycleHigh
Get Duty Cycle	PILFG_GetDutyCycleHigh
Set Pulse Width	PILFG_SetPulseWidth
Get Pulse Width	PILFG_GetPulseWidth
Set Waveform	PILFG_SetWaveform
Get Waveform	PILFG_GetWaveform
Configure Waveform	PILFG_ConfigureWaveform
Set Generation Mode	PILFG_SetGenerationMode
Get Generation Mode	PILFG_GetGenerationMode
Initiate Generation	PILFG_InitiateGeneration
Abort Generation	PILFG_AbortGeneration
Start/Stop Generation	PILFG_StartStopGeneration
Get Generation State	PILFG_GetGenerationState
Create Arbitrary Waveform	PILFG_CreateArbitraryWaveform
Set Input Trigger Config	PILFG_SetInputTriggerConfig
Get Input Trigger Config	PILFG_GetInputTriggerConfig
Set Output Trigger Config	PILFG_SetOutputTriggerConfig
Get Output Trigger Config	PILFG_GetOutputTriggerConfig
Set Input Trigger Enable	PILFG_SetInputTriggerEnable
Get Input Trigger Enable	PILFG_GetInputTriggerEnable
Set Output Trigger Enable	PILFG_SetOutputTriggerEnable
Get Output Trigger Enable	PILFG_GetOutputTriggerEnable
Generate Output Trigger	PILFG_GenerateOutputTrigger
Get Trigger Monitor State	PILFG_GetTriggerMonitorState
Set Clock Source	PILFG_SetClockSource
Get Clock Source	PILFG_GetClockSource
Get Parameter Limits	PILFG_GetParameterLimitsDWORD PILFG_GetParameterLimitsDOUBLE

Supported Waveforms:

```
enum PILFG_WAVEFORM
{
     PILFG_WAVEFORM_SINE             = 0x0,
     PILFG_WAVEFORM_SQUARE           = 0x1,
     PILFG_WAVEFORM_TRIANGLE         = 0x2,
     PILFG_WAVEFORM_RAMP_UP          = 0x3,
     PILFG_WAVEFORM_RAMP_DOWN        = 0x4,
     PILFG_WAVEFORM_DC               = 0x5,
     PILFG_WAVEFORM_PULSE            = 0x6,
     PILFG_WAVEFORM_PWM              = 0x7,
     PILFG_WAVEFORM_ARB              = 0x8
};
```

Supported Generation Modes:

```
enum PILFG_GEN_MODE
{
     PILFG_GEN_CONT         = 0x0,
     PILFG_GEN_ONE_SHOT     = 0x1
};
```

Supported Input Trigger Sources:

```
enum PILFG_TRIG_IN_SRC
{
     PILFG_TRIG_IN_FRONT    = 0x0
};
```

Supported Input Trigger Signals:

```
enum PILFG_TRIG_IN_MODE
{
     PILFG_TRIG_IN_EDGE_RISING       = 0x0,
     PILFG_TRIG_IN_EDGE_FALLING      = 0x1,
     PILFG_TRIG_IN_LEVEL_HIGH        = 0x2,
     PILFG_TRIG_IN_LEVEL_LOW         = 0x3
};
```

Supported Output Trigger Signals:

```
enum PILFG_TRIG_OUT_MODE
{
     PILFG_TRIG_OUT_GEN_PULSE_POS    = 0x0,
     PILFG_TRIG_OUT_GEN_PULSE_NEG    = 0x1,
     PILFG_TRIG_OUT_SOFT_PULSE_POS   = 0x2,
     PILFG_TRIG_OUT_SOFT_PULSE_NEG   = 0x3,
     PILFG_TRIG_OUT_GEN_LEVEL_HIGH   = 0x4,
     PILFG_TRIG_OUT_GEN_LEVEL_LOW    = 0x5,
     PILFG_TRIG_OUT_SOFT_LEVEL       = 0x6
};
```

Supported Clock Sources:

```
enum PILFG_CLK_SRC
{
     PILFG_CLK_ONB_OSC      = 0x0,
     PILFG_CLK_PXI          = 0x1
};
```

DWORD-type Parameters:

```
enum PILFG_PARAM_DWORD
{
     PILFG_PARAM_BUFFER_SIZE    = 0x0,
};
```

DOUBLE-type Parameters:

```
enum PILFG_PARAM_DOUBLE
{
     PILFG_PARAM_AMPLITUDE      = 0x0,
     PILFG_PARAM_VOLTAGE        = 0x1,
     PILFG_PARAM_FREQUENCY      = 0x2,
};
```

Pickering VISA Compliant driver API for 41/43-625 modules - PIPXFG:

Parameter	Units
Set Amplitude	pipxfg_setAmplitude
Get Amplitude	pipxfg_getAmplitude
Set DC Offset	pipxfg_setDcOffset
Get DC Offset	pipxfg_getDcOffset
Set Frequency	pipxfg_setFrequency
Get Frequency	pipxfg_getFrequency
Set Start Phase	pipxfg_setStartPhase
Get Start Phase	pipxfg_getStartPhase
Set Duty Cycle	pipxfg_setDutyCycleHigh
Get Duty Cycle	pipxfg_getDutyCycleHigh
Set Pulse Width	pipxfg_setPulseWidth
Get Pulse Width	pipxfg_getPulseWidth
Set Waveform	pipxfg_setWaveform
Get Waveform	pipxfg_getWaveform
Configure Waveform	pipxfg_configureWaveform
Set Generation Mode	pipxfg_setGenerationMode
Get Generation Mode	pipxfg_getGenerationMode
Initiate Generation	pipxfg_initiateGeneration
Abort Generation	pipxfg_abortGeneration
Start/Stop Generation	pipxfg_startStopGeneration
Get Generation State	pipxfg_getGenerationState
Create Arbitrary Waveform	pipxfg_createArbitraryWaveform
Set Input Trigger Config	pipxfg_setInputTriggerConfig
Get Input Trigger Config	pipxfg_getInputTriggerConfig
Set Output Trigger Config	pipxfg_setOutputTriggerConfig
Get Output Trigger Config	pipxfg_getOutputTriggerConfig
Set Input Trigger Enable	pipxfg_setInputTriggerEnable
Get Input Trigger Enable	pipxfg_getInputTriggerEnable
Set Output Trigger Enable	pipxfg_setOutputTriggerEnable
Get Output Trigger Enable	pipxfg_getOutputTriggerEnable
Generate Output Trigger	pipxfg_generateOutputTrigger
Get Trigger Monitor State	pipxfg_getTriggerMonitorState
Set Clock Source	pipxfg_setClockSource
Get Clock Source	pipxfg_getClockSource
Get Parameter Limits	pipxfg_getParameterLimitsViUInt32 pipxfg_getParameterLimitsViReal64

Supported Waveforms:

```
#define pipxfg_WAVEFORM_SINE          0x0
#define pipxfg_WAVEFORM_SQUARE        0x1
#define pipxfg_WAVEFORM_TRIANGLE      0x2
#define pipxfg_WAVEFORM_RAMP_UP       0x3
#define pipxfg_WAVEFORM_RAMP_DOWN     0x4
#define pipxfg_WAVEFORM_DC            0x5
#define pipxfg_WAVEFORM_PULSE         0x6
#define pipxfg_WAVEFORM_PWM           0x7
#define pipxfg_WAVEFORM_ARB           0x8
```

Supported Generation Modes:

```
#define pipxfg_GEN_CONT               0x0
#define pipxfg_GEN_ONE_SHOT           0x1
```

Supported Input Sources:

```
#define pipxfg_TRIG_IN_FRONT          0x0
```

Supported Input Trigger Signals:

```
#define pipxfg_TRIG_IN_EDGE_RISING        0x0
#define pipxfg_TRIG_IN_EDGE_FALLING       0x1
#define pipxfg_TRIG_IN_LEVEL_HIGH         0x2
#define pipxfg_TRIG_IN_LEVEL_LOW          0x3
#define pipxfg_TRIG_OUT_GEN_LEVEL_HIGH    0x4
#define pipxfg_TRIG_OUT_GEN_LEVEL_LOW     0x5
#define pipxfg_TRIG_OUT_SOFT_LEVEL        0x6
```

Supported Output Trigger Signals:

```
#define pipxfg_TRIG_OUT_GEN_PULSE_POS     0x0
#define pipxfg_TRIG_OUT_GEN_PULSE_NEG     0x1
#define pipxfg_TRIG_OUT_SOFT_PULSE_POS    0x2
#define pipxfg_TRIG_OUT_SOFT_PULSE_NEG    0x3
```

Supported Clock Sources:

```
#define pipxfg_CLK_ONB_OSC            0x0
#define pipxfg_CLK_PXI                0x1
```

ViUInt32-type Parameters:

```
#define pipxfg_PARAM_BUFFER_SIZE      0x0
```

ViReal64-type Parameters:

```
#define pipxfg_PARAM_AMPLITUDE        0x0
#define pipxfg_PARAM_VOLTAGE          0x1
#define pipxfg_PARAM_FREQUENCY        0x2
```

Set Amplitude:

PILFG:

```
/* ============================================================================
* Function:   Set the signal amplitude (peak-to-peak) of the given sub-unit.
* Arguments:  CardNum - Pickering card reference of target (unity-based),
*             SubNum - sub-unit of target to access (unity-based),
*             Amplitude - amplitude to set in Volts.
* Returns:    0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_SetAmplitude(DWORD CardNum, DWORD SubNum, double Amplitude) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Set the signal amplitude (peak-to-peak) of the given sub-unit
//       vi (in)           Instrument handle
//       subUnit (in)      Sub-unit number
//       amplitude (in)    amplitude (peak-to-peak) to set in Volts
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_setAmplitude(ViSession vi,
                                      ViUInt32 subUnit,
                                      ViReal64 amplitude) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Amplitude in Volts
double Amplitude = 10.0;
status = PILFG_SetAmplitude(cardNum, channel, Amplitude);
```

PIPXFG:

```
// Amplitude in Volts
ViReal64 Amplitude = 10.0;
status = pipxfg_setAmplitude(handle, channel, Amplitude);
```

NOTE: If the Amplitude is outside the specified range, the above functions will return the `ER_BAD_AMPLITUDE`/`pipx40_ERROR_AMPLITUDE` error.

NOTE: If the Amplitude to set along with the current DC Offset moves any samples of the signal outside the specified output range, the above functions will return the `ER_BAD_VOLTAGE`/`pipx40_ERROR_BAD_VOLTAGE` error.

NOTE: The feature is unavailable for `PILFG_WAVEFORM_ARB`/`pipxfg_WAVEFORM_ARB`. The above functions will return the `ER_BAD_WAVEFORM`/`pipx40_ERROR_BAD_WAVEFORM` error when attempting to call them for `PILFG_WAVEFORM_ARB`/`pipxfg_WAVEFORM_ARB`.

NOTE: On-the-fly amplitude changes are available for channels in the continuous generation mode only. The above functions will return the `ER_DDS_ON`/`pipx40_ERROR_DDS_ON` error when attempting to call them for channels in the one-shot generation mode.

Get Amplitude:

PILFG:

```
/* ============================================================================
* Function:   Get the signal amplitude (peak-to-peak) of the given sub-unit.
* Arguments:  CardNum - Pickering card reference of target (unity-based),
*             SubNum - sub-unit of target to access (unity-based),
*             Amplitude - pointer to double variable to accept result.
* Returns:    0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetAmplitude(DWORD CardNum, DWORD SubNum, double* Amplitude) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Get the signal amplitude (peak-to-peak) of the given sub-unit
//       vi (in)           Instrument handle
//       subUnit (in)      Sub-unit number
//       amplitude (out)   amplitude (peak-to-peak) in Volts
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_getAmplitude(ViSession vi,
                                      ViUInt32 subUnit,
                                      ViPReal64 amplitude) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Amplitude in Volts
double Amplitude;
status = PILFG_GetAmplitude(cardNum, channel_index, &Amplitude);
```

PIPXFG:

```
// Amplitude in Volts
ViReal64 Amplitude;
status = pipxfg_getAmplitude(handle, channel, &Amplitude);
```

NOTE: The above functions will provide the Max Amplitude stated in the specification as the result, when calling them for `PILFG_WAVEFORM_ARB`/`pipxfg_WAVEFORM_ARB`.

Set DC Offset:

PILFG:

```
/* ============================================================================
* Function:   Set the signal DC offset of the given sub-unit.
* Arguments:  CardNum - Pickering card reference of target (unity-based),
*             SubNum - sub-unit of target to access (unity-based),
*             DcOffset - DC offset to set in Volts.
* Returns:    0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_SetDcOffset(DWORD CardNum, DWORD SubNum, double DcOffset) MYATTR
```

PIPXFG:

```
//*****************************************************************************
// Set the signal DC offset of the given sub-unit
//       vi (in)       Instrument handle
//       subUnit (in)  Sub-unit number
//       dcOffset (in) DC offset to set in Volts
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_setDcOffset(ViSession vi,
                                     ViUInt32 subUnit,
                                     ViReal64 dcOffset) MYATTR;
```

Example Code Snippets:

PILFG:

```
// DC Offset in Volts
double DcOffset = 0.0;
status = PILFG_SetDcOffset(cardNum, channel, DcOffset);
```

PIPXFG:

```
// DC Offset in Volts
ViReal64 DcOffset = 0.0;
status = pipxfg_setDcOffset(handle, channel, DcOffset);
```

NOTE: If the DC Offset is outside the specified range, the above functions will return the `ER_BAD_DC_OFFSET/pipx40_ERROR_DC_OFFSET` error.

NOTE: If the DC Offset to set along with the current Amplitude moves any samples of the signal outside the specified output range, the above functions will return the `ER_BAD_VOLTAGE/pipx40_ERROR_BAD_VOLTAGE` error. This does not apply to waveforms in `PILFG_WAVEFORM_DC/pipxfg_WAVEFORM_DC` mode where a user can set DC Offset within the specified output range regardless of current Amplitude.

NOTE: The feature is unavailable for `PILFG_WAVEFORM_ARB/pipxfg_WAVEFORM_ARB`. The functions will return the `ER_BAD_WAVEFORM/pipx40_ERROR_BAD_WAVEFORM` error when attempting to call them for `PILFG_WAVEFORM_ARB/pipxfg_WAVEFORM_ARB`.

NOTE: On-the-fly DC offset changes are available for channels in the continuous generation mode only. The above functions will return the `ER_DDS_ON/pipx40_ERROR_DDS_ON` error when attempting to call them for channels in the one-shot generation mode.

Get DC Offset:

PILFG:

```
/* ============================================================================
* Function:   Get the signal DC offset of the given sub-unit.
* Arguments:  CardNum - Pickering card reference of target (unity-based),
*             SubNum - sub-unit of target to access (unity-based),
*             DcOffset - pointer to double variable to accept result.
* Returns:    0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetDcOffset(DWORD CardNum, DWORD SubNum, double* DcOffset) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Get the signal DC offset of the given sub-unit
//       vi (in)           Instrument handle
//       subUnit (in)      Sub-unit number
//       dcOffset (out)         DC offset in Volts
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_getDcOffset(ViSession vi,
                                     ViUInt32 subUnit,
                                     ViPReal64 dcOffset) MYATTR;
```

Example Code Snippets:

PILFG:

```
// DC Offset in Volts
double DcOffset;
status = PILFG_GetDcOffset(cardNum, channel, &DcOffset);
```

PIPXFG:

```
// DC Offset in Volts
ViReal64 DcOffset;
status = pipxfg_getDcOffse(handle, channel, &DcOffset);
```

NOTE: The above functions will provide 0.0 VDC as the result, when calling them for `PILFG_WAVEFORM_ARB/pipxfg_WAVEFORM_ARB`.

Set Frequency:

PILFG:

```
/* ============================================================================
* Function:   Set the signal ferquency of the given sub-unit.
* Arguments:  CardNum - Pickering card reference of target (unity-based),
*             SubNum - sub-unit of target to access (unity-based),
*             Frequency - frequency to set in Herz.
* Returns:    0 - success; non-zero - error code,
*/
DWORD _stdcall PILFG_SetFrequency(DWORD CardNum, DWORD SubNum, double Frequency) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Set the signal ferquency of the given sub-unit
//       vi (in)           Instrument handle
//       subUnit (in)      Sub-unit number
//       frequency (in)    frequency to set in Herz
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_setFrequency(ViSession vi,
                                      ViUInt32 subUnit,
                                      ViReal64 frequency) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Frequency in Herz
double Frequency = 1000.0;
status = PILFG_SetFrequency(cardNum, channel, Frequency);
```

PIPXFG:

```
// Frequency in Herz
ViReal64 Frequency = 1000.0;
status = pipxfg_setFrequency(handle, channel, Frequency);
```

NOTE: If the Frequency is outside the specified range, the above functions will return the `ER_BAD_FREQUENCY/pipx40_ERROR_FREQUENCY` error.

