VeriStand Custom Device Handbook

Release 1.0.0

NI

Dec 15, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Introduction to Custom Devices

VeriStand is an open software environment you can use to configure real-time testing applications, including hardware-in-the-loop (HIL) systems.

With VeriStand, you can complete the following objectives.

- Configure real-time input/output (I/O), stimulus profiles, data logging, alarming, and other tasks.
- Implement control algorithms or system simulations by importing models from a variety of software environments.
- Build test system interfaces quickly with a run-time editable user interface complete with ready-to-use tools.

For more information on VeriStand, refer to the NI Developer Zone tutorial What is NI VeriStand?

You can customize and extend the VeriStand environment with LabVIEW to meet application requirements. This document provides the background, design decisions, and technical information required to understand and develop custom devices in VeriStand.

Before you begin creating a custom device, you must understand the VeriStand Engine. For more information on the VeriStand Engine refer to the VeriStand Manual.

1.1.1 What is a Custom Device?

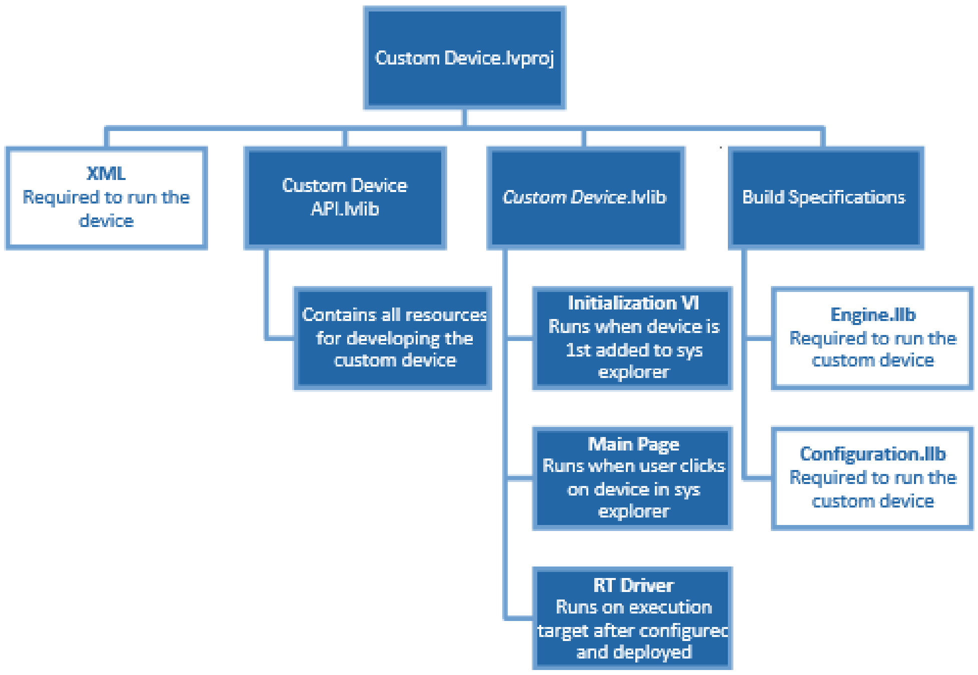
While VeriStand provides most of the functionality required by a real-time testing application, the environment can be customized to meet application requirements.

Custom devices are one way to extend VeriStand. For more ways to customize NI VeriStand, refer to the NI Developer Zone tutorial Using LabVIEW and Other Software Environments with NI VeriStand.

Developers can use custom devices to dictate how VeriStand executes. Any LabVIEW callable code can be made into a custom device. Custom devices allow customization to the operator interface within System Explorer.

Custom devices can display many different configuration experiences. This include simple controls on a VI front panel, pop-up windows and silent routines to scrape the configuration from a database.

A custom device typically consists of an XML file and two VI libraries. The following chart displays the organizational structure of a custom device LabVIEW project.


Custom Device.lvproj
XML Required to run the device
Custom Device API.lvlib
Custom Device.lvlib
Build Specifications
Contains all resources for developing the custom device
Initialization VI Runs when device is 1st added to sys explorer
Main Page Runs when user clicks on device in sys explorer
RT Driver Runs on execution target after configured and deployed
Engine.llb Required to run the custom device
Configuration.llb Required to run the custom device

The XML file tells VeriStand how to load, display, use and deploy the device. The VI Libraries define the behavior of the device. One library is for configuration and the other is for the engine.

Custom devices can be created by NI, 3rd parties, and in-house developers. The developer builds the configuration and engine library, and the XML file from Source Distributions in LabVIEW.

Most custom devices begin as a LabVIEW template project. The latest niveristand-custom-device-wizard release scripts the template project based on user inputs. You can then modify the template project to fulfill the requirements of the custom device.

A LabVIEW project is needed to build a custom device, but only the configuration library, engine library and XML file are required to use the custom device in VeriStand.

After obtaining (or building himself) the custom device's libraries, the operator places them in the VeriStand `<CommonData>\Custom Devices` directory. This directory location varies with the host operating system.

1.1.2 Table of Directories and Aliases:

The following tables list paths to common VeriStand directories by operating system. The heading before each table indicates how NI documentation refers to the directory. For directories with aliases listed, the alias is the text that appears with a relative path in an API or XML file. This text defines the directory that the path is relative to.

<Common Data>	Alias: To Common Doc Dir
Windows	<Public Documents>\National Instruments\NI VeriStand <xxxx>

<Application Data>	Alias: To Application Data Dir
Windows	<Application Data>\National Instruments\VeriStand

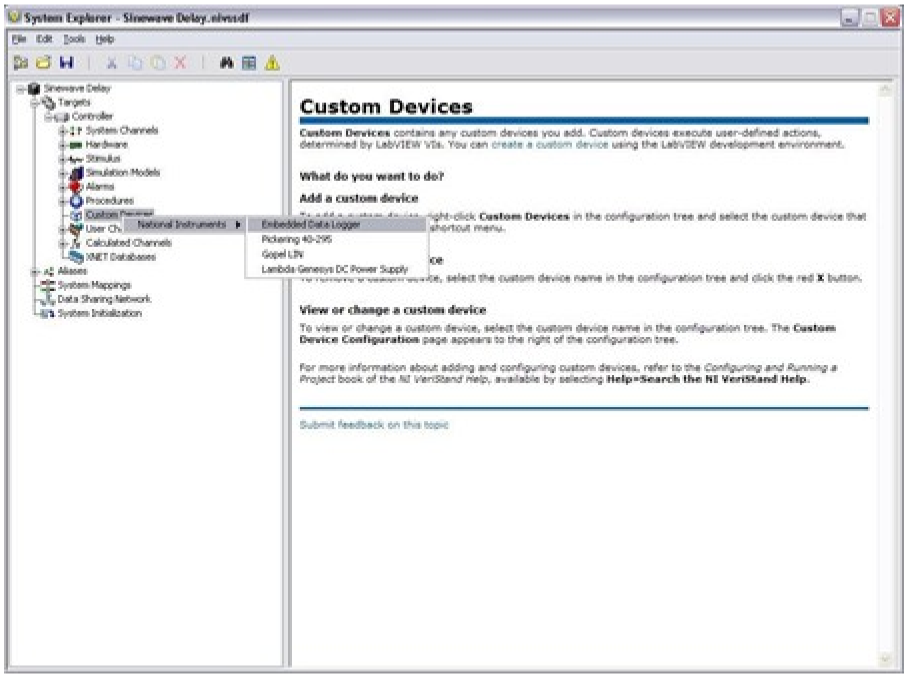
<Base>	Alias: To Base
Windows	<Program Files>\National Instruments\VeriStand <xxxx>

<Custom Device Engine Destination>	Alias: To Base
Linux	c:\ni-rt\NIVeriStand\Custom Devices\<custom device name>\

Note: <xxxx> is the VeriStand version number.

VeriStand parses `Common Data\Custom Devices` for custom device XML files when it first launches. You must restart VeriStand to recognize newly added or modified custom device XML files.

Add the custom device to the system definition in the configuration tree by navigating to **System Definition** » **Targets** » **Controller** and right-clicking **Custom Devices**.



Custom Devices

Custom Devices contains any custom devices you add. Custom devices execute user-defined actions, determined by LabVIEW VIs. You can create a custom device using the LabVIEW development environment.

What do you want to do?

Add a custom device

[illegible] right-click **Custom Devices** in the configuration tree and select the custom device that [illegible] shortcut menu.

[illegible]ce

[illegible]e, select the custom device name in the configuration tree and click the red **X** button.

View or change a custom device

To view or change a custom device, select the custom device name in the configuration tree. The **Custom Device Configuration** page appears to the right of the configuration tree.

For more information about adding and configuring custom devices, refer to the *Configuring and Running a Project* book of the *NI VeriStand Help*, available by selecting **Help»Search the NI VeriStand Help**.

Submit feedback on this topic

Custom devices consist of three parts.

- Custom Device Framework
- Custom Code
- Custom Device XML File

1.1.3 Custom Device Framework

The custom device framework consists of type definitions, specifically named controls and indicators, template VIs and a LabVIEW API. Together these items form the rules, or framework, that allows any conforming VI to interact with VeriStand. There are several prebuilt types of custom devices. Almost any requirement can be accomplished by adding or modifying code in one of the prebuilt devices.

The prebuilt devices start with the niveristand-custom-device-wizard. The developer specifies the type of custom device before running the niveristand-custom-device-wizard. The wizard generates the LabVIEW Project for the new custom device. The exact resources in the project depend on the type of custom device selected.

The project is pre-populated with VIs, LabVIEW Libraries, an XML File, and two build specifications. These resources provide the framework upon which almost all custom devices are built.

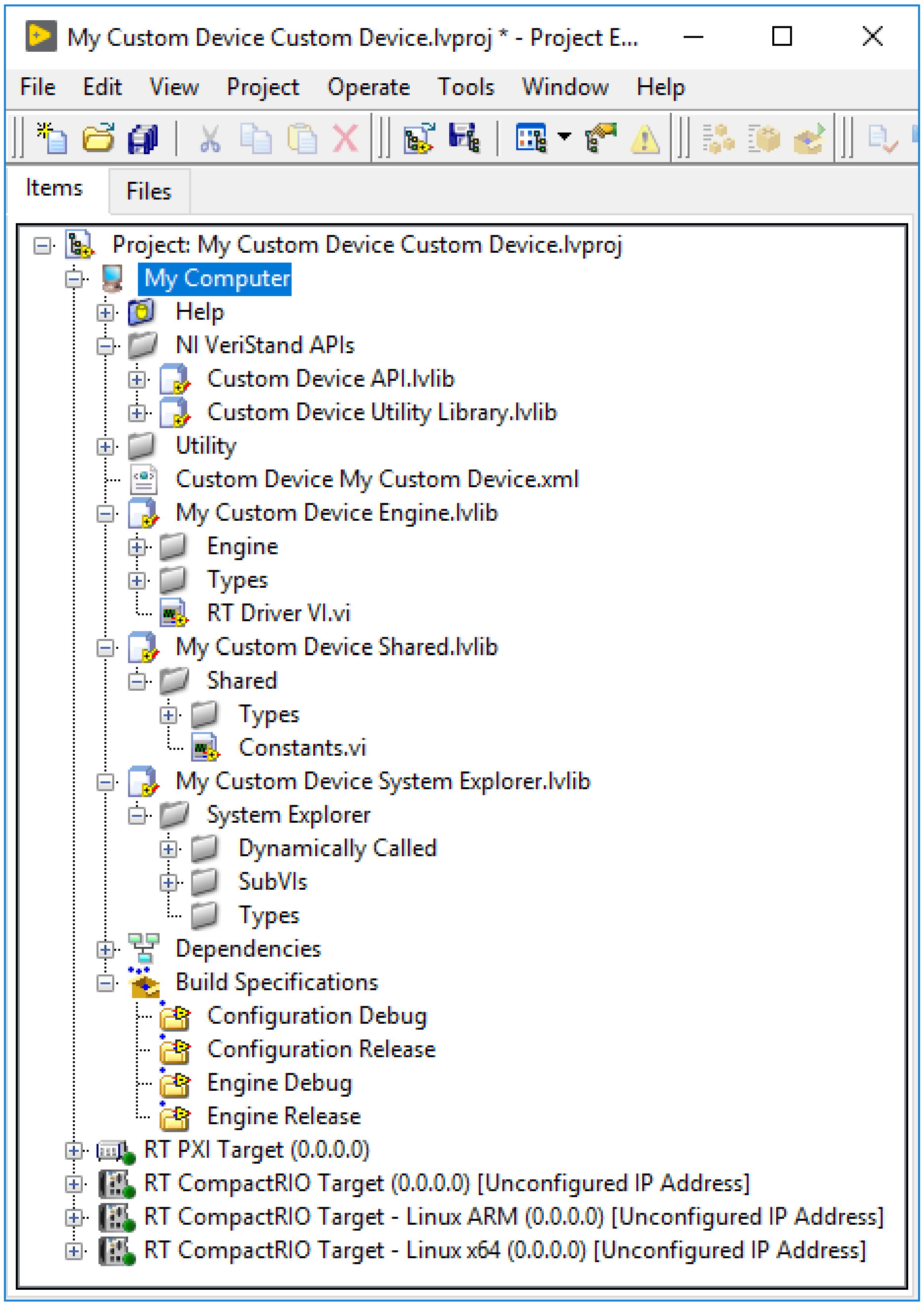
VeriStand evolved from NI Dynamic Test Software (NI-DTS). NI-DTS evolved from 3rd party intellectual property (IP) called EASE. The IP made basic provisions for add-on LabVIEW code.

These provisions could be considered the first custom device framework on which several "custom devices" were built. If you find a custom device that does not fit the niveristand-custom-device-wizard framework, you may be operating an EASE based custom devices.

For each type of custom device in the LabVIEW source project, you will find the following.

- An NI VeriStand APIs virtual folder containing two VI libraries called *Custom Device API.lvlib* and *Custom Device Utility Library.lvlib*.
- A *<Custom Device Name>* VI library.
- A *<Custom Device Name> Shared.lvlib* VI library.
- A *<Custom Device Name> System Explorer.lvlib* VI library.

The following image displays a new custom device template project.



```
Project: My Custom Device Custom Device.lvproj
  My Computer
    Help
    NI VeriStand APIs
      Custom Device API.lvlib
      Custom Device Utility Library.lvlib
    Utility
    Custom Device My Custom Device.xml
    My Custom Device Engine.lvlib
      Engine
      Types
      RT Driver VI.vi
    My Custom Device Shared.lvlib
      Shared
        Types
        Constants.vi
    My Custom Device System Explorer.lvlib
      System Explorer
        Dynamically Called
        SubVIs
        Types
    Dependencies
    Build Specifications
      Configuration Debug
      Configuration Release
      Engine Debug
      Engine Release
  RT PXI Target (0.0.0.0)
  RT CompactRIO Target (0.0.0.0) [Unconfigured IP Address]
  RT CompactRIO Target - Linux ARM (0.0.0.0) [Unconfigured IP Address]
  RT CompactRIO Target - Linux x64 (0.0.0.0) [Unconfigured IP Address]
```

The Custom Device API library and Custom Device Utility Library contain most of the type definitions, template VIs and LabVIEW API needed to interact with VeriStand data and timing resources. They allow the VI to behave as a native task in the VeriStand Engine.