NOTE: On-the-fly frequency changes are available for channels in the continuous generation mode only. The above functions will return the `ER_DDS_ON/pipx40_ERROR_DDS_ON` error when attempting to call them for channels in the one-shot generation mode.

Get Frequency:

PILFG:

```
/* ============================================================================
* Function:  Get the signal frequency of the given sub-unit.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            SubNum - sub-unit of target to access (unity-based),
*            Frequency - pointer to double variable to accept result.
* Returns:   0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetFrequency(DWORD CardNum, DWORD SubNum, double* Frequency) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Get the signal frequency of the given sub-unit
//       vi (in)           Instrument handle
//       subUnit (in)      Sub-unit number
//       frequency (out)   frequency in Herz
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_getFrequency(ViSession vi,
                                      ViUInt32 subUnit,
                                      ViPReal64 frequency) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Frequency in Herz
double Frequency;
status = PILFG_GetFrequency(cardNum, channel, &Frequency);
```

PIPXFG:

```
// Frequency in Herz
ViReal64 Frequency;
status = pipxfg_getFrequency(handle, channel, &Frequency);
```

Set Start Phase:

PILFG:

```
/* ============================================================================
* Function:   Set the signal start phase of the given sub-unit.
* Arguments:  CardNum - Pickering card reference of target (unity-based),
*             SubNum - sub-unit of target to access (unity-based),
*             StartPhase - start phase to set in Degrees.
* Returns:    0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_SetStartPhase(DWORD CardNum, DWORD SubNum, double StartPhase) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Set the signal start phase of the given sub-unit
//       vi (in)           Instrument handle
//       subUnit (in)      Sub-unit number
//       startPhase (in)   start phase to set in Degrees
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_setStartPhase(ViSession vi,
                                       ViUInt32 subUnit,
                                       ViReal64 startPhase) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Start Phase in degrees
double StartPhase = 0.0;
status = PILFG_SetStartPhase(cardNum, channel, StartPhase);
```

PIPXFG:

```
// Start Phase in degrees
ViReal64 StartPhase = 0.0;
status = pipxfg_setStartPhase(handle, channel, StartPhase);
```

NOTE: If the Start Phase is outside the specified range, the above functions will return the `ER_BAD_PHASE/pipx40_ERROR_PHASE` error.

NOTE: The feature is available for channels in configuration state only. The function will return the `ER_DDS_ON/pipx40_ERROR_DDS_ON` error if the channel is generating a signal.

Get Start Phase:

PILFG:

```
/* ============================================================================
* Function:  Get the signal start phase of the given sub-unit.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            SubNum - sub-unit of target to access (unity-based),
*            StartPhase - pointer to double variable to accept result.
* Returns:   0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetStartPhase(DWORD CardNum, DWORD SubNum, double* StartPhase) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Get the signal start phase of the given sub-unit
//       vi (in)           Instrument handle
//       subUnit (in)      Sub-unit number
//       startPhase (out)  start phase in Degrees
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_getStartPhase(ViSession vi,
                                       ViUInt32 subUnit,
                                       ViPReal64 startPhase) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Start Phase in degrees
double StartPhase;
status = PILFG_GetStartPhase(cardNum, channel, &StartPhase);
```

PIPXFG:

```
// Start Phase in degrees
ViReal64 StartPhase;
status = pipxfg_getStartPhase(handle, channel, &StartPhase);
```

Set Duty Cycle:

PILFG:

```
/* ============================================================================
* Function:   Set the signal duty cycle (high) of the given sub-unit.
* Arguments:  CardNum - Pickering card reference of target (unity-based),
*             SubNum - sub-unit of target to access (unity-based),
*             DutyCycleHigh - duty cycle (high) to set in Percents.
* Returns:    0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_SetDutyCycleHigh(DWORD CardNum,
                                      DWORD SubNum,
                                      double DutyCycleHigh) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Set the signal duty cycle (high) of the given sub-unit
//       vi (in)               Instrument handle
//       subUnit (in)          Sub-unit number
//       dutyCycleHigh (in)    duty cycle (high) to set in Percents
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_setDutyCycleHigh(ViSession vi,
                                          ViUInt32 subUnit,
                                          ViReal64 dutyCycleHigh) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Duty Cycle (High) in Percents
double DutyCycleHigh = 50.0;
status = PILFG_SetDutyCycleHigh(cardNum, channel, DutyCycleHigh);
```

PIPXFG:

```
// Duty Cycle (High) in Percents
ViReal64 DutyCycleHigh = 50.0;
status = pipxfg_setDutyCycleHigh(handle, channel, DutyCycleHigh);
```

NOTE: If the Duty Cycle is outside the specified range, the above functions will return the `ER_BAD_DUTY_CYCLE/pipx40_ERROR_DUTY_CYCLE` error.

NOTE: The feature is available for `PILFG_WAVEFORM_PWM/pipxfg_WAVEFORM_PWM` only. For other waveforms, the function returns the `ER_BAD_WAVEFORM/pipx40_ERROR_BAD_WAVEFORM` error.

NOTE: On-the-fly duty cycle changes are available for channels in the continuous generation mode only. The above functions will return the `ER_DDS_ON/pipx40_ERROR_DDS_ON` error when attempting to call them for channels in the one-shot generation mode.

Get Duty Cycle:

PILFG:

```
/* ============================================================================
* Function:  Get the signal duty cycle (high) of the given sub-unit.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            SubNum - sub-unit of target to access (unity-based),
*            DutyCycleHigh - pointer to double variable to accept result.
* Returns:   0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetDutyCycleHigh(DWORD CardNum,
                                      DWORD SubNum,
                                      double* DutyCycleHigh) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Get the signal duty cycle (high) of the given sub-unit
//       vi (in)               Instrument handle
//       subUnit (in)          Sub-unit number
//       dutyCycleHigh (out)   duty cycle (high) in Percents
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_getDutyCycleHigh(ViSession vi,
                                          ViUInt32 subUnit,
                                          ViPReal64 dutyCycleHigh) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Duty Cycle (High) in Percents
double DutyCycleHigh;
status = PILFG_GetDutyCycleHigh(cardNum, channel, &DutyCycleHigh);
```

PIPXFG:

```
// Duty Cycle (High) in Percents
ViReal64 DutyCycleHigh;
status = pipxfg_getDutyCycleHigh(handle, channel, &DutyCycleHigh);
```

Set Pulse Width:

PILFG:

```
/* ============================================================================
* Function:   Set the signal pulse width of the given sub-unit.
* Arguments:  CardNum - Pickering card reference of target (unity-based),
*             SubNum - sub-unit of target to access (unity-based),
*             PulseWidth - pulse width to set in microseconds.
* Returns:    0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_SetPulseWidth(DWORD CardNum, DWORD SubNum, double PulseWidth) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Set the signal pulse width of the given sub-unit
//       vi (in)           Instrument handle
//       subUnit (in)      Sub-unit number
//       pulseWidth (in)   pulse width to set in microseconds
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_setPulseWidth(ViSession vi,
                                       ViUInt32 subUnit,
                                       ViReal64 pulseWidth) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Pulse Width in microseconds
double PulseWidth = 10.0;
status = PILFG_SetPulseWidth(cardNum, channel, PulseWidth);
```

PIPXFG:

```
// Pulse Width in microseconds
ViReal64 PulseWidth = 10.0;
status = pipxfg_setPulseWidth(handle, channel, PulseWidth);
```

NOTE: If the Pulse Width is outside the specified range, the above functions will return the `ER_BAD_PULSE_WIDTH/pipx40_ERROR_PULSE_WIDTH` error.

NOTE: The feature is available for `PILFG_WAVEFORM_PULSE/pipxfg_WAVEFORM_PULSE` only. For other waveforms, the function returns the `ER_BAD_WAVEFORM/pipx40_ERROR_BAD_WAVEFORM` error.

NOTE: On-the-fly pulse width changes are available for channels in the continuous generation mode only. The above functions will return the `ER_DDS_ON/pipx40_ERROR_DDS_ON` error when attempting to call them for channels in the one-shot generation mode.

Get Pulse Width:

PILFG:

```
/* ============================================================================
* Function:   Get the signal pulse width of the given sub-unit.
* Arguments:  CardNum - Pickering card reference of target (unity-based),
*             SubNum - sub-unit of target to access (unity-based),
*             PulseWidth - pointer to double variable to accept result.
* Returns:    0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetPulseWidth(DWORD CardNum, DWORD SubNum, double* PulseWidth) MYATTR;
```

PIPXFG:

```
//****************************************************************************
// Get the signal pulse width of the given sub-unit
//       vi (in)           Instrument handle
//       subUnit (in)      Sub-unit number
//       pulseWidth (out)  pulse width in microseconds
//       Return = ViStatus
//****************************************************************************
ViStatus _VI_FUNC pipxfg_getPulseWidth(ViSession vi,
                                       ViUInt32 subUnit,
                                       ViPReal64 pulseWidth) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Pulse Width in microseconds
double PulseWidth;
status = PILFG_GetPulseWidth(cardNum, channel, &PulseWidth);
```

PIPXFG:

```
// Pulse Width in microseconds
ViReal64 PulseWidth;
status = pipxfg_getPulseWidth(handle, channel, &PulseWidth);
```

Set Waveform:

PILFG:

```
/* ============================================================================
* Function:   Set the signal waveform type of the given sub-unit.
* Arguments:  CardNum - Pickering card reference of target (unity-based),
*             SubNum - sub-unit of target to access (unity-based),
*             Waveform - waveform type to set (specified in PILFG_WAVEFORM).
* Returns:    0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_SetWaveform(DWORD CardNum, DWORD SubNum, DWORD Waveform) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Set the signal waveform type of the given sub-unit
//       vi (in)           Instrument handle
//       subUnit (in)      Sub-unit number
//       waveform (in)     waveform type to set
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_setWaveform(ViSession vi,
                                     ViUInt32 subUnit,
                                     ViUInt32 waveform) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Supported Waveform Types:
//   PILFG_WAVEFORM_SINE        = 0x0,
//   PILFG_WAVEFORM_SQUARE      = 0x1,
//   PILFG_WAVEFORM_TRIANGLE    = 0x2,
//   PILFG_WAVEFORM_RAMP_UP     = 0x3,
//   PILFG_WAVEFORM_RAMP_DOWN   = 0x4,
//   PILFG_WAVEFORM_DC          = 0x5,
//   PILFG_WAVEFORM_PULSE       = 0x6,
//   PILFG_WAVEFORM_PWM         = 0x7.
DWORD Waveform = PILFG_WAVEFORM_SINE;
status = PILFG_SetWaveform(cardNum, channel, Waveform);
```

PIPXFG:

```
// Supported Waveform Types:
// 0x0 - > pipxfg_WAVEFORM_SINE
// 0x1 - > pipxfg_WAVEFORM_SQUARE
// 0x2 - > pipxfg_WAVEFORM_TRIANGLE
// 0x3 - > pipxfg_WAVEFORM_RAMP_UP
// 0x4 - > pipxfg_WAVEFORM_RAMP_DOWN
// 0x5 - > pipxfg_WAVEFORM_DC
// 0x6 - > pipxfg_WAVEFORM_PULSE
// 0x7 - > pipxfg_WAVEFORM_PWM.
ViUInt32 Waveform = pipxfg_WAVEFORM_SINE;
status = pipxfg_setWaveform(handle, channel, Waveform);
```

NOTE: If the Waveform is outside the specified range, the above functions will return the ER_WAVEFORM_UNSUPPORTED/ `pipx40_ERROR_WAVEFORM_UNSUPPORTED` error.

NOTE: The feature is unavailable for PILFG_WAVEFORM_ARB/`pipxfg_WAVEFORM_ARB`. The above functions will return the ER_BAD_WAVEFORM/`pipx40_ERROR_BAD_WAVEFORM` error when attempting to call them for PILFG_WAVEFORM_ARB/ `pipxfg_WAVEFORM_ARB`. Arbitrary Waveform configuration is available via Create Arbitrary Waveform feature.

NOTE: The feature is available for channels in configuration state only. The above functions will return the ER_DDS_ON/ `pipx40_ERROR_DDS_ON` error if the channel is generating a signal.

Get Waveform:

PILFG:

```
/* ============================================================================
* Function:  Get the signal waveform type of the given sub-unit.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            SubNum - sub-unit of target to access (unity-based),
*            Waveform - pointer to DWORD variable to accept result.
* Returns:   0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetWaveform(DWORD CardNum, DWORD SubNum, DWORD* Waveform) MYATTR;
```

PIPXFG:

```
//****************************************************************************
// Get the signal waveform type of the given sub-unit
//       vi (in)           Instrument handle
//       subUnit (in)      Sub-unit number
//       waveform (out)    waveform type
//       Return = ViStatus
//****************************************************************************
ViStatus _VI_FUNC pipxfg_getWaveform(ViSession vi,
                                     ViUInt32 subUnit,
                                     ViPUInt32 waveform) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Supported Waveform Types:
//   PILFG_WAVEFORM_SINE        = 0x0,
//   PILFG_WAVEFORM_SQUARE      = 0x1,
//   PILFG_WAVEFORM_TRIANGLE    = 0x2,
//   PILFG_WAVEFORM_RAMP_UP     = 0x3,
//   PILFG_WAVEFORM_RAMP_DOWN   = 0x4,
//   PILFG_WAVEFORM_DC          = 0x5,
//   PILFG_WAVEFORM_PULSE       = 0x6,
//   PILFG_WAVEFORM_PWM         = 0x7,
//   PILFG_WAVEFORM_ARB         = 0x8.
DWORD Waveform;
status = PILFG_GetWaveform(cardNum, channel, &Waveform);
```

PIPXFG:

```
// Supported Waveform Types:
// 0x0 - > pipxfg_WAVEFORM_SINE
// 0x1 - > pipxfg_WAVEFORM_SQUARE
// 0x2 - > pipxfg_WAVEFORM_TRIANGLE
// 0x3 - > pipxfg_WAVEFORM_RAMP_UP
// 0x4 - > pipxfg_WAVEFORM_RAMP_DOWN
// 0x5 - > pipxfg_WAVEFORM_DC
// 0x6 - > pipxfg_WAVEFORM_PULSE
// 0x7 - > pipxfg_WAVEFORM_PWM
// 0x8 - > pipxfg_WAVEFORM_ARB
ViUInt32 Waveform;
status = pipxfg_getWaveform(handle, channel, &Waveform);
```

Configure Waveform:

PILFG:

```
/* ============================================================================
* Function:   Configure the signal parameters of the given sub-unit.
* Arguments:  CardNum - Pickering card reference of target (unity-based),
*             SubNum - sub-unit of target to access (unity-based),
*             Waveform - waveform type to set (specified in PILFG_WAVEFORM),
*             Amplitude - amplitude (peak-to-peak) to set in Volts,
*             DcOffset - DC offset to set in Volts,
*             Frequency - frequency to set in Herz,
*             StartPhase - start phase to set in Degrees,
*             DutyCycleHigh - duty cycle (high) to set in Percents,
*             PulseWidth - pulse width to set in microseconds.
* Returns:    0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_ConfigureWaveform(DWORD CardNum,
                                       DWORD SubNum,
                                       DWORD Waveform,
                                       double Amplitude,
                                       double DcOffset,
                                       double Frequency,
                                       double StartPhase,
                                       double DutyCycleHigh,
                                       double PulseWidth) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Configure the signal parameters of the given sub-unit
//       vi (in)                Instrument handle
//       subUnit (in)           Sub-unit number
//       waveform (out)         waveform to set
//       amplitude (in)         amplitude (peak-to-peak) to set in Volts
//       dcOffset(in)           DC offset to set in Volts
//       frequency (in)         frequency to set in Herz
//       startPhase (in)        start phase to set in Degrees
//       dutyCycleHigh (in)     duty cycle (high) to set in Percents
//       pulseWidth (in)        pulse width to set in microseconds
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_configureWaveform(ViSession vi,
                                           ViUInt32 subUnit,
                                           ViUInt32 waveform,
                                           ViReal64 amplitude,
                                           ViReal64 dcOffset,
                                           ViReal64 frequency,
                                           ViReal64 startPhase,
                                           ViReal64 dutyCycleHigh,
                                           ViReal64 pulseWidth) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Configure parameters in one go
DWORD Waveform = PILFG_WAVEFORM_SINE;
double Amplitude = 10.0;
double DcOffset = 0.0;
double Frequency = 1000.0;
double StartPhase = 0.0;
double DutyCycleHigh = 50.0;
double PulseWidth = 10.0;
status = PILFG_ConfigureWaveform(cardNum,
                                 channel,
                                 Waveform,
                                 Amplitude,
                                 DcOffset,
                                 Frequency,
                                 StartPhase,
                                 DutyCycleHigh,
                                 PulseWidth);
```

PIPXFG:

```
// Configure parameters in one go
ViUInt32 Waveform = pipxfg_WAVEFORM_SINE;
ViReal64 Amplitude = 10.00;
ViReal64 DCOffset = 0.0;
ViReal64 Frequency = 1000.0;
ViReal64 StartPhase = 0.0;
ViReal64 DutyCycleHigh = 50.0;
ViReal64 PulseWidth = 10.0;
status = pipxfg_configureWaveform(handle,
                                  channel,
                                  Waveform,
                                  Amplitude,
                                  DCOffset,
                                  Frequency,
                                  StartPhase,
                                  DutyCycleHigh,
                                  PulseWidth);
```

NOTE: The feature is unavailable for PILFG_WAVEFORM_ARB/pipxfg_WAVEFORM_ARB. The above functions will return the ER_BAD_WAVEFORM/pipx40_ERROR_BAD_WAVEFORM error when attempting to call them for PILFG_WAVEFORM_ARB/pipxfg_WAVEFORM_ARB. Arbitrary Waveform configuration is available via Create Arbitrary Waveform feature. Frequency and Start Phase setup is available via Set Frequency and Set Start Phase features for PILFG_WAVEFORM_ARB/pipxfg_WAVEFORM_ARB.

NOTE: The waveform type update is available for channels in configuration state only. The above functions will return the ER_DDS_ON/pipx40_ERROR_DDS_ON error if the channel is generating a signal and the waveform parameter is different from the actual waveform type.

NOTE: All passed signal parameters are validated with respect to the specification. The above functions will return corresponding error code when respective parameter is outside the range.

NOTE: On-the-fly parameter changes are available for channels in the continuous generation mode only. The above functions will return the ER_DDS_ON/pipx40_ERROR_DDS_ON error when attempting to call them for channels in the one-shot generation mode.

NOTE: If the Amplitude along with the DC Offset moves any samples of the signal outside the specified output range, the above functions will return the ER_BAD_VOLTAGE/pipx40_ERROR_BAD_VOLTAGE error. This does not apply to waveforms in PILFG_WAVEFORM_DC/pipxfg_WAVEFORM_DC mode where a user can set DC Offset within a specified output range regardless of the Amplitude.

NOTE: Start Phase parameter takes effect only when the channel is in the configuration state. Otherwise, the parameter is ignored.

NOTE: Duty Cycle (High) applies to PILFG_WAVEFORM_PWM/pipxfg_WAVEFORM_PWM only. For other waveforms, the parameter is ignored.

NOTE: Pulse Width parameter applies to PILFG_WAVEFORM_PULSE/pipxfg_WAVEFORM_PULSE only. For other waveforms, the parameter is ignored.

Set Generation Mode:

PILFG:

```
/* ============================================================================
* Function:  Set the generation mode of the given sub-unit.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            SubNum - sub-unit of target to access (unity-based),
*            Mode - DWORD-type array defining generation mode to set.
*              (*Mode) = 0 -> sub-unit will be set to continuous mode,
*              (*Mode) = 1 -> sub-unit will be set to one-shot mode.
*
*    If the SubNum parameter is set to 0, the function determines the generation mode of all
* sub-units. Then, each bit of the array defines the generation mode of the corresponding sub-unit.
*        Sub-units are mapped as follows:
*             LSB -> sub-unit 1, MSB (for 32 channel card) -> sub-unit 32.
*        The generation mode is encoded as follows:
*             Bit = 0 -> sub-unit will be set to continuous mode,
*             Bit = 1 -> sub-unit will be set to one-shot mode,
*
*        Size - size of the array.
* Returns:        0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_SetGenerationMode(DWORD CardNum,
                                       DWORD SubNum,
                                       DWORD* Mode,
                                       DWORD Size) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Set the generation mode of the given sub-unit.
//       vi (in)  Instrument handle
//       subUnit (in)  Sub-unit number
//       mode (in)     ViUInt32-type array defining generation mode.
//                     (*mode) = 0 -> sub-unit will be set to continuous mode,
//                     (*mode) = 1 -> sub-unit will be set to one-shot mode.
//
//       If the SubNum parameter is set to 0, the function determines the generation
//       mode of all sub-units. Then, each bit of the array defines the generation
//       mode of the corresponding sub-unit.
//       Sub-units are mapped as follows:
//            LSB->sub - unit 1, MSB (for 32 channel card) -> sub-unit 32.
//       The generation mode is encoded as follows:
//            Bit = 0->sub - unit will be set to continuous mode,
//            Bit = 1->sub - unit will be set to one-shot mode,
//
//       size (in)         size of the array.
//       Return = ViStatus
//*****************************************************************************
```

```
ViStatus _VI_FUNC pipxfg_setGenerationMode(ViSession vi,
                                           ViUInt32 subUnit,
                                           ViPUInt32 mode,
                                           ViUInt32 size) MYATTR;
```

Example Code Snippets:

PILFG:

```
//Generation Modes:
//   0x0 -> PILFG_GEN_CONT -> Continuous Mode
//   0x1 -> PILFG_GEN_ONE_SHOT  -> One-Shot Mode
DWORD generationMode = 0x1;
DWORD size = 0x1;
status = PILFG_SetGenerationMode(cardNum, channel, &generationMode, size);
```

PIPXFG:

```
//Generation Modes:
//   0x0 -> pipxfg_GEN_CONT-> Continuous Mode
//   0x1 -> pipxfg_GEN_ONE_SHOT -> One-Shot Mode
ViUInt32 generationMode = 0x1;
ViUInt32 size = 0x1;
status = pipxfg_setGenerationMode(handle, channel, &generationMode, size);
```

NOTE: The feature is available for channels in configuration state only. The above functions will return the `ER_DDS_ON/ pipx40_ERROR_DDS_ON` error if the channel is generating a signal.

NOTE: If the sub-unit parameter is set to 0 and the size of the array does not correspond the number of channels, the above functions return `ER_BAD_ATTR_VALUE/pipx40_ERROR_BAD_ATTR_VALUE` error.

Get Generation Mode:

PILFG:

```
/* ============================================================================
* Function:  Get the generation mode of the given sub-unit.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            SubNum - sub-unit of target to access (unity-based),
*            Mode - DWORD-type array storing generation mode.
*               (*Mode) = 0 -> sub-unit is in the continuous mode,
*               (*Mode) = 1 -> sub-unit is in the one-shot mode.
*
*        If the SubNum parameter is set to 0, the function provides the generation mode
*        of all sub-units. Then, each bit of the array stores the generation mode of the
*        corresponding sub-unit.
*        Sub-units are mapped as follows:
*            LSB -> sub-unit 1, MSB (for 32 channel card) -> sub-unit 32.
*        The generation mode is decoded as follows:
*            Bit = 0 -> sub-unit is in the continuous mode,
*            Bit = 1 -> sub-unit is in the one-shot mode,
*
*        Size - size of the array.
* Returns:       0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetGenerationMode(DWORD CardNum,
                                       DWORD SubNum,
                                       DWORD* Mode,
                                       DWORD Size) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Get the generation mode of the given sub-unit.
//       vi (in)           Instrument handle
//       subUnit (in)  Sub-unit number
//       mode (out)    ViUInt32-type array storing generation mode.
//                     (*mode) = 0 -> sub-unit is in the continuous mode,
//                     (*mode) = 1 -> sub-unit is in the one-shot mode.
//
//       If the SubNum parameter is set to 0, the function provides the generation mode
//       of all sub-units. Then, each bit of the array stores the generation mode of the
//       corresponding sub-unit.
//       Sub-units are mapped as follows:
//            LSB -> sub-unit 1, MSB (for 32 channel card) -> sub-unit 32.
//       The generation mode is decoded as follows:
//            Bit = 0 -> sub-unit is in the continuous mode,
//            Bit = 1 -> sub-unit is in the one-shot mode,
//
//       size (in)         size of the array.
//       Return = ViStatus
```

```
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_getGenerationMode(ViSession vi,
                                           ViUInt32 subUnit,
                                           ViPUInt32 mode,
                                           ViUInt32 size) MYATTR;
```

Example Code Snippets:

PILFG:

```
//Generation Modes:
//   0x0 -> PILFG_GEN_CONT  -> Continuous Mode
//   0x1 -> PILFG_GEN_ONE_SHOT  -> One-Shot Mode
DWORD generationMode;
DWORD size = 0x1;
status = PILFG_GetGenerationMode(cardNum, channel, &generationMode, size);
```

PIPXFG:

```
//Generation Modes:
//   0x0 -> pipxfg_GEN_CONT-> Continuous Mode
//   0x1 -> pipxfg_GEN_ONE_SHOT -> One-Shot Mode
ViUInt32 generationMode;
ViUInt32 size = 0x1;
status = pipxfg_getGenerationMode(handle, channel, &generationMode, size);
```

NOTE: If the sub-unit parameter is set to 0 and the size of the array does not correspond the number of channels, the above functions return `ER_BAD_ATTR_VALUE/pipx40_ERROR_BAD_ATTR_VALUE` error.

Initiate Generation:

PILFG:

```
/* ============================================================================
* Function:   Initiate the signal generation of the given sub-unit.
* Arguments:  CardNum - Pickering card reference of target (unity-based),
*             SubNum - sub-unit of target to access (unity-based).
* Returns:    0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_InitiateGeneration(DWORD CardNum, DWORD SubNum) MYATTR;
```

PIPXFG:

```
//****************************************************************************
// Initiate the signal generation of the given sub-unit
//       vi (in)       Instrument handle
//       subUnit (in)  Sub-unit number
//       Return = ViStatus
//****************************************************************************
ViStatus _VI_FUNC pipxfg_initiateGeneration(ViSession vi,
                                            ViUInt32 subUnit) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Initiate Generation
status = PILFG_InitiateGeneration(cardNum, channel);
```

PIPXFG:

```
// Initiate Generation
status = pipxfg_initiateGeneration(handle, channel);
```

NOTE: If the channel is in the `PILFG_WAVEFORM_PULSE`/`pipxfg_WAVEFORM_PULSE` mode, the above functions will trigger a single pulse of configured width.

NOTE: If the channel is in the one-shot mode mode and the signal is being generated, the above functions will return `ER_DDS_ON`/`pipxfg_ERROR_DDS_ON` error. For other waveform types the functions have no effect when the channel is generating a signal.

Abort Generation:

PILFG:

```
/* ============================================================================
* Function:  Abort the signal generation of the given sub-unit.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            SubNum - sub-unit of target to access (unity-based),
* Returns:   0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_AbortGeneration(DWORD CardNum, DWORD SubNum) MYATTR;
```

PIPXFG:

```
//****************************************************************************
// Abort the signal generation of the given sub-unit
//       vi (in)           Instrument handle
//       subUnit (in)      Sub-unit number
//       Return = ViStatus
//****************************************************************************
ViStatus _VI_FUNC pipxfg_abortGeneration(ViSession vi,
                                         ViUInt32 subUnit) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Abort Generation
status = PILFG_AbortGeneration(cardNum, channel);
```

PIPXFG:

```
// Abort Generation
status = pipxfg_abortGeneration(handle, channel);
```

Start/Stop Generation:

PILFG:

```
/* ============================================================================
* Function:   Initiate/Abort signal generation of multiple sub-units simultaneously.
* Arguments:  CardNum - Pickering card reference of target (unity-based),
*             SubNum - sub-unit of target to access (unity-based),
*             State - DWORD-type array containing start/stop pattern.
*             Each bit of the array holds action to undertake for the corresponding sub-unit.
*             Sub-units are mapped as follows:
*                  LSB -> sub-unit 1, MSB (for 32 channel card) -> sub-unit 32,
*             Action to undertake is encoded as follows:
*                  Bit = 0 -> the signal generation of the corresponding sub-unit
*                       will be aborted,
*                  Bit = 1 -> the signal generation of the corresponding sub-unit
*                       will be initiated.
*             Size - size of the array containing start/stop pattern.
* Returns:    0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_StartStopGeneration(DWORD CardNum, DWORD* State, DWORD Size) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Initiate/Abort signal generation of multiple sub-units simultaneously
//       vi (in)       Instrument handle
//       state (in)    ViUInt32-type array containing start/stop pattern.
//       Each bit of the array holds action to undertake for the corresponding sub-unit.
//       Sub-units are mapped as follows:
//            LSB->sub - unit 1, MSB (for 32 channel card)->sub - unit 32
//       Action to undertake is encoded as follows:
//            Bit = 0-> the signal generation of the corresponding
//                      sub-unit will be aborted
//            Bit = 1-> the signal generation of the corresponding
//                      sub-unit will be initiated
//       size (in)     size of the array containing start/stop pattern
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_startStopGeneration(ViSession vi,
                                             ViPUInt32 state,
                                             ViUInt32 size) MYATTR;
```

Example Code Snippets:

PILFG:

```
//Example: 0x80400801 initiates channels 1, 12, 23, and 32
DWORD state = 0x80400801;
DWORD size = 0x1;
status = PILFG_StartStopGeneration(cardNum, channel, &state, size);
```

PIPXFG:

```
//Example: 0x80400801 initiates channels 1, 12, 23, and 32
ViUInt32 state = 0x80400801;
ViUInt32 size = 0x1;
status = pipxfg_startStopGeneration(handle, channel, &state, size);
```

NOTE: If the size of the array does not correspond the number of channels, the above functions return `ER_BAD_ATTR_VALUE`/`pipx40_ERROR_BAD_ATTR_VALUE` error.

NOTE: If the channel to initiate generation is in the one-shot mode and the signal is being generated, the above functions will return `ER_DDS_ON`/`pipxfg_ERROR_DDS_ON` error. For other waveform types the functions have no effect when the channel is generating a signal.

Get Generation State:

PILFG:

```
/* ============================================================================
* Function:  Get the signal generation state of the given sub-unit.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            SubNum - sub-unit of target to access (unity-based),
*            State - DWORD-type array to store generation state,
*                 (*State) = 0 -> sub-unit is in the Configuration State
*                                 (the output signal is not generated),
*                 (*State) = 1 -> sub-unit is in the Output Generation State
*                                 (the output signal is generated).
*            Size - size of the array to store generation state.
*
*    If the function is called with SubNum parameter set to 0, it provides the state
*    of all channels to the memory location specified by the State and the Size parameters.
*    Then, each bit of the array stores the generation state of the corresponding sub-unit.
*        Sub-units are mapped as follows:
*            LSB -> sub-unit 1, MSB (for 32 channel card) -> sub-unit 32.
*        Generation State is decoded as follows:
*            Bit = 0 -> sub-unit is in the Configuration State
*                 (the output signal is not being generated),
*            Bit = 1 -> sub-unit is in the Output Generation State
*                 (the output signal is being generated).
* Returns:   0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetGenerationState(DWORD CardNum,
                                        DWORD SubNum,
                                        DWORD* State,
                                        DWORD Size) MYATTR;
```

PIPXFG:

```
//***************************************************************************
// Get the signal generation state of the given sub-unit
//       vi (in)  Instrument handle
//       subUnit (in)  Sub-unit number
//       state (out)   ViUInt32-type array to store generation state
//            (*state) = 0 -> sub-unit in the Configuration State
//                     (the output signal is not generated)
//            (*state) = 1 -> sub-unit in the Output Generation State
//                     (the output signal is generated)
//       size (in)          size of the array to store generation state
//
//   If the function is called with subUnit parameter set to 0, it provides the state
//   of all channels to the memory location specified by the State and the Size parameters.
//   Then, each bit of the array stores the generation state of the corresponding sub-unit.
//       Sub-units are mapped as follows:
//            LSB -> sub - unit 1, MSB (for 32 channel card)-> sub-unit 32.
```

```
//       Generation State is decoded as follows:
//            Bit = 0 -> sub-unit is in the Configuration State
//                       (the output signal is not being generated),
//            Bit = 1 -> sub-unit is in the Output Generation State
//                       (the output signal is being generated).
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_getGenerationState(ViSession vi,
                                            ViUInt32 subUnit,
                                            ViPUInt32 state,
                                            ViUInt32 size) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Get Generation State:
//   0x0 -> channels is not generating the configured waveform
//   0x1 -> channels is generating the configured waveform
DWORD state;
DWORD size = 0x1;
status = PILFG_GetGenerationState(cardNum, channel, &state, size);
```

PIPXFG:

```
// Get Generation State:
//   0x0 -> channels is not generating the configured waveform
//   0x1 -> channels is generating the configured waveform
ViUInt32 state;
ViUInt32 size = 0x1;
status = pipxfg_getGenerationState(handle, channel, &state, size);
```

NOTE: If the sub-unit parameter is set to 0 and the size of the array does not correspond the number of channels, the above functions return `ER_BAD_ATTR_VALUE`/`pipx40_ERROR_BAD_ATTR_VALUE` error.