Note: Some of these VIs also appear on the LabVIEW palette in **NI VeriStand** » **Custom Device API**.

The API library contains the custom device's configuration and real-time engine VIs. These correspond to the configuration and engine VI libraries (LLBs). The front panel and block diagram of these VIs are populated with objects from the Custom Device API libraries.

1.1.4 Configuration

The custom device configuration defines how the operator adds and configures the custom device through a user interface (UI). The Custom Device Template Tool provides the Initialization VIs for configuration purposes. You can add more VIs during development.

When a custom device VI's front panel is presented to the operator in System Explorer, that VI is called a page. Pages are a subset of the VIs that make up a custom device.

1.1.5 Initialization VI

The niveristand-custom-device-wizard adds the *Initialization VI.vi.* inside the Dinamically Called virtual folder of the <Custom Device Name> System Explorer library. This VI runs in the background when the custom device is first added to the system definition. The initialization page does not run again unless the operator removes and re-adds the custom device.

While you may rename certain objects in the custom device's LabVIEW Project, it's important to understand the ramifications of doing so. For example, the Initialization VI is referenced by name in the custom device XML file.

This file is generated when you first run the niveristand-custom-device-wizard. If you rename the Initialization VI after running the wizard, you'll need to manually change the path to the Initialization VI in the custom device XML file.

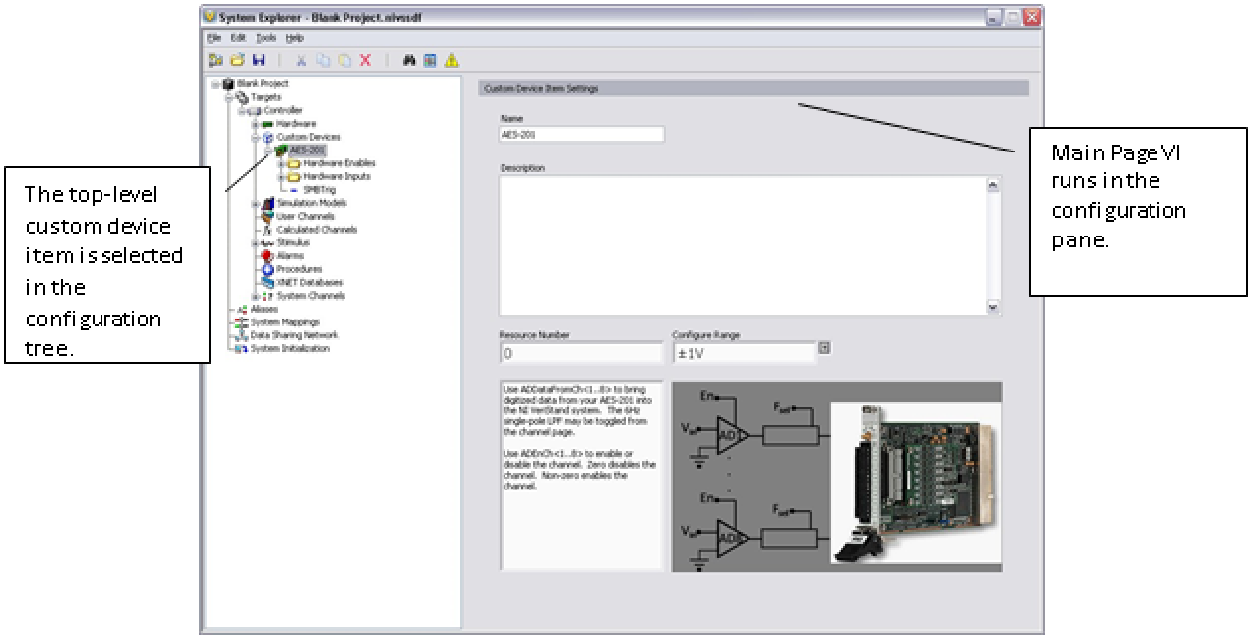
The Initialization Page runs each time a new instance of the same custom device is added to the system definition. VeriStand retains state information for each instance of a custom device in the system definition (.nivssdf) file.

State is defined by the value of each control, indicator, and property of the page. The system definition is human-readable XML, so you can open the file with a text editor.

Note: You can use the .NET API to programmatically modify the system definition.

1.1.6 Main Page

The niveristand-custom-device-wizard creates Main Page.vi inside Dynamically Called virtual folder of the *<Custom Device Name>* System Explorer library. After the custom device has been added to the system definition, the Main Page runs whenever the operator clicks on the custom device's top-level item in the System Explorer configuration tree. The following image displays the top-level item.



Name: AES-201

Description:

Resource Number: 0

Configure Range: ±1V

Use ADCDataFromCh<1..8> to bring digitized data from your AES-201 into the NI VeriStand system. The 6Hz single-pole LPF may be toggled from the channel page.

Use ADEnCh<1..8> to enable or disable the channel. Zero disables the channel. Non-zero enables the channel.

1.1.7 Engine

The niveristand-custom-device-wizard creates the *RT Driver.vi.* inside the <*Custom Device Name*> Engine library. This VI defines the behavior of the custom device on the Target.

The RT Driver VI runs on the Target regardless of the operating system. VeriStand deploys the engine when the operator runs the project from VeriStand or when the system definition is deployed using the VeriStand Execution API.

The engine runs after the custom device deployed to the execution host. You can usually add initialization, steady-state, and shutdown code to the engine template. There aren't any hard boundaries on what code you can put into the engine, but each additional code that is added can increase the size of the engine, and the time required to deploy your system.

Each of the five prebuilt custom devices has a different engine VI. Each engine VI executes at a different time with respect to other VeriStand components. The timing requirements of a custom device, and thus the type of device selected, are functions of when the device needs to execute with respect to other VeriStand Engine components.

Not all requirements can be satisfied by one of the five types of prebuilt custom devices. Some custom devices will require multiple engine libraries. For example, a device may need to support different real-time operating systems. The NI VeriStand – Set Custom Device Driver VI allows you to programmatically change the driver library for a custom device.

Some custom devices use the prebuilt template as a launching pad for multiple parallel processes or complex frameworks. For more information, refer to **Beyond the Template Frameworks**.

1.1.8 Custom Code

Custom code performs any functionality desired by the custom device developer. While the initialization and engine frameworks provide access to VeriStand data and timing resources, you must implement the code to meet specification.

For example, custom code can perform a single A/D conversion on a 3rd party digitizer. The framework provides the means for sending the digitized value to the rest of the VeriStand system so that it can be mapped to a channel or used in a stimulus profile.

1.1.9 Custom Device XML

Each custom device has an XML file that contains information used by VeriStand to load, configure, display, deploy and run the device. The basic information includes VI and dependency paths, page names, action items, menu items, and meta data for the various pages that make up the custom device.

The niveristand-custom-device-wizard generates an XML file in the template LabVIEW Project. Any properly formatted XML file will be parsed by VeriStand. After the XML file is created by the Custom Device Template Tool, all updates have to be manual.

The custom device XML file does not automatically synchronize with changes to the LabVIEW project. Also, the file does not automatically deploy. You must modify the XML file in the LabVIEW Project directory when making changes. Building the initialization specification overwrites the XML in the `<Common Data>\Custom Devices` folder.