Create Arbitrary Waveform:

PILFG:

```
/* ============================================================================
* Function:  Create the arbitrary waveform of the given sub-unit.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            SubNum - sub-unit of target to access (unity-based).
*            SampleSource - pointer to character string specifying path of the file
*                containing arbitrary waveform samples.
* Returns:   0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_CreateArbitraryWaveform(DWORD CardNum,
                                             DWORD SubNum,
                                             CHAR* SampleSource) MYATTR;
```

PIPXFG:

```
//============================================================================
// Create the arbitrary waveform of the given sub-unit
//       vi (in)           Instrument handle
//       subUnit (in)      Sub-unit number
//       SampleSource (in) character string specifying path of the file containing
//                         arbitrary waveform samples
//       Return = ViStatus
//============================================================================
ViStatus _VI_FUNC pipxfg_createArbitraryWaveform(ViSession vi,
                                                 ViUInt32 subUnit,
                                                 ViPChar SampleSource) MYATTR;
```

Example Code Snippets:

PILFG:

```
//ARB sample file location
CHAR* filePath = "MyARB.txt";
status = PILFG_CreateArbitraryWaveform(cardNum, channel, filePath);
```

PIPXFG:

```
//ARB sample file location
ViPChar filePath = "MyARB.txt";
status = pipxfg_createArbitraryWaveform(handle, channel, filePath);
```

NOTE: If the sample source file is unavailable, the above functions will return the `ER_FILE_UNAVAILABLE`/`pipxfg_ERROR_FILE_UNAVAILABLE` error.

NOTE: If the content of the sample source file is invalid, the above functions will return the `ER_BAD_FILE`/`pipxfg_ERROR_FILE` error.

Set Input Trigger Config:

PILFG:

```
/* ============================================================================
* Function:  Configure the Input Trigger of the given card.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            Source - source of trigger to set (specified in PILFG_TRIG_IN_SRC)
*            Trigger - input trigger type to set (specified in PILFG_TRIG_IN_MODE)
* Returns:   0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_SetInputTriggerConfig(DWORD CardNum,
                                           DWORD Source,
                                           DWORD Trigger) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Configure the Input Trigger of the given card
//       vi (in)           Instrument handle
//       source (in)       source of trigger to set
//       trigger (in)      input trigger type to set
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_setInputTriggerConfig(ViSession vi,
                                               ViUInt32 source,
                                               ViUInt32 trigger) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Source:
//   0x0 -> PILFG_TRIG_IN_FRONT -> Front Panel
// Trigger Input Modes:
//   0x0  -> PILFG_TRIG_IN_EDGE_RISING   -> Rising Edge
//   0x1  -> PILFG_TRIG_IN_EDGE_FALLING  -> Falling Edge
//   0x2  -> PILFG_TRIG_IN_LEVEL_HIGH    -> Generation on High Level of TRIG IN
//   0x3  -> PILFG_TRIG_IN_LEVEL_LOW     -> Generation on Low Level of TRIG IN
DWORD in_trigger_source = PILFG_TRIG_IN_FRONT;
DWORD in_trigger_type = PILFG_TRIG_IN_EDGE_RISING;
status = PILFG_SetInputTriggerConfig(cardNum, in_trigger_source, in_trigger_type);
```

PIPXFG:

```
// Source:
//   0x0 -> pipxfg_TRIG_IN_FRONT -> Front Panel
//   Trigger Input Modes:
//   0x0  -> pipxfg_TRIG_IN_EDGE_RISING  -> Rising Edge
//   0x1  -> pipxfg_TRIG_IN_EDGE_FALLING -> Falling Edge
//   0x2  -> pipxfg_TRIG_IN_LEVEL_HIGH   -> Generation on High Level of TRIG IN
//   0x3  -> pipxfg_TRIG_IN_LEVEL_LOW    -> Generation on Low Level of TRIG IN
ViUInt32 in_trigger_source = pipxfg_TRIG_IN_FRONT;
ViUInt32 in_trigger_type = pipxfg_TRIG_IN_EDGE_RISING;
status = status = pipxfg_setInputTriggerConfig(handle, in_trigger_source, in_trigger_type);
```

NOTE: If Source or Trigger parameter is outside the specified range, the above functions will return the `ER_BAD_ATTR_VALUE`/`pipx40_ERROR_BAD_ATTR_VALUE` error.

Get Input Trigger Config:

PILFG:

```
/* ============================================================================
* Function:   Get the Input Trigger configuration data of the given card.
* Arguments:  CardNum - Pickering card reference of target (unity-based),
*             Source - pointer to DWORD variable to accept result
*             Trigger - pointer to DWORD variable to accept result
* Returns:    0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetInputTriggerConfig(DWORD CardNum,
                                           DWORD* Source,
                                           DWORD* Trigger) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Get the Clock Source of the given card
//       vi (in)           Instrument handle
//       source (out)      source of the clock signal
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_getClockSource(ViSession vi, ViPUInt32 source) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Source:
//   0x0 -> PILFG_TRIG_IN_FRONT -> Front Panel
//   Trigger Input Modes:
//   0x0  -> PILFG_TRIG_IN_EDGE_RISING   -> Rising Edge
//   0x1  -> PILFG_TRIG_IN_EDGE_FALLING  -> Falling Edge
//   0x2  -> PILFG_TRIG_IN_LEVEL_HIGH    -> Generation on High Level of TRIG IN
//   0x3  -> PILFG_TRIG_IN_LEVEL_LOW     -> Generation on Low Level of TRIG IN
DWORD in_trigger_source;
DWORD in_trigger_type;
status = PILFG_GetInputTriggerConfig(cardNum, &in_trigger_source, &in_trigger_type);
```

PIPXFG:

```
// Source:
//   0x0 -> pipxfg_TRIG_IN_FRONT -> Front Panel
// Trigger Input Modes:
//   0x0  -> pipxfg_TRIG_IN_EDGE_RISING  -> Rising Edge
//   0x1  -> pipxfg_TRIG_IN_EDGE_FALLING -> Falling Edge
//   0x2  -> pipxfg_TRIG_IN_LEVEL_HIGH   -> Generation on High Level of TRIG IN
//   0x3  -> pipxfg_TRIG_IN_LEVEL_LOW    -> Generation on Low Level of TRIG IN
ViUInt32 in_trigger_source;
ViUInt32 in_trigger_type;
status = status = pipxfg_getInputTriggerConfig(handle, &in_trigger_source, &in_trigger_type);
```

Set Output Trigger Config:

PILFG:

```
/* ============================================================================
* Function:  Configure the Output Trigger of the given card.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            Trigger - output trigger type to set (specified in PILFG_TRIG_OUT_MODE).
* Returns:   0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_SetOutputTriggerConfig(DWORD CardNum, DWORD Trigger) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Configure the Output Trigger of the given card
//       vi (in)           Instrument handle
//       trigger (in)      output trigger type to set
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_setOutputTriggerConfig(ViSession vi, ViUInt32 trigger) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Trigger Output Modes:
//   0x0  -> PILFG_TRIG_OUT_GEN_PULSE_POS -> Positive Pulse upon Start Generation Event
//   0x1  -> PILFG_TRIG_OUT_GEN_PULSE_NEG -> Negative Pulse upon Start Generation Event
//   0x2  -> PILFG_TRIG_OUT_SOFT_PULSE_POS -> Positive Pulse upon Gennerate Output Trigger CMD
//   0x3  -> PILFG_TRIG_OUT_SOFT_PULSE_NEG -> Negative Pulse upon Gennerate Output Trigger CMD
//   0x4  -> PILFG_TRIG_OUT_GEN_LEVEL_HIGH -> TRIG IN High when generation ongoing
//   0x5  -> PILFG_TRIG_OUT_GEN_LEVEL_LOW -> TRIG IN Low when generation ongoing
//   0x6  -> PILFG_TRIG_OUT_SOFT_LEVEL -> TRIG IN set High/Low via Gennerate Output Trigger CMD
DWORD out_trigger_source = PILFG_TRIG_OUT_GEN_PULSE_POS;
status = PILFG_SetOutputTriggerConfig(cardNum, out_trigger_type);
```

PIPXFG:

```
// Trigger Output Modes:
//   0x0  -> pipxfg_TRIG_OUT_GEN_PULSE_POS -> Positive Pulse upon Start Generation Event
//   0x1  -> pipxfg_TRIG_OUT_GEN_PULSE_NEG -> Negative Pulse upon Start Generation Event
//   0x2  -> pipxfg_TRIG_OUT_SOFT_PULSE_POS -> Positive Pulse upon Gennerate Output Trigger CMD
//   0x3  -> pipxfg_TRIG_OUT_SOFT_PULSE_NEG -> Negative Pulse upon Gennerate Output Trigger CMD
//   0x4  -> pipxfg_TRIG_OUT_GEN_LEVEL_HIGH -> TRIG IN High when generation ongoing
//   0x5  -> pipxfg_TRIG_OUT_GEN_LEVEL_LOW -> TRIG IN Low when generation ongoing
//   0x6  -> pipxfg_TRIG_OUT_SOFT_LEVEL -> TRIG IN set High/Low via Gennerate Output Trigger CMD
ViUInt32 out_trigger_type = pipxfg_TRIG_OUT_GEN_PULSE_POS;
status = pipxfg_setOutputTriggerConfig(handle, out_trigger_type);
```

NOTE: If the Trigger parameter is outside the specified range, the above functions will return the `ER_BAD_ATTR_VALUE/ pipx40_ERROR_BAD_ATTR_VALUE` error.

Get Output Trigger Config:

PILFG:

```
/* ============================================================================
* Function:  Get the Output Trigger configuration data of the given card.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            Trigger - pointer to DWORD variable to accept result.
* Returns:   0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetOutputTriggerConfig(DWORD CardNum, DWORD* Trigger) MYATTR;
```

PIPXFG:

```
//****************************************************************************
// Get the Output Trigger configuration data of the given card
//       vi (in)           Instrument handle
//       trigger (out)     output trigger type
//       Return = ViStatus
//****************************************************************************
ViStatus _VI_FUNC pipxfg_getOutputTriggerConfig(ViSession vi, ViPUInt32 trigger) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Trigger Output Modes:
//   0x0  -> PILFG_TRIG_OUT_GEN_PULSE_POS -> Positive Pulse upon Start Generation Event
//   0x1  -> PILFG_TRIG_OUT_GEN_PULSE_NEG -> Negative Pulse upon Start Generation Event
//   0x2  -> PILFG_TRIG_OUT_SOFT_PULSE_POS -> Positive Pulse upon Gennerate Output Trigger CMD
//   0x3  -> PILFG_TRIG_OUT_SOFT_PULSE_NEG -> Negative Pulse upon Gennerate Output Trigger CMD
//   0x4  -> PILFG_TRIG_OUT_GEN_LEVEL_HIGH-> TRIG IN High when generation ongoing
//   0x5  -> PILFG_TRIG_OUT_GEN_LEVEL_LOW -> TRIG IN Low when generation ongoing
//   0x6  -> PILFG_TRIG_OUT_SOFT_LEVEL -> TRIG IN set High/Low via Gennerate Output Trigger CMD
DWORD out_trigger_source;
status = PILFG_SetOutputTriggerConfig(cardNum, &out_trigger_type);
```

PIPXFG:

```
// Trigger Output Modes:
//   0x0  -> pipxfg_TRIG_OUT_GEN_PULSE_POS -> Positive Pulse upon Start Generation Event
//   0x1  -> pipxfg_TRIG_OUT_GEN_PULSE_NEG -> Negative Pulse upon Start Generation Event
//   0x2  -> pipxfg_TRIG_OUT_SOFT_PULSE_POS -> Positive Pulse upon Gennerate Output Trigger CMD
//   0x3  -> pipxfg_TRIG_OUT_SOFT_PULSE_NEG -> Negative Pulse upon Gennerate Output Trigger CMD
//   0x4  -> pipxfg_TRIG_OUT_GEN_LEVEL_HIGH -> TRIG IN High when generation ongoing
//   0x5  -> pipxfg_TRIG_OUT_GEN_LEVEL_LOW -> TRIG IN Low when generation ongoing
//   0x6  -> pipxfg_TRIG_OUT_SOFT_LEVEL -> TRIG IN set High/Low via Gennerate Output Trigger CMD
ViUInt32 out_trigger_type;
status = pipxfg_setOutputTriggerConfig(handle, out_trigger_type);
```

Set Input Trigger Enable:

PILFG:

```
/* ============================================================================
* Function:  Enable Input Trigger of the given sub-unit.
*            The function determines whether the configured input trigger signal
*            sets off generation of the given sub-unit.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            SubNum - sub-unit of target to access (unity-based),
*            InputTriggerEnable - DWORD-type array defining the input trigger enable data.
*           (*InputTriggerEnable) = 0 -> sub-unit ignores the trigger input signal,
*                (the configured trigger signal will not trigger the generation),
*           (*InputTriggerEnable) = 1 -> sub-unit responds to the trigger input signal,
*                (the configured trigger signal will trigger the generation).
*
*    If the SubNum parameter is set to 0, the function determines the input trigger
*    handling of all sub-units. The memory location, specified by the Trigger and the Size
*    parameters, stores information defining the way respective sub-units react to the
*    input trigger signal.
*    Sub-units are mapped as follows:
*            LSB -> sub-unit 1, MSB (for 32 channel card) -> sub-unit 32.
*    The input trigger signal handling is encoded as follows:
*            Bit = 0 -> sub-unit ignores the trigger input signal,
*                (the configured trigger signal will not trigger the generation),
*            Bit = 1 -> sub-unit responds to the trigger input signal,
*                (the configured trigger signal will trigger the generation).
*
*        Size - size of the array.
*        Returns:     0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_SetInputTriggerEnable(DWORD CardNum,
                                           DWORD SubNum,
                                           DWORD* InputTriggerEnable,
                                           DWORD Size) MYATTR;
```

PIPXFG:

```
//****************************************************************************
// Enable Input Trigger of the given sub-unit.
// The function determines whether the configured input trigger signal
// sets off generation of the given sub-unit.
//       vi (in)                   Instrument handle
//       subUnit (in)              Sub-unit number
//       inputTriggerEnable (in)   ViUInt32-type array defining the input trigger enable data.
//            (*inputTriggerEnable) = 0-> sub-unit ignores the trigger input signal,
//                (the configured trigger signal will not trigger the generation),
//            (*inputTriggerEnable) = 1-> sub-unit responds to the trigger input signal,
//                (the configured trigger signal will trigger the generation).
//
```

```
//   If the subUnit parameter is set to 0, the function determines the input trigger
//   handling of all sub-units. The memory location, specified by the Trigger and the Size
//   parameters, stores information defining the way respective sub-units react to the
//   input trigger signal.
//   Sub-units are mapped as follows:
//        LSB-> sub-unit 1, MSB (for 32 channel card) -> sub-unit 32.
//   The input trigger signal handling is encoded as follows:
//        Bit = 0-> sub-unit ignores the trigger input signal,
//             (the configured trigger signal will not trigger the generation),
//        Bit = 1-> sub-unit responds to the trigger input signal,
//             (the configured trigger signal will trigger the generation).
//
//        size (in)          size of the array
//        Return = ViStatus
//*************************************************************************
ViStatus _VI_FUNC pipxfg_setInputTriggerEnable(ViSession vi,
                                               ViUInt32 subUnit,
                                               ViPUInt32 inputTriggerEnable,
                                               ViUInt32 size) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Input Trigger Enable:
//   0x0 -> Input Trigger Disabled - channel is triggered via Initiate Generation CMD only
//   0x1 -> Input Trigger Enabled - channel is triggered via Input Trigger
DWORD in_trigger_enable = 0x0;
DWORD size = 0x1;
status = PILFG_SetInputTriggerEnable(cardNum, channel, &in_trigger_enable, size);
```

PIPXFG:

```
// Input Trigger Enable:
//   0x0 -> Input Trigger Disabled - channel is triggered via Initiate Generation CMD only
//   0x1 -> Input Trigger Enabled - channel is triggered via Input Trigger
ViUInt32 in_trigger_enable = 0x0;
ViUInt32 size = 0x1;
status = pipxfg_setInputTriggerEnable(handle, channel, &in_trigger_enable, size);
```

NOTE: If the size of the array does not correspond the number of channels, the above functions return `ER_BAD_ATTR_VALUE`/`pipx40_ERROR_BAD_ATTR_VALUE` error.

NOTE: The feature is available for channels in configuration state only. The above functions will return the `ER_DDS_ON`/`pipx40_ERROR_DDS_ON` error if the channel is generating a signal or is actively sampling the trigger input line for the configured trigger signal.

Get Input Trigger Enable:

PILFG:

```
/* ============================================================================
* Function:  Get the Input Trigger Enable data of the given sub-unit.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            SubNum - sub-unit of target to access (unity-based),
*            InputTriggerEnable - DWORD-type array storing the input trigger enable data.
*            (*InputTriggerEnable) = 0 -> sub-unit ignores the trigger input signal,
*            (*InputTriggerEnable) = 1 -> sub-unit responds to the trigger input signal.
*
*    If the SubNum parameter is set to 0, the function provides the input trigger handling
*    of all sub-units. The memory location, specified by the Trigger and the Size
*    parameters, stores information reflecting the way respective sub-units react to the
*    input trigger signal.
*    Sub-units are mapped as follows:
*        LSB -> sub-unit 1, MSB (for 32 channel card) -> sub-unit 32.
*    The input trigger signal handling is decoded as follows:
*        Bit = 0 -> sub-unit ignores the trigger input signal,
*        Bit = 1 -> sub-unit responds to the trigger input signal.
*
*        Size - size of the array.
* Returns:        0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetInputTriggerEnable(DWORD CardNum,
                                           DWORD SubNum,
                                           DWORD* InputTriggerEnable,
                                           DWORD Size) MYATTR;
```

PIPXFG:

```
//***************************************************************************
// Get the Input Trigger Enable data of the given sub-unit.
//       vi (in)               Instrument handle
//       subUnit (in)          Sub-unit number
//       inputTriggerEnable (out)   ViUInt32-type array storing the input trigger
//                                  enable data.
//            (*inputTriggerEnable) = 0-> sub-unit ignores the trigger input signal,
//            (*inputTriggerEnable) = 1-> sub-unit responds to the trigger input signal,
//
//   If the subUnit parameter is set to 0, the function provides the input trigger handling
//   of all sub-units. The memory location, specified by the Trigger and the Size
//   parameters, stores information reflecting the way respective sub-units react to the
//   input trigger signal.
//       Sub-units are mapped as follows:
//            LSB-> sub-unit 1, MSB (for 32 channel card) -> sub-unit 32.
//       The input trigger signal handling is decoded as follows:
//            Bit = 0 -> sub-unit ignores the trigger input signal,
//            Bit = 1 -> sub-unit responds to the trigger input signal,
```

```
//
//       size (in)         size of the array
//       Return = ViStatus
//*************************************************************************
ViStatus _VI_FUNC pipxfg_getInputTriggerEnable(ViSession vi,
                                               ViUInt32 subUnit,
                                               ViPUInt32 inputTriggerEnable,
                                               ViUInt32 size) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Input Trigger Enable:
//   0x0 -> Input Trigger Disabled - channel is triggered via Initiate Generation CMD only
//   0x1 -> Input Trigger Enabled - channel is triggered via Input Trigger
DWORD in_trigger_enable;
DWORD size = 0x1;
status = PILFG_GetInputTriggerEnable(cardNum, channel, &in_trigger_enable, size);
```

PIPXFG:

```
// Input Trigger Enable:
//   0x0 -> Input Trigger Disabled - channel is triggered via Initiate Generation CMD only
//   0x1 -> Input Trigger Enabled - channel is triggered via Input Trigger
ViUInt32 in_trigger_enable;
ViUInt32 size = 0x1;
status = pipxfg_getInputTriggerEnable(handle, channel, &in_trigger_enable, size);
```

NOTE: If the size of the array does not correspond the number of channels, the above functions return `ER_BAD_ATTR_VALUE`/`pipx40_ERROR_BAD_ATTR_VALUE` error.

Set Output Trigger Enable:

PILFG:

```
/* ============================================================================
* Function:  Enable Output Trigger of the given sub-unit.
*            The function determines whether the configured output trigger signal is
*            generated when the given sub-unit starts generation.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            SubNum - sub-unit of target to access (unity-based),
*            OutputTriggerEnable - DWORD-type array defining the output trigger enable data.
*            (*OutputTriggerEnable) = 0 -> sub-unit does not generate the trigger
*                output signal,
*            (*OutputTriggerEnable) = 1 -> sub-unit generates the trigger output
*                signal.
*
*    If the SubNum parameter is set to 0, the function determines the output trigger
*    handling of all sub-units. The memory location, specified by the Trigger and the Size
*    parameters, stores information defining the way respective sub-units proceed the
*    output trigger signal.
*    Sub-units are mapped as follows:
*        LSB -> sub-unit 1, MSB (for 32 channel card) -> sub-unit 32.
*    The output trigger signal handling is encoded as follows:
*        Bit = 0 -> sub-unit does not generate the trigger output signal,
*        Bit = 1 -> sub-unit generates the trigger output signal.
*
*        Size - size of the array.
* Returns:        0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_SetOutputTriggerEnable(DWORD CardNum,
                                            DWORD SubNum,
                                            DWORD* OutputTriggerEnable,
                                            DWORD Size) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Enable Output Trigger of the given sub-unit.
// The function determines whether the configured output trigger signal is generated
// when the given sub-unit starts generation.
//       vi (in)                   Instrument handle
//       subUnit (in)              Sub-unit number
//       outputTriggerEnable (in)  ViUInt32-type array defining the output trigger
//                          enable data.
//            (*outputTriggerEnable) = 0 -> sub-unit does not generate the trigger
//                output signal,
//            (*outputTriggerEnable) = 1 -> sub-unit generates the trigger output signal.
//
//   If the SubNum parameter is set to 0, the function determines the output trigger
//   handling of all sub-units.The memory location, specified by the Trigger and the Size
```

```
//   parameters, stores information defining the way respective sub-units proceed the
//   output trigger signal.
//   Sub-units are mapped as follows:
//        LSB-> sub-unit 1, MSB(for 32 channel card) -> sub-unit 32.
//   The output trigger signal handling is encoded as follows:
//        Bit = 0 -> sub-unit does not generate the trigger output signal,
//        Bit = 1 -> sub-unit generates the trigger output signal.
//
//        size (in)         size of the array
//        Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_setOutputTriggerEnable(ViSession vi,
                                                ViUInt32 subUnit,
                                                ViPUInt32 outputTriggerEnable,
                                                ViUInt32 size) MYATTR;
```

Example Code Snippets:

PILFG:

```
//   Output Trigger Enable:
//   0x0 -> Output Trigger Disabled - Start Generation Event has no effect on the Trigger
//   Output Signal
//   0x1 -> Output Trigger Enabled - Start Generation Event will be followed by the Trigger
//   Output Signal
DWORD out_trigger_enable = 0x0;
DWORD size = 0x1;
status = PILFG_SetOutputTriggerEnable(cardNum, channel, &out_trigger_enable, size);
```

PIPXFG:

```
// Output Trigger Enable:
//   0x0 -> Output Trigger Disabled - Start Generation Event has no effect on the Trigger
//   Output Signal
//   0x1 -> Output Trigger Enabled - Start Generation Event will be followed by the Trigger
//   Output Signal
ViUInt32 out_trigger_enable = 0x0;
ViUInt32 size = 0x1;
status = pipxfg_setOutputTriggerEnable(handle, channel, &out_trigger_enable, size);
```

NOTE: If the size of the array does not correspond the number of channels, the above functions return `ER_BAD_ATTR_VALUE`/`pipx40_ERROR_BAD_ATTR_VALUE` error.

NOTE: The feature is available for channels in configuration state only. The above functions will return the `ER_DDS_ON`/`pipx40_ERROR_DDS_ON` error if the channel is generating a signal or is actively sampling the trigger input line for the configured trigger signal.

Get Output Trigger Enable:

PILFG:

```
/* ============================================================================
* Function:  Get the Output Trigger Enable data of the given sub-unit.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            SubNum - sub-unit of target to access (unity-based),
*            OutputTriggerEnable - DWORD-type array storing the output trigger enable data.
*            (*OutputTriggerEnable) = 0 -> sub-unit does not generate the trigger
*                          output signal,
*            (*OutputTriggerEnable) = 1 -> sub-unit generates the trigger output signal.
*
*    If the SubNum parameter is set to 0, the function provides the output trigger handling
*    of all sub-units. The memory location, specified by the Trigger and the Size
*    parameters, stores information reflecting the way respective sub-units proceed the
*    output trigger signal.
*    Sub-units are mapped as follows:
*        LSB -> sub-unit 1, MSB (for 32 channel card) -> sub-unit 32.
*    The output trigger signal handling is decoded as follows:
*        Bit = 0 -> sub-unit does not generate the trigger output signal,
*        Bit = 1 -> sub-unit generates the trigger output signal.
*
*        Size - size of the array.
* Returns:        0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetOutputTriggerEnable(DWORD CardNum,
                                            DWORD SubNum,
                                            DWORD* OutputTriggerEnable,
                                            DWORD Size) MYATTR;
```

PIPXFG:

```
//***************************************************************************
// Get the Output Trigger Enable data of the given sub-unit.
//       vi (in)                   Instrument handle
//       subUnit (in)              Sub-unit number
//       outputTriggerEnable (out) ViUInt32-type array storing the output trigger enable data.
//            (*outputTriggerEnable) = 0 -> sub-unit does not generate the trigger
//                          output signal,
//            (*outputTriggerEnable) = 1 -> sub-unit generates the trigger output signal.
//
//   If the SubNum parameter is set to 0, the function provides the output trigger handling
//   of all sub-units. The memory location, specified by the Trigger and the Size
//   parameters, stores information reflecting the way respective sub-units proceed the
//   output trigger signal.
//   Sub-units are mapped as follows:
//       LSB-> sub-unit 1, MSB(for 32 channel card) -> sub-unit 32.
//   The output trigger signal handling is decoded as follows:
//       Bit = 0 -> sub-unit does not generate the trigger output signal,
```

```
//       Bit = 1 -> sub-unit generates the trigger output signal.
//
//       size (in)         size of the array
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_getOutputTriggerEnable(ViSession vi,
                                                ViUInt32 subUnit,
                                                ViPUInt32 outputTriggerEnable,
                                                ViUInt32 size) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Output Trigger Enable:
//   0x0 -> Output Trigger Disabled - Start Generation Event has no effect on the Trigger
//   Output Signal
//   0x1 -> Output Trigger Enabled - Start Generation Event will be followed by the Trigger
//   Output Signal
DWORD out_trigger_enable;
DWORD size = 0x1;
status = PILFG_GetOutputTriggerEnable(cardNum, channel, &out_trigger_enable, size);
```

PIPXFG:

```
// Output Trigger Enable:
//   0x0 -> Output Trigger Disabled - Start Generation Event has no effect on the Trigger
//   Output Signal
//   0x1 -> Output Trigger Enabled - Start Generation Event will be followed by the Trigger
//   Output Signal
ViUInt32 out_trigger_enable;
ViUInt32 size = 0x1;
status = pipxfg_getOutputTriggerEnable(handle, channel, &out_trigger_enable, size);
```

NOTE: If the size of the array does not correspond the number of channels, the above functions return `ER_BAD_ATTR_VALUE`/`pipx40_ERROR_BAD_ATTR_VALUE` error.

Generate Output Trigger:

PILFG:

```
/* ============================================================================
* Function:   Generate the Output Trigger of the given card.
* Arguments:  CardNum - Pickering card reference of target (unity-based),
*             State - State of the trigger output pin to set.
* Returns:    0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GenerateOutputTrigger(DWORD CardNum, DWORD State) MYATTR;
```

PIPXFG:

```
//****************************************************************************
// Generate the Output Trigger of the given card
//       vi (in)           Instrument handle
//       state (in)        state of the trigger output pin to set
//       Return = ViStatus
//****************************************************************************
ViStatus _VI_FUNC pipxfg_generateOutputTrigger(ViSession vi, ViUInt32 state) MYATTR;
```

Example Code Snippets:

PILFG:

```
//   state = 0x0 -> TRIG OUT pin will be reset
//   state = 0x1 -> TRIG OUT pin will be set
DWORD state = 0x1;
status = PILFG_GenerateOutputTrigger (cardNum, state);
```

PIPXFG:

```
//   state = 0x0 -> TRIG OUT pin will be reset
//   state = 0x1 -> TRIG OUT pin will be set
ViUInt32 out_trigger_enable;
status = pipxfg_getOutputTriggerEnable(handle, state);
```

NOTE: State parameter has effect only when output triggering mode is set to `PILFG_TRIG_OUT_SOFT_LEVEL`/`pipxfg_TRIG_OUT_SOFT_LEVEL`. For other output triggering modes, the TRIG OUT pin state is defined via Set Output Trigger Config feature.

NOTE: If the State parameter is outside the specified range, the above functions will return the `ER_BAD_ATTR_VALUE`/`pipx40_ERROR_BAD_ATTR_VALUE` error.

NOTE: If the output trigger is not in a software mode, the above functions will return the `ER_BAD_MODE`/`pipx40_ERROR_BAD_MODE` error.

Get Trigger Monitor State:

PILFG:

```
/* ============================================================================
* Function:  Get the trigger monitor state of the given sub-unit.
*            The function determines whether the sub-unit is actively sampling
*            the trigger input line for the configured trigger signal.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            SubNum - sub-unit of target to access (unity-based),
*            State - DWORD-type array storing trigger monitor data,
*                 (*State) = 0 -> sub-unit is not sampling the trigger input line
*                      (the configured trigger signal will not trigger the generation),
*                 (*State) = 1 -> sub-unit is sampling the trigger input line
*                      (the configured trigger signal will trigger the generation).
*
*    If the function is called with SubNum parameter set to 0, it provides the trigger
*    monitor data of all channels to the memory location specified by the State and the
*    Size parameters. Then, each bit of the array stores the trigger monitor data of the
*    corresponding sub-unit.
*    Sub-units are mapped as follows:
*        LSB -> sub-unit 1, MSB (for 32 channel card) -> sub-unit 32.
*    Trigger Monitor State is decoded as follows:
*        Bit = 0 -> sub-unit is not sampling the trigger input line
*             (the configured trigger signal will not trigger the generation),
*        Bit = 1 -> sub-unit is sampling the trigger input line
*             (the configured trigger signal will trigger the generation).
*
*        Size - size of the array.
* Returns:        0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetTriggerMonitorState(DWORD CardNum,
                                            DWORD SubNum,
                                            DWORD* State,
                                            DWORD Size) MYATTR;
```

PIPXFG:

```
//****************************************************************************
// Get the trigger monitor state of the given sub-unit.
// The function determines whether the sub-unit is actively sampling
// the trigger input line for the configured trigger signal.
//       vi (in)       Instrument handle
//       subUnit (in)  Sub-unit number
//       trigger (out) ViUInt32-type array storing trigger monitor data.
//            (*State) = 0 -> sub-unit is not sampling the trigger input line
//                 (the configured trigger signal will not trigger the generation),
//            (*State) = 1 -> sub-unit is sampling the trigger input line
//                 (the configured trigger signal will trigger the generation).
//
```

```
//   If the function is called with SubNum parameter set to 0, it provides the trigger
//   monitor data of all channels to the memory location specified by the State and the
//   Size parameters. Then, each bit of the array stores the trigger monitor data of the
//   corresponding sub-unit.
//   Sub-units are mapped as follows:
//        LSB-> sub-unit 1, MSB(for 32 channel card) -> sub-unit 32.
//   Trigger Monitor State is decoded as follows:
//        Bit = 0 -> sub-unit is not sampling the trigger input line
//             (the configured trigger signal will not trigger the generation),
//        Bit = 1 -> sub-unit is sampling the trigger input line
//             (the configured trigger signal will trigger the generation).
//
//        size (in)         size of the array
//        Return = ViStatus
//***************************************************************************
ViStatus _VI_FUNC pipxfg_getTriggerMonitorState(ViSession vi,
                                                ViUInt32 subUnit,
                                                ViPUInt32 state,
                                                ViUInt32 size) MYATTR;
```

Example Code Snippets:

PILFG:

```
//   0x0 -> channels is not listening for the configured trigger signal
//   0x1 -> channels is listening for the configured trigger signal
DWORD trigger_monitor_state = 0x1;
DWORD size = 0x1;
status = PILFG_GetTriggerMonitorState(cardNum, channel, &trigger_monitor_state, size);
```

PIPXFG:

```
//   0x0 -> channels is not listening for the configured trigger signal
//   0x1 -> channels is listening for the configured trigger signal
ViUInt32 out_trigger_enable;
ViUInt32 size = 0x1;
status = pipxfg_getTriggerMonitorState(handle, channel, &trigger_monitor_state, size);
```

NOTE: If the size of the array does not correspond the number of channels, the above functions return `ER_BAD_ATTR_VALUE`/`pipx40_ERROR_BAD_ATTR_VALUE` error.