The XML file alters the appearance and behavior of the custom device in System Explorer. For example, you can add a right-click menu to a custom device by adding tags to the custom device XML file.

VeriStand parses `<Common Data>` for custom devices when it launches. A corrupt custom device XML file can affect the overall VeriStand system. You should exercise care and make a backup of the custom device XML before modifying it.

1.2 When do you Need a Custom Device?

VeriStand provides the general functionality required by most real-time (RT) testing applications. However, NI has designed the VeriStand environment to be customizable to meet any additional application requirements.

The built-in components of a VeriStand Project are listed in the *VeriStand Help* topic VeriStand Environment. If these components do not fulfill your specifications, try one of the customization methods in Using NI VeriStand with Other Software Environments to Create Real-Time Test Applications.

There is a collection of VeriStand add-ons that have been gathered from internal NI developers and the VeriStand community. You should check for an existing custom device before building one.

There are three specifications that are best-suited for a custom device.

1. 3rd Party Hardware
2. Unsupported Measurement or Generation Mode
3. Feature

1.2.1 3rd Party Hardware

Determine if your hardware is natively supported by VeriStand. For more information, refer to the *VeriStand Help* topic NI Hardware Support. If your application requires other hardware, it can be implemented in a custom device.

Note: Several hardware vendors have created custom devices for their hardware. Contact the manufacturer before building a custom device.

1.2.2 Unsupported Measurement or Generation Mode

Determine if the required measurement or generation mode of your hardware is supported. For more information, refer to the *VeriStand Help* topic Adding and Configuring Hardware Devices.

If not supported, it can be implemented in a custom device. For example, single-point hardware-timed analog acquisition on NI-DAQ devices is supported in VeriStand. Continuous analog acquisition can be implemented as a custom device.

1.2.3 Feature

Determine if VeriStand's built-in functionality meets your needs. Most RT testing application features, such as host interface communication, data logging, and stimulus generation, are provided.

If a required feature is not built-in, it can be implemented by extending VeriStand. For a list of ways to customize VeriStand, refer to Using NI VeriStand with Other Software Environments to Create Real-Time Test Applications

Certain features are best implemented as custom devices. To determine when a custom device is the most appropriate mechanism to meet a specification, you should be familiar with all the customization methods available. A general rule is that custom devices implement features that require or use VeriStand channel data on the execution host.

For example, a TDMS File Viewer is built into the VeriStand Workspace. If you need to log VeriStand channels to TDMS without first sending it back to the Workspace (as with high-speed streaming), a custom device called the Embedded Data Logger fulfills this requirement.

If you need to display previous test results on the workspace while a new test is running, a custom workspace object may be more appropriate. For more information, refer to Creating Custom Workspace Objects for VeriStand.

1.2.4 Custom Device Risk Analysis

The open nature of VeriStand is a strong advantage over other RT/HIL testing solutions. You can take advantage of this extensibility by using custom devices written by other developers. Writing your own custom device incurs a set of manageable risks. This section provides a list of risks that should be considered before custom device development begins.

1.2.5 LabVIEW Application Development

Custom devices are written in LabVIEW. The framework generated by the NI VeriStand Custom Device Wizard is single-loop or an action-engine VI. This architecture may be suitable for simple custom devices. Advanced custom devices will require more complicated architecture.

A prerequisite for custom device development is thorough knowledge of LabVIEW programming and application architectures. This knowledge represents NI Certified LabVIEW Developer (CLD) level expertise. You can obtain this experience through NI's Training and Certification program by completing the LabVIEW Core 1, Core 2, and Core 3 courses.

VeriStand custom devices are typically not large LabVIEW applications. Custom devices are designed to be modular, self-contained add-ons that add a specific functionality to VeriStand. While custom devices are typically developed by a single programmer, large application development best-practices may still apply. For more information, refer to the *LabVIEW Help* topic Best Practices for Large Application Development.

1.2.6 LabVIEW RT Application Development

Custom devices are typically designed to execute on RT systems. This allows the operator to perform deterministic HIL and RT test procedures.

Programming for an RT system requires knowledge of real-time operating systems (RTOS) and specialized LabVIEW development techniques. This knowledge is typically obtained through NI's Training and Certification program by completing the Real-Time Application Development course and refined by working on several LabVIEW RT applications.

1.2.7 VeriStand Background

Familiarity with the VeriStand Engine is crucial to successful custom device development. Selecting the appropriate type in the Custom Device Template Tool is difficult without understanding each type. For more information, refer to the *VeriStand Help* and the Understanding the VeriStand Engine topic.

Experience with VeriStand from an operator's perspective is helpful. This experience enables you to build operator-friendly interfaces that conform to the standard look and feel of other VeriStand components. Familiarity with VeriStand allows the developer to build a complex system definition, which allows thorough testing and benchmarking.

1.2.8 Hardware Driver Development

The custom device must call a hardware or instrument driver to support 3rd-party hardware. All NI hardware comes with a LabVIEW Application Program Interface (API) that can be used in the custom device. However, just because a LabVIEW API exists does not guarantee the custom device can be easily implemented.

Consider the following points when evaluating the feasibility of a custom device for 3rd-party hardware.

1. Does an Instrument Driver exist? For more information, refer to Instrument Driver Network.
2. Is a hardware driver available?
3. Is the driver well documented?
4. Can the hardware requirement be met by passing LabVIEW double data type (DBLs) to and from the custom device during steady state operation?

VeriStand uses channels to pass data between different parts of the system, including to and from custom devices. All VeriStand channels are LabVIEW DBLs. For more information on LabVIEW data types, refer to the *LabVIEW Help* topic Floating Point Numbers.

If the hardware driver returns a vector, structure, or any non-DBL data, that information cannot be passed directly from the custom device to the rest of the VeriStand system. The developer is responsible for finding a way to pass data. For more information on available communication mechanisms, refer to the *LabVIEW Real-Time Module Help* topic Exploring Remote Communication Methods.

VeriStand also exposes its TCP pipe through dynamic event registration. This pipe may suite your remote communication requirements. for more information, refer to Custom Device Engine Events section.

1.2.9 Testing

A custom device is one part of a VeriStand system. The complete state of the operator's system is seldom known by the custom device developer. System state includes the following information.

1. Target and host computer specifications.
2. System definition components.
3. Computational intensity of the simulation models.
4. Required loop rates.
5. System health and resource utilization.

Ideally, the custom device is implemented to be minimally burdensome, extremely efficient, and easy to use. Depending on complexity, it may become necessary to test, debug, and optimize the code on systems representative of the operator's system.

Consider the following example. A custom device developer needs to benchmark a 3rd-party hardware custom device. They add the custom device to the VeriStand Engine Demo and deploy the system definition to a quad-core PXIe-8880 Controller. Adding the custom device to the system increased the target's CPU load by 10% per-core and RAM utilization by 120KB.

If the operator is deploying the same custom device to a single-core PXIe-8821 Controller, with an average CPU load of 60% because of a computationally intense model, it's unlikely the operator will achieve the same loop rate. Their system may be incapable of running the custom device at all.

Time to test, debug and optimize the code must be factored into the development timeline. If you are developing for a specific operator, test on a representative system. If you are developing for unknown systems, include the benchmarked system specifications and timing information with the custom device documentation.