Set Clock Source:

PILFG:

```
/* ============================================================================
* Function:   Set the Clock Source of the given card.
* Arguments:  CardNum - Pickering card reference of target (unity-based),
*             Source - source of the clock signal to set (specified in PILFG_CLK_SRC).
* Returns:    0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_SetClockSource(DWORD CardNum, DWORD Source) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Set the Clock Source of the given card
//       vi (in)           Instrument handle
//       source (in)       source of the clock signal to set
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_setClockSource(ViSession vi, ViUInt32 source) MYATTR;
```

Example Code Snippets:

PILFG:

```
//   0x0 -> pipxfg_CLK_ONB_OSC  -> On-Board Oscillator
//   0x1 -> pipxfg_CLK_PXI  -> 10MHz PXI
DWORD clk = 0x0;
status = PILFG_SetClockSource(cardNum, clk);
```

PIPXFG:

```
//   0x0 -> pipxfg_CLK_ONB_OSC  -> On-Board Oscillator
//   0x1 -> pipxfg_CLK_PXI  -> 10MHz PXI
ViUInt32 clk = 0x0;
status = pipxfg_setClockSource(handle, clk);
```

NOTE: If the clock source is unavailable the above functions will return `ER_HARDWARE_FAULT/pipx40_ERROR_HARDWARE_FAULT` error.

NOTE: If the Source parameter is outside the specified range, the above functions will return the `ER_BAD_ATTR_VALUE/pipx40_ERROR_BAD_ATTR_VALUE` error.

Get Clock Source:

PILFG:

```
/* ============================================================================
* Function:  Get the Clock Source of the given card.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            Source - pointer to DWORD variable to accept result.
* Returns:   0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetClockSource(DWORD CardNum, DWORD* Source) MYATTR;
```

PIPXFG:

```
//*****************************************************************************
// Get the Clock Source of the given card
//       vi (in)           Instrument handle
//       source (out)      source of the clock signal
//       Return = ViStatus
//*****************************************************************************
ViStatus _VI_FUNC pipxfg_getClockSource(ViSession vi, ViPUInt32 source) MYATTR;
```

Example Code Snippets:

PILFG:

```
//   0x0 -> pipxfg_CLK_ONB_OSC  -> On-Board Oscillator
//   0x1 -> pipxfg_CLK_PXI  -> 10MHz PXI
DWORD clk;
status = PILFG_GetClockSource(cardNum, &clk);
```

PIPXFG:

```
//   0x0 -> pipxfg_CLK_ONB_OSC  -> On-Board Oscillator
//   0x1 -> pipxfg_CLK_PXI  -> 10MHz PXI
ViUInt32 clk;
status = pipxfg_getClockSource(handle, &clk);
```

Get Parameter Limits:

PILFG:

```
/* ============================================================================
* Function:  Retrieve parameter limit values.
*            The function provides limit values of parameters specified in
*            PILFG_PARAM_DWORD.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            Parameter - DWORD variable indicating parameter limit values to retrieve,
*            MinLimit - pointer to DWORD variable to store minimum limit value,
*            MaxLimit - pointer to DWORD variable to store maximum limit value.
* Returns:   0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetParameterLimitsDWORD(DWORD CardNum, DWORD Parameter,
                                             DWORD* MinLimit, DWORD* MaxLimit) MYATTR;

/* ============================================================================
* Function:  Retrieve parameter limit values.
*            The function provides limit values of parameters specified in
*            PILFG_PARAM_DOUBLE.
* Arguments: CardNum - Pickering card reference of target (unity-based),
*            Parameter - DWORD variable indicating parameter limit values to retrieve,
*            MinLimit - pointer to double variable to store minimum limit value,
*            MaxLimit - pointer to double variable to store maximum limit value.
* Returns:   0 - success; non-zero - error code.
*/
DWORD _stdcall PILFG_GetParameterLimitsDOUBLE(DWORD CardNum, DWORD Parameter,
                                              double* MinLimit, double* MaxLimit) MYATTR;
```

Example Code Snippets:

PILFG:

```
// Parameters DWORD:
// PILFG_PARAM_BUFFER_SIZE = 0x0,
DWORD parameter = PILFG_PARAM_BUFFER_SIZE;
DWORD min_val;
DWORD max_val;
status = PILFG_GetParameterLimitsDWORD(cardNum, parameter, &min_val, &max_val);

// Parameters DOUBLE:
// PILFG_PARAM_AMPLITUDE   = 0x0,
// PILFG_PARAM_VOLTAGE= 0x1,
// PILFG_PARAM_FREQUENCY   = 0x2
DWORD parameter = PILFG_PARAM_BUFFER_SIZE;
double min_val;
double max_val;
status = PILFG_GetParameterLimitsDOUBLE(cardNum, parameter, &min_val, &max_val);
```

PIPXFG:

```
//************************************************************************
// Retrieve parameter limit values.
// The function provides limit values of the ViUInt32-type parameters.
//       vi (in)           Instrument handle
//       parameter (in)    variable indicating parameter limit values to retrieve
//       minLimit (out)    minimum limit value of the specified parameter
//       minLimit (out)    maximum limit value of the specified parameter
//       Return = ViStatus
//************************************************************************
ViStatus _VI_FUNC pipxfg_getParameterLimitsViUInt32(ViSession vi, ViUInt32 parameter,
                                          ViPUInt32 minLimit, ViPUInt32 maxLimit) MYATTR;

//************************************************************************
// Retrieve parameter limit values.
// The function provides limit values of the ViReal64-type parameters.
//       vi (in)           Instrument handle
//       parameter (in)    variable indicating parameter limit values to retrieve
//       minLimit (out)    minimum limit value of the specified parameter
//       minLimit (out)    maximum limit value of the specified parameter
//       Return = ViStatus
//************************************************************************
ViStatus _VI_FUNC pipxfg_getParameterLimitsViReal64(ViSession vi, ViUInt32 parameter,
                                          ViPReal64 minLimit, ViPReal64 maxLimit) MYATTR;
```

Example Code Snippets:

PIPXFG:

```
// Parameters DWORD:
// pipxfg_PARAM_BUFFER_SIZE = 0x0,
ViUInt32 parameter = pipxfg_PARAM_BUFFER_SIZE;
ViUInt32 min_val;
ViUInt32 max_val;
status = pipxfg_getParameterLimitsViUInt32(handle, parameter, &min_val, &max_val);

// Parameters DOUBLE:
// pipxfg_PARAM_AMPLITUDE  = 0x0,
// pipxfg_PARAM_VOLTAGE    = 0x1,
// pipxfg_PARAM_FREQUENCY  = 0x2
ViUInt32 parameter = pipxfg_PARAM_AMPLITUDE;
ViReal64 min_val;
ViReal64 max_val;
status = pipxfg_getParameterLimitsViReal64(handle, parameter, &min_val, &max_val);
```

NOTE: If the Parameter argument is outside the specified range, the above functions will return the `ER_BAD_ATTR_VALUE/ pipx40_ERROR_BAD_ATTR_VALUE` error.

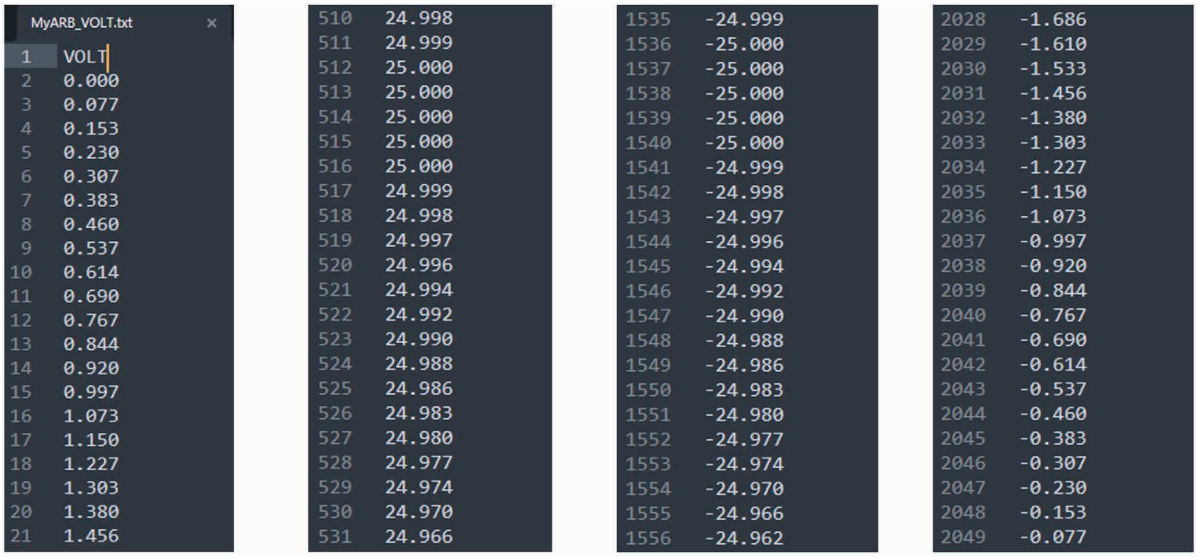
Sample Source File Requirements for Arbitrary Waveform

Samples can be expressed in the form of voltage or percentage. The first line of the file determines the units in which data is presented.

If the first line of the file is VOLT, the samples will be considered as output voltage.

If the first line of the file is PERCENT, the samples will be considered as percentage of the output range where 0% corresponds to minimum output Voltage, 50 % to 0 VDC, and 100 % corresponds to maximum output Voltage.

Every sample must compose a string no longer than 15 characters containing a dot character. The string must include a new line character which terminates the sample. The number of samples in the file must not be less than the buffer size.



MyARB_VOLT.txt

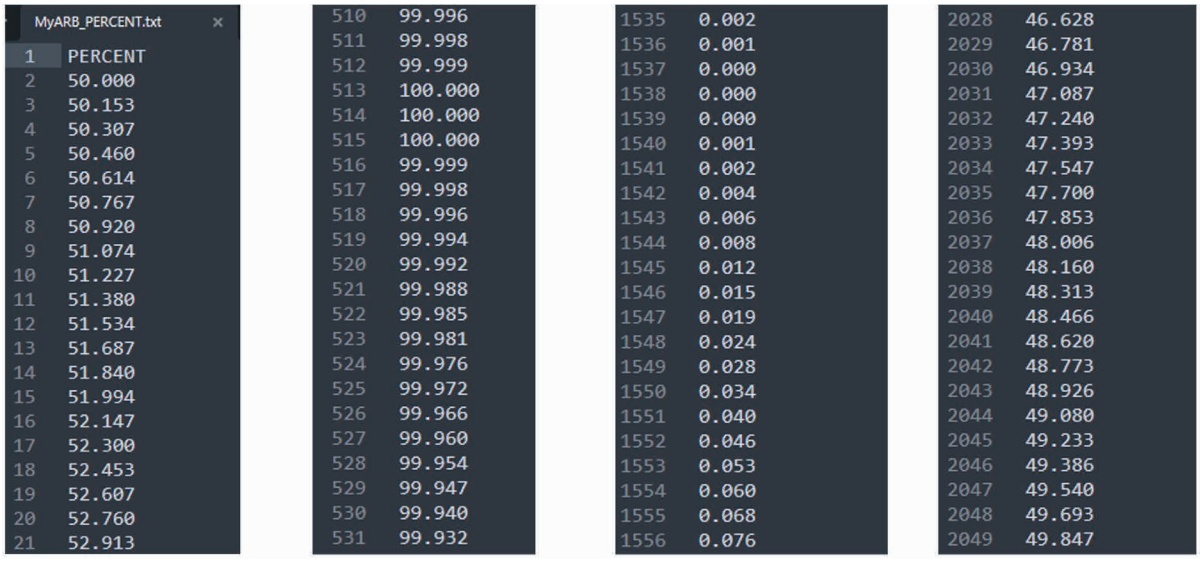
```
VOLT
0.000
0.077
0.153
0.230
0.307
0.383
0.460
0.537
0.614
0.690
0.767
0.844
0.920
0.997
1.073
1.150
1.227
1.303
1.380
1.456
```

```
510  24.998
511  24.999
512  25.000
513  25.000
514  25.000
515  25.000
516  25.000
517  24.999
518  24.998
519  24.997
520  24.996
521  24.994
522  24.992
523  24.990
524  24.988
525  24.986
526  24.983
527  24.980
528  24.977
529  24.974
530  24.970
531  24.966
```

```
1535  -24.999
1536  -25.000
1537  -25.000
1538  -25.000
1539  -25.000
1540  -25.000
1541  -24.999
1542  -24.998
1543  -24.997
1544  -24.996
1545  -24.994
1546  -24.992
1547  -24.990
1548  -24.988
1549  -24.986
1550  -24.983
1551  -24.980
1552  -24.977
1553  -24.974
1554  -24.970
1555  -24.966
1556  -24.962
```

```
2028  -1.686
2029  -1.610
2030  -1.533
2031  -1.456
2032  -1.380
2033  -1.303
2034  -1.227
2035  -1.150
2036  -1.073
2037  -0.997
2038  -0.920
2039  -0.844
2040  -0.767
2041  -0.690
2042  -0.614
2043  -0.537
2044  -0.460
2045  -0.383
2046  -0.307
2047  -0.230
2048  -0.153
2049  -0.077
```

Example Source File - VOLT



MyARB_PERCENT.txt

```
PERCENT
50.000
50.153
50.307
50.460
50.614
50.767
50.920
51.074
51.227
51.380
51.534
51.687
51.840
51.994
52.147
52.300
52.453
52.607
52.760
52.913
```

```
510  99.996
511  99.998
512  99.999
513  100.000
514  100.000
515  100.000
516  99.999
517  99.998
518  99.996
519  99.994
520  99.992
521  99.988
522  99.985
523  99.981
524  99.976
525  99.972
526  99.966
527  99.960
528  99.954
529  99.947
530  99.940
531  99.932
```

```
1535  0.002
1536  0.001
1537  0.000
1538  0.000
1539  0.000
1540  0.001
1541  0.002
1542  0.004
1543  0.006
1544  0.008
1545  0.012
1546  0.015
1547  0.019
1548  0.024
1549  0.028
1550  0.034
1551  0.040
1552  0.046
1553  0.053
1554  0.060
1555  0.068
1556  0.076
```

```
2028  46.628
2029  46.781
2030  46.934
2031  47.087
2032  47.240
2033  47.393
2034  47.547
2035  47.700
2036  47.853
2037  48.006
2038  48.160
2039  48.313
2040  48.466
2041  48.620
2042  48.773
2043  48.926
2044  49.080
2045  49.233
2046  49.386
2047  49.540
2048  49.693
2049  49.847
```

Example Source File - PERCENT

General Soft Front Panel (GSFP)

The following section describes how to control 41/43-625 modules via the General Soft Front Panel.