1.3 Planning the Custom Device

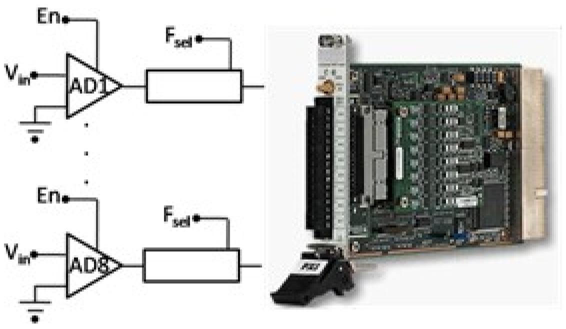
The most critical phase of custom device development is planning. Several VeriStand idiosyncrasies require more thorough planning than smaller stand-alone LabVIEW applications.

As the use-cases and flexibility of a custom device increases, so does the complexity of planning and implementing the device. The design tradeoff of this increase is a more robust device that requires less customization by the operator.

There are five areas to plan before you begin implementing.

1. Channels
2. Properties
3. Hierarchy
4. Pages
5. Device Type

In the following discussion, we will refer to a hypothetical 3rd party analog to digital (A/D) converter called the AES-201. The device was chosen to simplify the discussion.



Note: For an actual device, refer to the *VeriStand Manual* topics under Creating Custom Devices.

The AES-201 has eight 32-bit analog input (AI) channels. The device can digitize on ±1V or ±500mV. The card has a single software trigger line. Each channel has a software enable that is `ON` by default and a 6Hz low pass filter that is `OFF` by default.

A call to the hardware API makes a single A/D conversion on the specified channel and returns raw data. The range of the device cannot be changed after the device has been initialized.

1.3.1 Channels

Channels are used to exchange data between the custom device and the rest of the VeriStand system.

All channels are 64-bit floating point numbers. There is no built-in mechanism for other channel data types.

There are three common use cases for planning a custom device channel.

1. Data generated by the custom device after it is deployed. This data can be required by other parts of the VeriStand system.
2. Data originating elsewhere in the VeriStand system. This data can be consumed by the custom device after it is deployed.
3. Dynamic properties that can change after the device is deployed. These properties can be implemented in channels.

Note: Custom devices should be designed for generic use. Just because your customer does not use all channels and settings of the hardware does not mean you should hide anything from the operator.

Given these use cases, the AES-201 custom device should have one channel each for *ADDataFromCh<1..8>*. The digitized data is going to change while the device is running. The operator may need that data to be available to the rest of the VeriStand system. For example, operators often map data from hardware to simulation model inputs.

The operator may need to map the AES-201 software trigger to another channel in System Explorer, such as a calculated channel. The developer should create a channel for *SWTrig*.

The operator may need to disable a channel or toggle the input filter or the AES-201 while the device is running. The developer should plan an additional 16 channels for *FilterEnCh<1..8>* and *ADEnCh<1..8>*.

VeriStand channels are always LabVIEW DBLs. It may be easier to flatten data to DBL than it is to implement a background communication loop that passes native data types to the rest of the system. While the AES-201 LabVIEW API calls for Boolean data to enable the channel or filter, you can use a DBL channel with the assumption that *0 = False* and *!0 = True*.

Channels are created with the Add Custom Device Channel VI. A channel is either an Input or an Output. Channel type is determined by the custom device in the following situations.

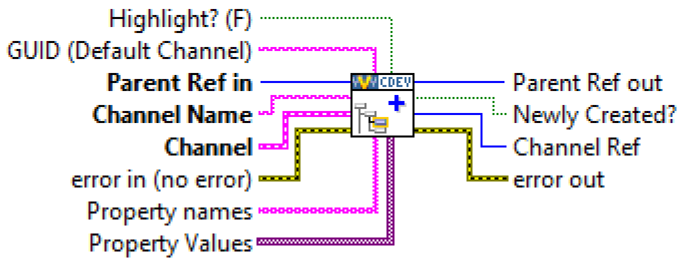
- If the custom device passes data to the rest of the VeriStand system, it requires an output channel.

- If the custom device receives data from the rest of the system, it requires an input channel. For example, the AES-201 may have eight output channels (ADDataFromCh<1..8>) and 17 input channels (ADEnCh<1..8>, FilterEnCh<1..8>, and SWTrig).

Once the custom device is loaded into VeriStand, the operator can map each input channel to a single data source. The operator can map each output channel to an arbitrary number of sinks. For example, you can map *ADDataFromCh1* to several simulation model inputs. *SWTrig* can be mapped to a user channel or a model output, but not both.

Add Custom Device Channel VI

Use the *NI VeriStand – Add Custom Device Channel VI* to add a channel to the device.



The device or device subsection is specified by *Parrent Ref in*. If the *Channel Name* you specify already exists, the VI overwrites the existing channel settings without affecting any custom properties. The VI can be called from any VI that runs on the host computer.

Other Useful Channel VIs

There are other VIs in the **NI VeriStand** » **Custom Device API** LabVIEW palette that operate on custom device channels.

- **Configuration** » **Get Custom Device Channel Data VI**
- **Configuration** » **Rename Custom Device Item VI**
- **Configuration** » **Remove Custom Device Item VI**
- **Channel Properties** » **Set Custom Device Channel Default Value VI**
- **Channel Properties** » **Set Custom Device Channel Faultability VI**
- **Channel Properties** » **Set Custom Device Channel Scalability VI**
- **Channel Properties** » **Set Custom Device Channel Type VI**
- **Channel Properties** » **Set Custom Device Channel Units VI**
- **Driver Functions** » **Get Custom Device Channel List VI**

In addition to these channel-specific VIs, any VI from the Item Properties palette can be used with a custom device channel.

1.3.2 Properties

Properties are used within the custom device to communicate state information.

Property names are case-sensitive strings. Unlike channels, property values can be any standard LabVIEW data type.

For one-time instances, you should use properties to transfer configuration and state information from the configuration to the engine. This transfer occurs when the system definition is deployed to the target.

After the system definition deploys, the engine can still read and write properties on the execution host. However, the engine cannot exchange properties with the host computer using the property VIs.

Because the AES-201 range cannot be changed after the card initializes, you should implement the AES-201 range setting as a property. The configuration routine on the host computer can set the *Range* property of the card based on operator input.

When the operator deploys the system definition, the engine can read the *Range* property. The engine can then call the hardware API to set the range.

After the AES-201 starts, the range cannot be changed. If the operator wants to change the range setting, they will need to use System Explorer to reconfigure the custom device and redeploy the system definition. The engine can still read and write the *Range* property, but the update is not reflected in System Explorer.

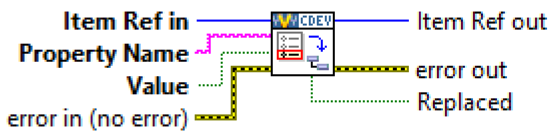
You can also implement the filter setting as a property. In System Explorer, the operator can enable or disable the filter on each channel page.

Doing so would allow the device to require eight fewer channels. However, the operator would no longer be able to toggle the input filter while the custom device is running.

Note: To illustrate several aspects of custom device development, we will implement the filter setting as a property.

Set Item Property VI

The *NI VeriStand – Set Item Property VI* can be called from any VI in the custom device.

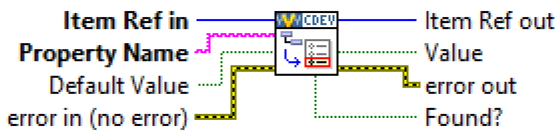


Properties can be applied to any channel or section. In addition to this VI, properties can be set when a channel or section is created by using *Property Names* and *Property Values*.

A property must be read from the item to which it was set. For example, if you set the *Filter_Enabled* property on the ADDataFromCh1 channel, you cannot read the value of the property directly from the parent section or any reference other than ADDataFromCh1. Properties do not inherit.

Get Item Property VI

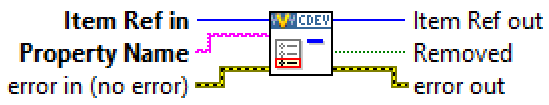
The *NI VeriStand - Get Item Property VI* returns the value of a specific item Property Name.



If the *Property Name* does not exist for the specified item, *Value* returns `Default Value`. Use the *Found?* output to check that the intended property name was found on the item.

Remove Item Property VI

The *NI VeriStand – Remove Item Property VI* removes the property name from an item.



The Get Item Property and Remove Item Property VIs may be called from any VI in the custom device.

Other Useful Property VIs

There are other VIs in the **NI VeriStand** » **Custom Device API** LabVIEW palette that operate on custom device properties.

- **Item Properties** » Get Item Description
- **Item Properties** » Get Item GUID
- **Item Properties** » Get Property Names List
- **Item Properties** » Set Item Description
- **Item Properties** » Set Item GUID
- **Device Properties** » Get Custom Device Decimation
- **Device Properties** » Get Custom Device Driver
- **Device Properties** » Get Custom Device Version
- **Device Properties** » Set Custom Device Decimation
- **Device Properties** » Set Custom Device Driver
- **Device Properties** » Set Custom Device Version
- **Device Properties** » Specify Custom Device Execution Mode

1.3.3 Custom Device Decimation

You can set the decimation for any type of custom device. However, decimation is handled differently for inline and asynchronous custom devices.

- An inline custom device is called when indicated by its decimation. For example, decimating an inline custom device by 4 causes the Primary Control Loop (PCL) to call the custom device on every fourth iteration. **Note:** The device must execute in a short enough time for the entire PCL to complete its iteration in addition to device execute time.
- Asynchronous devices have their channel FIFOs read on the Nth iteration of the PCL. N is the decimation rate of the asynchronous device.

1.3.4 Hierarchy

VeriStand allows each custom device to be presented as a hierarchy in System Explorer. A hierarchy allows developers to organize and present a custom device to the operator.

Within a Custom Device hierarchy, there are sections and channels. All items in a custom device configuration tree are either channels or sections. *Sections* provide a way to group items in a hierarchy.

The following rules apply to channels and sections.

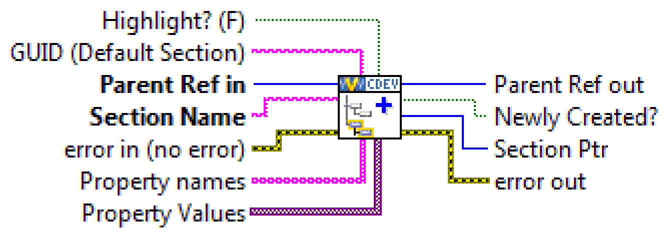
1. You cannot create additional levels of a custom device hierarchy from channels.
2. You cannot map sections to other items in VeriStand.
3. You cannot use sections to exchange data during run-time.

Use the NI VeriStand - Add Custom Device Section VI to create sections.

The default section glyph (icon) is a folder. You can change the glyph by modifying the custom device XML. A collection of glyphs that install with VeriStand is found in `<Application Data>\System Explorer\Glyphs`.

Add Custom Device Section VI

The *NI VeriStand – Add Custom Device Section VI* adds a section with the *Section Name* to the device specified by *Parent Ref in*.



If the name you specify already exists for that device, this VI updates only the GUID of that section without affecting any properties or child items.

This VI can be called from any VI that runs on the host computer. You build-up the custom device hierarchy by using the *Parent Ref out* and *Section Ptr* outputs.

Parent Reference is the level of the hierarchy that will contain the new section. Section Pointer is the reference to the new section, one level deeper in the hierarchy than the Parent Reference.

Hierarchy Examples

You should plan a custom device to use the minimum number of sections necessary to make the hierarchy well-organized, intuitive, and user friendly.

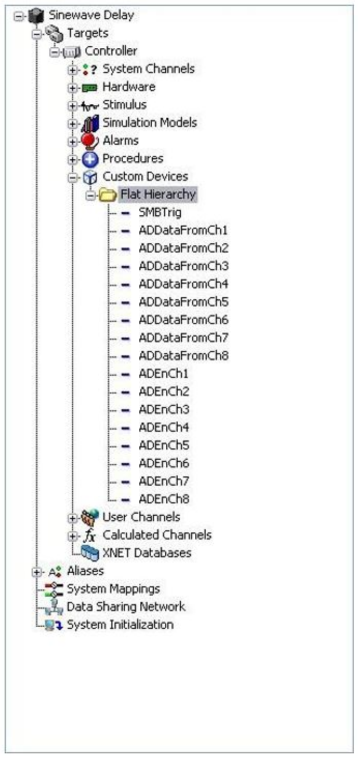
There are two types of hierarchies.

1. Flat
2. Nested

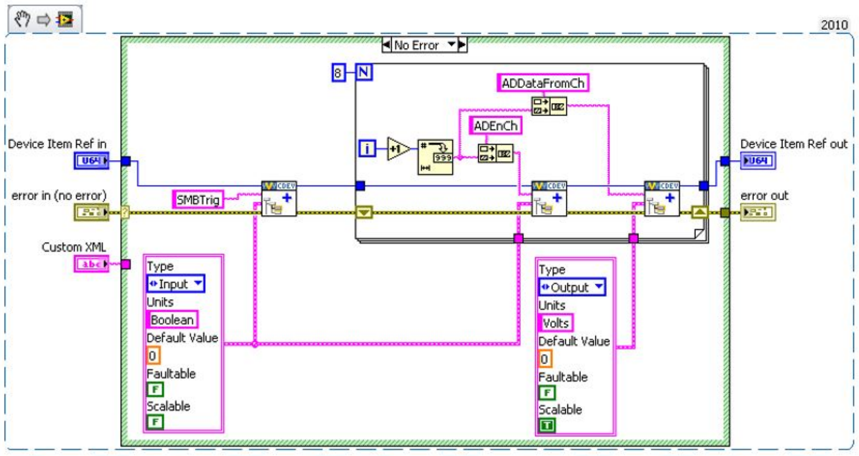
We will examine these hierarchy types in relation to AES-201 and discuss the advantages and disadvantages of each.

Flat Hierarchy

The following is an example of a flat, or single-level, hierarchy for the AES-201.



```
Sinewave Delay
  Targets
    Controller
      System Channels
      Hardware
      Stimulus
      Simulation Models
      Alarms
      Procedures
      Custom Devices
        Flat Hierarchy
          SMBTrig
          ADDataFromCh1
          ADDataFromCh2
          ADDataFromCh3
          ADDataFromCh4
          ADDataFromCh5
          ADDataFromCh6
          ADDataFromCh7
          ADDataFromCh8
          ADEnCh1
          ADEnCh2
          ADEnCh3
          ADEnCh4
          ADEnCh5
          ADEnCh6
          ADEnCh7
          ADEnCh8
      User Channels
      Calculated Channels
      XNET Databases
  Aliases
  System Mappings
  Data Sharing Network
  System Initialization
```



In this example, all of the channels are under the main section in the configuration tree. While it is easy for the operator to determine how many channels are available, the type of channel is unknown. Also, the channel function is only implied by the name.

A *flat hierarchy* is best suited for devices with a small number of channels that all perform the same function. The hierarchy is less useful for devices with many channels, or when channels perform different functions. For example, a custom device for a multifunction data acquisition board would be difficult to present in a flat hierarchy.