Channel View

Use configuration tabs to set up waveforms and tailor their parameters to your needs. Once configured, *Initiate/Abort* generation using provided controls. *Generation State* and *Trigger Monitor State* are constantly monitored and displayed in the form of LEDs.

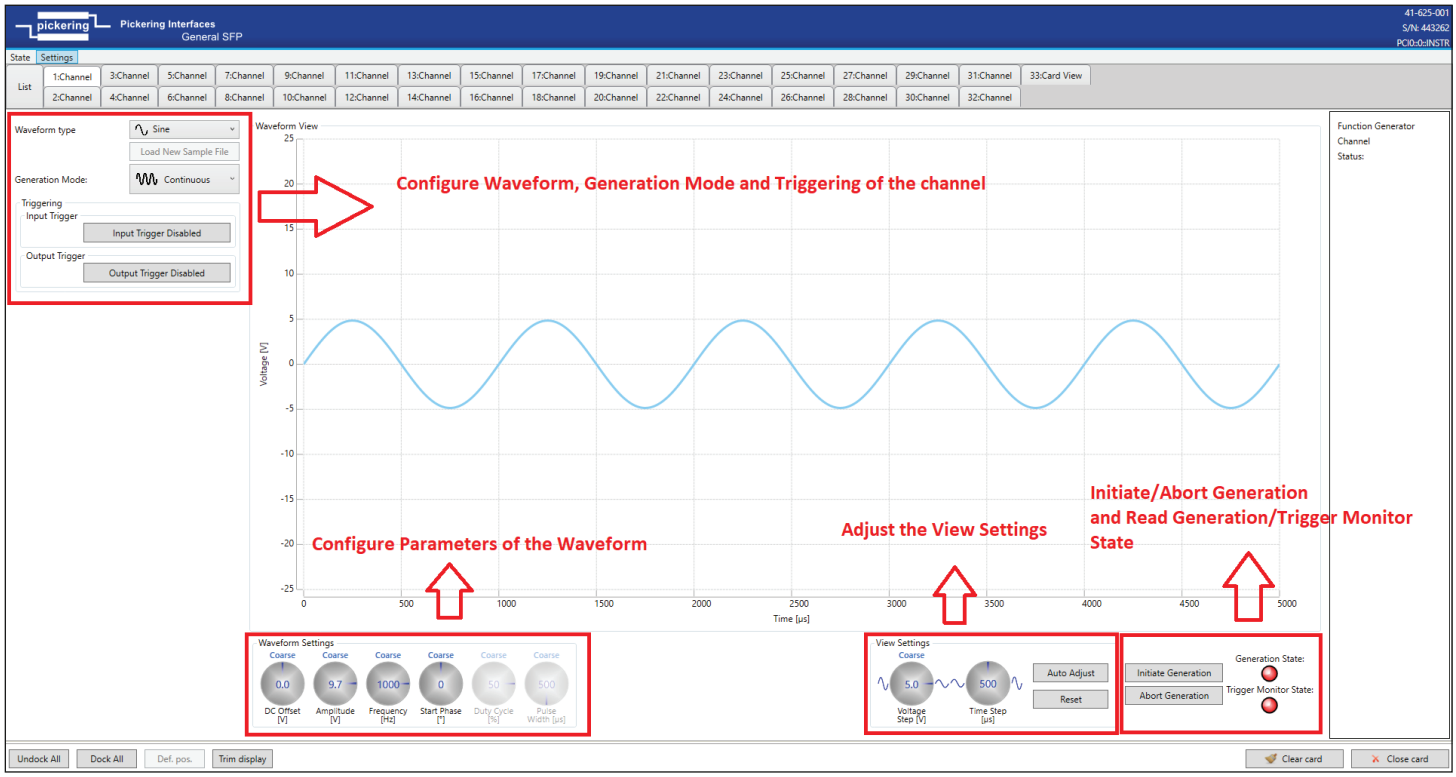


Figure 4.2 - GSFP Channel View

NOTE: Some parameters are applicable only to specific waveform types. Therefore, if the selected waveform does not support a particular parameter, the corresponding knob becomes inactive.

NOTE: Some operations are available only in specific stages of generation. Corresponding buttons/combo boxes become inactive, if the operation is unobtainable.

Card View

Card View tab provides an overview of the generation/triggering indicators and controls.

It also allows a user to set card-specific parameters (like clock source, input/output trigger) and generate output trigger signals (available when a software Output Trigger Mode is selected).

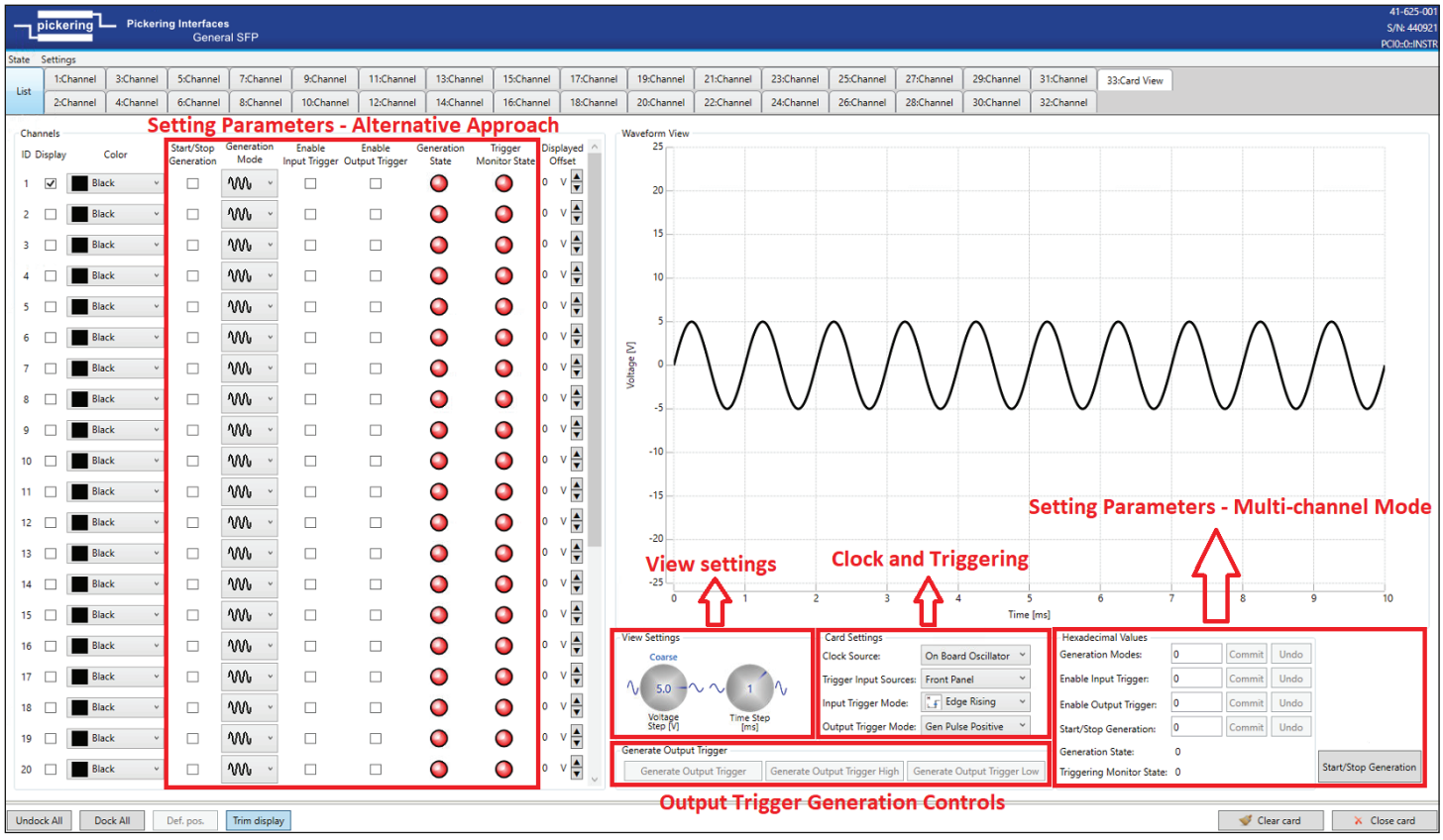


Figure 4.3 - GSFP Card View

The *Hexadecimal Values* section contains features in the multi-channel mode, where a function call affects all channels. It enables a user to set parameters as 32-bit hexadecimal values, where every bit corresponds to a specified channel (e.g. *StartStopGeneration* called with 0x3 initiates channels 1 and 2 and aborts channels 3, 4, ..., 32). Once the value is entered, the *Commit* button becomes active, and the function is called with the provided parameter after clicking the button. *Undo* button discards entered changes unless they are already committed.

If setting all channels at once is not necessary, a user can select a channel by clicking the check box in the corresponding column. In this case, the function is called straight after the check box is updated.

NOTE: *StartStopGeneration* is called via *Start/Stop Generation* button. In this case, the *Commit* button confirms that the parameter value is entered.

Using Pickering Drivers In LabVIEW

Most Pickering drivers include a LabVIEW wrapper to permit full operation of the Pickering product from the LabVIEW environment. These wrappers are normally installed to the current LabVIEW folder system during installation of the Pickering driver.

The process of controlling a Pickering product normally consists of the following steps:

1. Open a control session on the product.
2. Use functions in the library to operate the product.
3. Close the control session.

Following is a simple example of the use of the most common Pickering drivers:

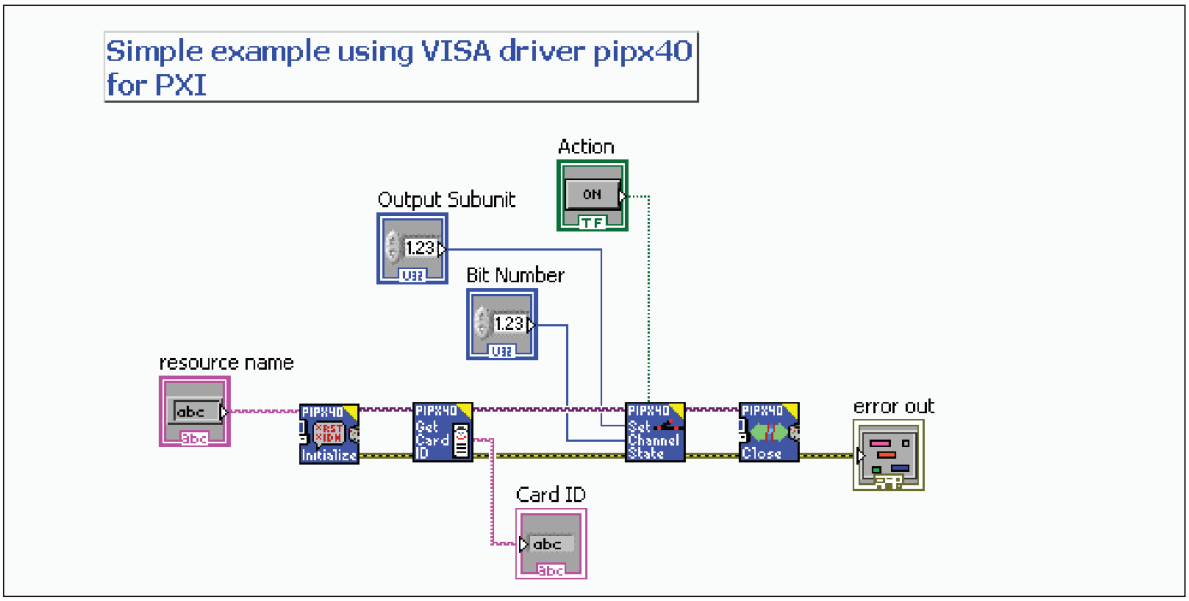


Figure 4.4 - Example Use of Pickering Drivers in LabVIEW

To download LabVIEW examples, please go to:

Using Pickering Drivers in LabVIEW

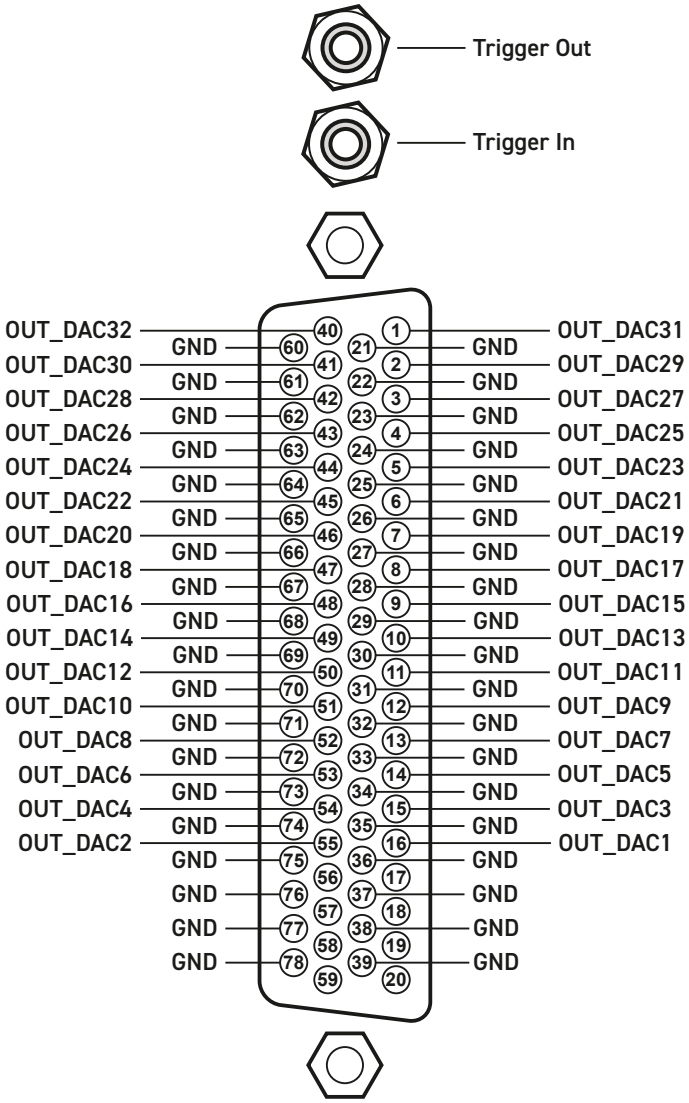


Figure 5.1 - Multi-Channel Function Generator Module 41/43-625-001 Pinout:
78-pin D-type Male Connector & 2xSMB Coaxial Connectors
(Viewed From Front of Module)

Section 6 - Troubleshooting

Refer to the Warnings and Cautions at the front of this manual. 

Installation Problems

The Plug & Play functionality of Pickering switch cards generally ensures trouble-free installation.

If you do experience any installation problems you should first ensure that all cards are properly seated in their slots. Improperly mated cards may go undetected by the operating system, or may be detected as a card of an unknown type. They can also cause the computer to freeze at various stages in the boot sequence.

If your system employs a remote control interface you should check the integrity of all remote control interface links. When the system is powered up, and during Windows start-up, you should expect to see periodic activity on the remote control interface RX/TX (yellow) indicators, clearing to leave only the PWR/LNK (green) LEDs illuminated. The RX/TX indicators should show activity when you attempt to access a card.

Diagnostic Utility

The Pickering Diagnostic Utility (accessible through the Programs>>Pickering>>PXI Utilities menu) generates a diagnostic report of the system's PCI configuration, highlighting any potential configuration problems. Specific details of all installed Pickering switch cards are included. All the installed Pickering switch cards should be listed in the "Pilpxi information" section - if one or more cards is missing it may be possible to determine the reason by referring to the PCI configuration dump contained in the report, but interpretation of this information is far from straightforward, and the best course is to contact Pickering support: support@pickeringtest.com, if possible including a copy of the diagnostic report.

In the "VISA information" section, if VISA is not installed its absence will be reported. This does not affect operation using the Direct I/O driver, and is not a problem unless you wish to use VISA. VISA is a component of National Instruments LabWindows/CVI and LabVIEW, or is available as a standalone environment.