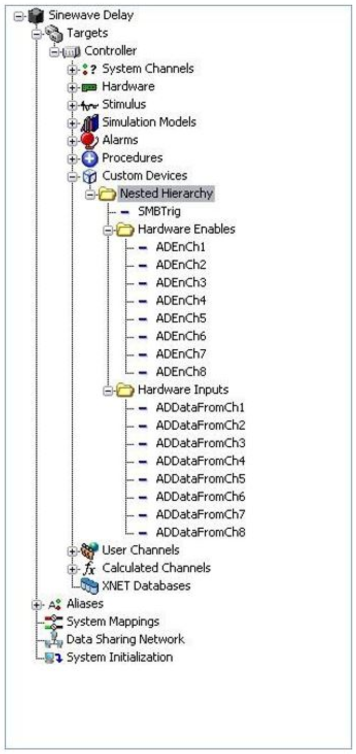
Notice that the same *Device Item Ref in* is used to create the *SWTrig*, *ADEnCh<1..8>*, and *ADDataFromCh<1..8>* channels. As a result, all of these channels appear at the same level of the hierarchy.

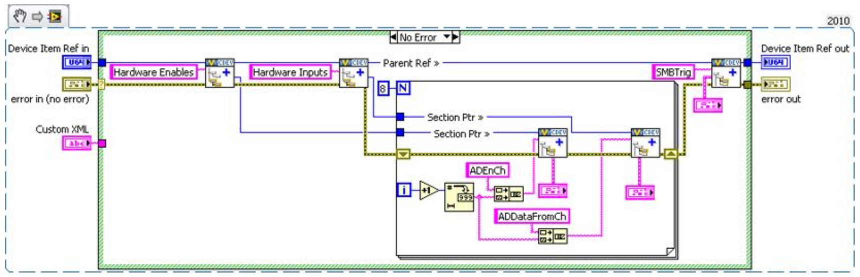
In the code, you should be able to identify the input and output channels. *SWTrig* and *ADEnCh<1..8>* are input channels because the custom device sinks data from them. *ADDataFromCh<1..8>* are output channels because they source data to the rest of VeriStand.

From an operator perspective, custom device inputs and outputs may seem backwards. Hardware inputs correspond to custom device outputs. The operator is not required to interact with the custom device source code. They will work in System Explorer. However, the channel direction should still make sense.

Nested Hierarchy

The following is an example of a nested hierarchy for the AES-201.





The channels have been organized into *Hardware Enables* and *Hardware Inputs* sections. This device is well-organized and fairly intuitive.

The *Section Ptr* outputs are used to create channels beneath the corresponding section in the initialization VI. The parent reference is also used to create the trigger channel at the same level as the two sections in the custom device hierarchy.

1.3.5 Pages

Pages are VIs that System Explorer displays in the configuration pane subpanel.

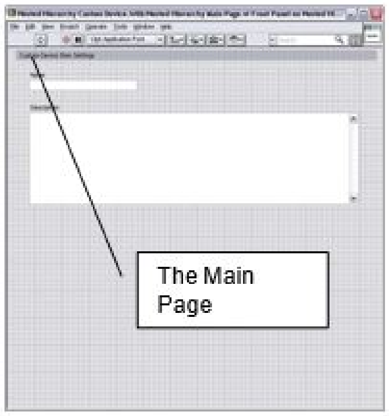
Subpanels are LabVIEW front pane containers that allow a VI to display the front panel of another VI. For more information, refer to the *LabVIEW Help* topic Container Controls and Indicators.

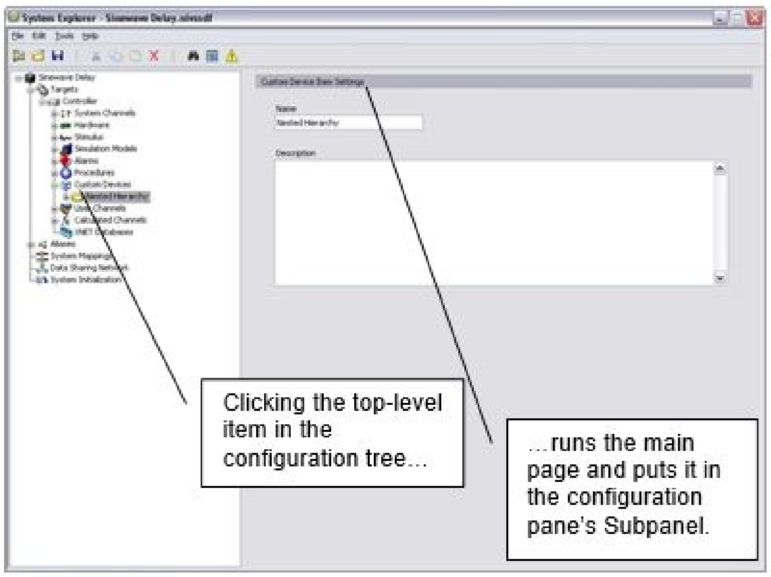
When you click an item in the configuration tree, a page displays in the Subpanel. Pages run on the host computer. They define the appearance and configuration experience of the custom device.

The niveristand-custom-device-wizard creates two configuration VIs by default.

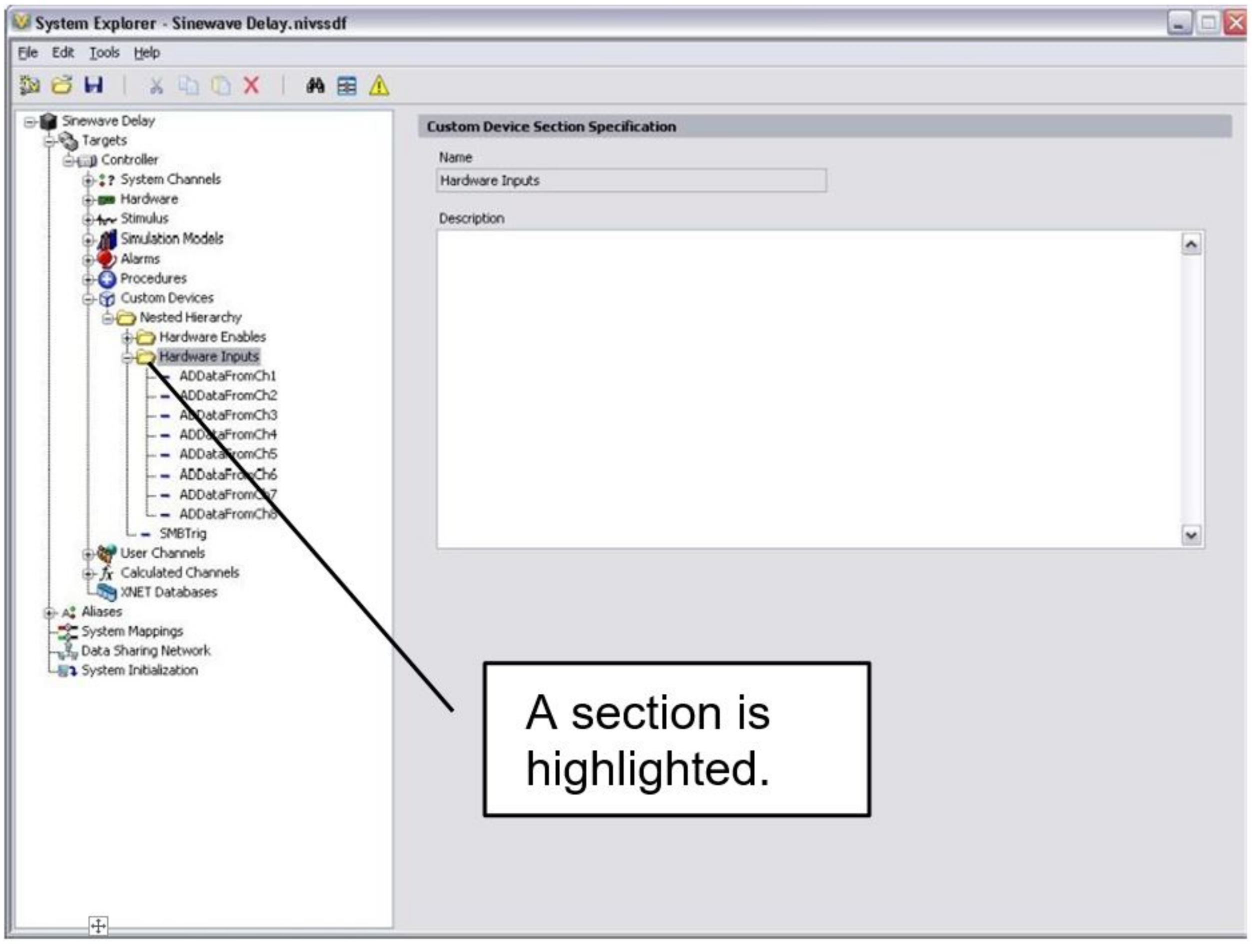
1. Initialization VI - A simple VI that does not populate in the Subpanel.
2. Main VI - A page.