If VISA is present and is of a sufficiently recent version, the section "Pipx40 information" should present a listing similar to "Pilpxi information".

Please note that the Diagnostic Utility cannot access cards if they are currently opened by some other application, such as the Test Panels or Terminal Monitor.

Section 7 - Maintenance Information

Refer to the Warnings and Cautions at the front of this manual. 

Periodic Maintenance

This product and its corresponding chassis do not require any periodic maintenance.

General Cleaning

- Isolate the electrical power from the chassis and ensure that no user I/O signals are being applied.
- Wipe the product & chassis surfaces with a clean dry anti-static cloth only.

Software Update

For PXI modules operating in a PXI chassis, no module software updates are required. For the latest version of the driver please refer to our web site pickeringtest.com where links to our Software Download page will provide the latest version of the driver software for the various programming environments encountered.

For PXI modules which are supported in one of Pickering Interfaces' Modular LXI Chassis (such as the 60-102D and 60-103D) no module software update is required. If the module was introduced after the LXI chassis was manufactured the module may not be recognized, in this case the chassis firmware may need upgrading. This is a simple process which is described in the manual for the Modular LXI Chassis.

Section 7 - Maintenance Information

Verification & Adjustment Procedure

This section provides a procedure for verifying the DC calibration of the 41/43-625 and guidance on how to make adjustments. Under normal usage, the 41/43-625 should not require regular adjustment. When procedures require periodic verification checks of instruments in the system, Pickering Interfaces recommends testing every other year. In many applications, longer calibration intervals can be applied.

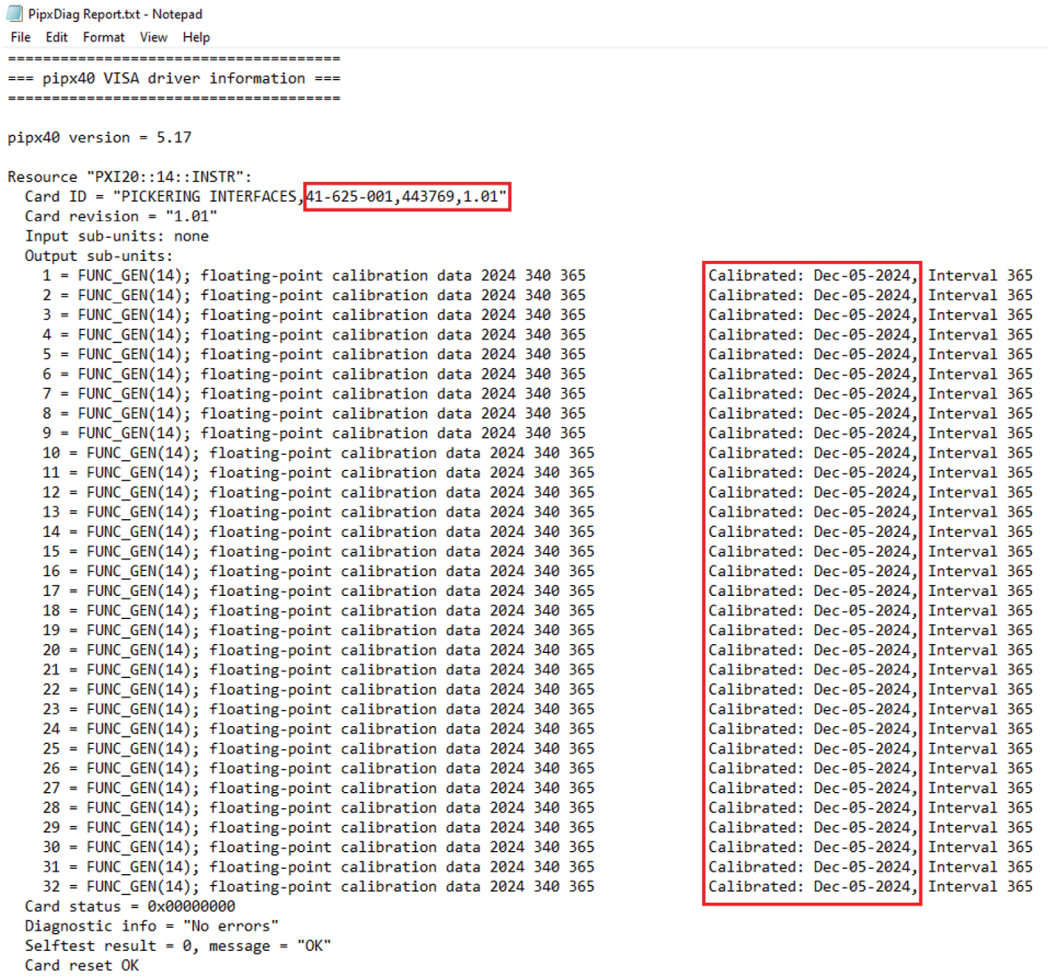
Verification

A verification check can be made periodically by connecting a DMM to the Front Panel (FP) channel port. The module has an internal date counter that shows the last calibration date. The customer has an option to read it via the following API function:

PIL_ReadCalDate

Alternatively, the calibration date can be obtained via Diagnostic Report (Check Pickering Interface website for how to generate a Diagnostic Report:
https://www.pickeringtest.com/kb/software-topics/how-to-create-pickering-diagnostics-report

Open the generated file and scroll down to the section with detected modules. Identify the module and read the calibration date.



PipxDiag Report.txt - Notepad

File Edit Format View Help

```
=====================================
=== pipx40 VISA driver information ===
=====================================

pipx40 version = 5.17

Resource "PXI20::14::INSTR":
  Card ID = "PICKERING INTERFACES,41-625-001,443769,1.01"
  Card revision = "1.01"
  Input sub-units: none
  Output sub-units:
    1 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
    2 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
    3 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
    4 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
    5 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
    6 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
    7 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
    8 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
    9 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   10 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   11 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   12 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   13 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   14 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   15 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   16 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   17 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   18 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   19 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   20 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   21 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   22 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   23 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   24 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   25 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   26 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   27 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   28 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   29 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   30 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   31 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
   32 = FUNC_GEN(14); floating-point calibration data 2024 340 365        Calibrated: Dec-05-2024, Interval 365
  Card status = 0x00000000
  Diagnostic info = "No errors"
  Selftest result = 0, message = "OK"
  Card reset OK
```

Figure 7.1 – An Example of the 41/43-625 Diagnostic Report

Verification Procedure

The verification procedure describes a set of measurements to check that the module performs to specification. The following equipment is recommended to perform the test:

- A DMM with a 6.5 digit accuracy or higher. Example: Keysight 3446XA or 3447XA.
- A PXI/PXIe Matrix to separately connect all channels of 41/43-625 to the DMM. Example: Pickering 2A 2-Pole 32x4 Matrix Module 40-519B-002. This will allow automating the connections to all channels, this can be done manually or by other options if desired.
- A cable assembly to connect the 41/43-625, PXI/PXIe Matrix and the DMM. This can be constructed by the user, or the user can purchase a ready-made cable from Pickering Interfaces (part number 40-975-625-0.3M) suitable for connecting to the FP connector.

The connection methods for verifying the 41/43-625 are described in Figure 7.2.

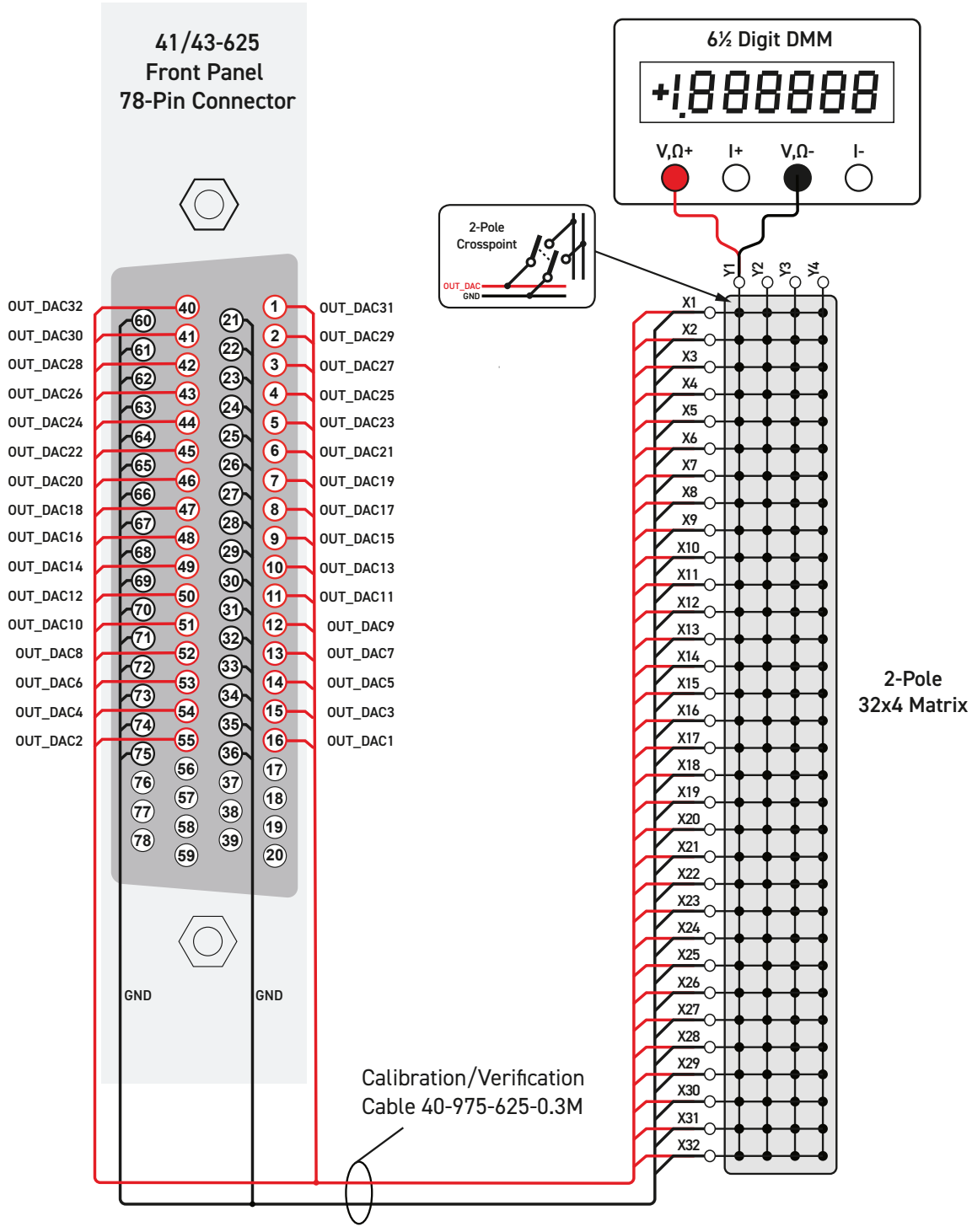


Figure 7.2 – 41/43-625 Verification Setup

Setup the Verification

- Connect each (positive and negative) 41/43-625 inputs to each 2-Pole 'X' port of the (2-Pole 32x4) Matrix^1, respectively.
- Connect the DMM to the 'Y1' port of the (2-Pole 32x4) Matrix^1.

Verification Procedure (DC Output)

1. **Connect a 41/43-625 channel from FP to the input of the DMM** manually or through switching the (2-Pole 32x4) Matrix (if the Matrix is applied).
2. **Click a "Channel" tab to select an output channel on Pickering GSFP**. The selected channel should be same as the physical channel connecting to the DMM.
3. **Select "DC" for waveform type on Pickering GSFP.**
4. **Change the "Coarse" on Pickering GSFP for setting the DC output.** See table 7.1 for a list of suggested voltages for DC verification.
5. **Click "Initiate Generation" button on Pickering GSFP to enable DC output from FP.**
6. **Set up the DMM to apply an input filter, a typical filter is 200kHz low pass.**
7. **Measure the DC voltage from the DMM to 3 significant digits right of the decimal point. Check if the voltage reading is stable (+/- 2mV) and within the specification.** Record the voltage reading and fill in the verification form (See table 7.1)
8. **Click "About Generation" button on Pickering GSFP to disable DC output from FP.**
9. **Repeat the steps 4-7 to set and verify a different DC output.**
10. **Repeat all the above steps verify all channels separately.**

Notes:

The PXI/PXIe Matrix is optional but recommended. User can connect the DMM to each input channel of 41/43-625 separately by switching the related cross points of the Matrix. Otherwise, user needs to directly connect/disconnect the DMM to each output channel of the 41/43-625 in turn.

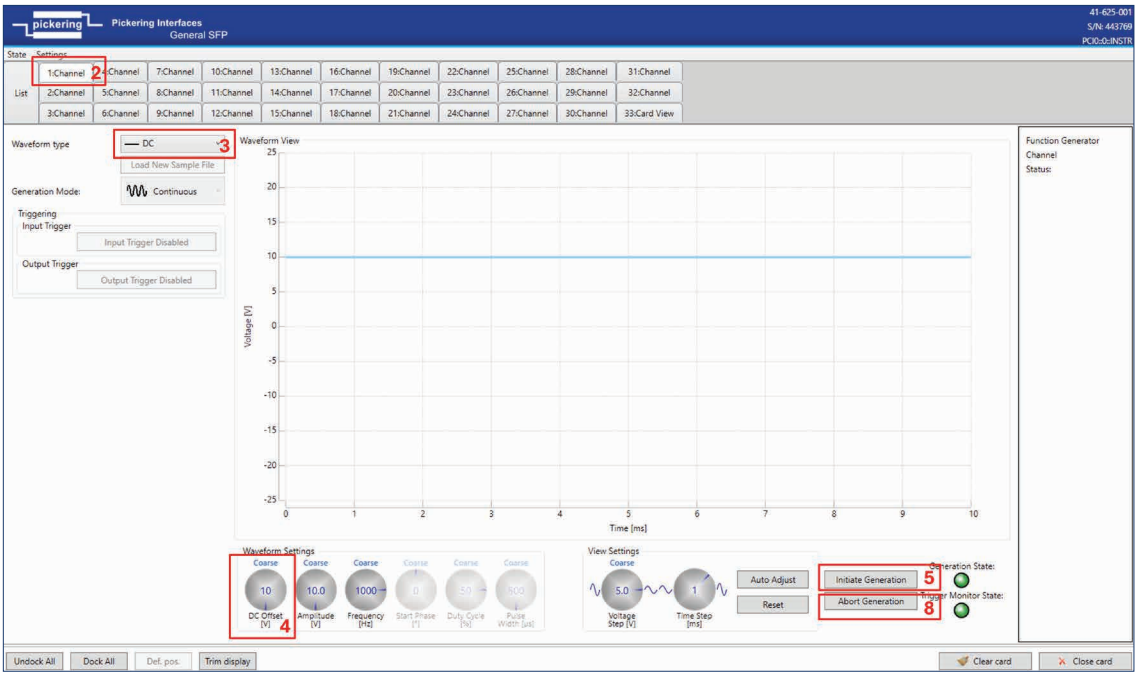


Figure 7.3 – An example of Pickering GSFP configuration (steps 2-5 and step 7 in DC output verification procedures) when testing 10V DC output of channel 1

Adjustment Procedure

The procedures of the 41/43-625 adjustment are provided through a separated manual of alignment. It is also recommended to return the 41/43-625 to Pickering Interface for adjustment.

Verification Test Results

For convenience, Table 7.1 on the following page are provided to record voltage measurement for each model's channel.

Table 7.1 – DC Output Verification Test Results (Accuracy: +/-0.1% of Settings +/-25mV)

Serial Number:	
Channel Number	
Date of Test	
Operator's Name	
Test Equipment Used	

Test DC Voltage (V)	Measured Value (V)	Spec -(V)	Spec + (V)	Result (delete as appropriate)
25		24.950	25.028	Pass/Fail
20		19.955	20.027	Pass/Fail
15		14.960	15.027	Pass/Fail
10		9.965	10.026	Pass/Fail
5		4.970	5.026	Pass/Fail
0		-0.025	0.025	Pass/Fail
-5		-5.030	-4.970	Pass/Fail
-10		-10.035	-9.965	Pass/Fail
-15		-15.040	-14.960	Pass/Fail
-20		-20.045	-19.955	Pass/Fail
-25		-25.050	-24.950	Pass/Fail