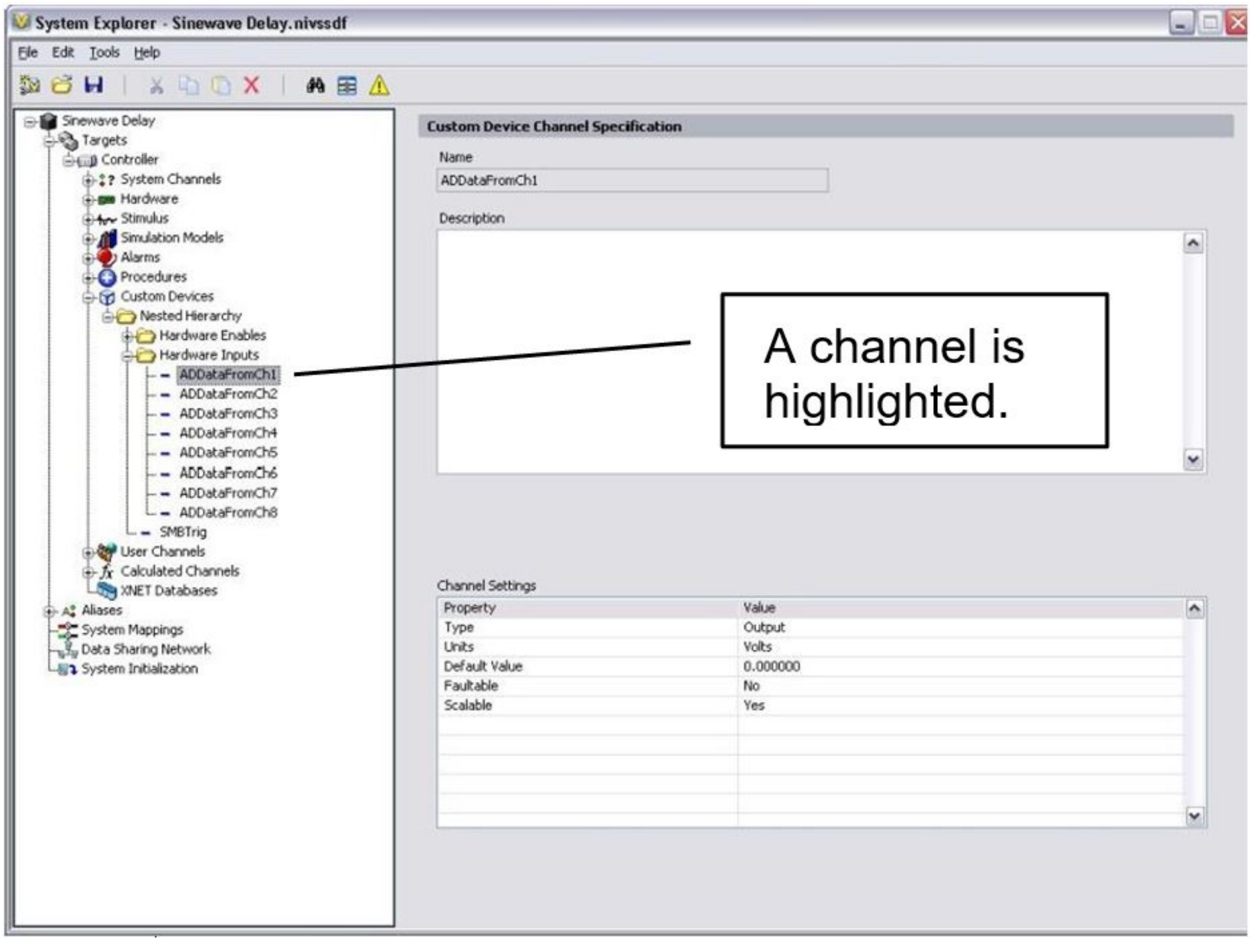
When you click on the top-most custom device item in the configuration tree, the <*Custom Device Name*> *Main Page* VI goes into the configuration pane and executes the block diagram.




Clicking the top-level item in the configuration tree…
…runs the main page and puts it in the configuration pane's Subpanel.

If the developer did not assign a custom page to a new section or channel, the default section or channel page is shown when the operator clicks on the item in the configuration tree.





Custom Device Channel Specification

Name: ADDataFromCh1

Description

Channel Settings

Property	Value
Type	Output
Units	Volts
Default Value	0.000000
Faultable	No
Scalable	Yes

The default pages allow the operator to set a description for the section or page. VeriStand retains this data in the System Definition (*.nivssdf*) file.

You cannot individually modify the block diagram or font panel of default pages. The *niveristand-custom-device-wizard* allows the developer to specify extra pages. Extra pages can be used to override the default page for an item.

When the developer creates an extra page and associates it with a section or channel, item's default page is overridden. You can individually modify the front panel and block diagram of extra pages. The block diagram executes when the operator clicks on the item in the configuration tree.

There are two rules for modifying custom device pages.

1. Do not change the front panel size. The front panel is loaded into a subpanel in the configuration pane. Changing the size may make the front panel unusable.
2. Do not change the names or connector pane associations of any terminal generated by the page template or niveristand-custom-device-wizard. VeriStand uses these objects to interface with the page. Changing these items may make the custom device unusable.

1.3.6 Extra Pages

Extra pages override default pages and allow you to customize the appearance and behavior of any item in the custom device hierarchy.

You should plan an extra page for each item in the custom device you want to customize differently. For example, to customize the page for each *ADDataFromCh* channel, you will need multiple extra pages. To customize all *ADDataFromCh* channels the same way, you will only need one extra page.

Note: VeriStand stores state data for each individual item in the custom device hierarchy in the system definition file.

The AES-201 may need five extra pages.

- A page for each section.
- A page for *ADDataFromCh<1..8>* channels.
- A page for *ADEnCh<1..8>* channels.
- A page for the *SWTrig* channel.
- A page for unforeseen needs.

You may not need all of these extra pages. It is better to have extra pages now rather than need more later.

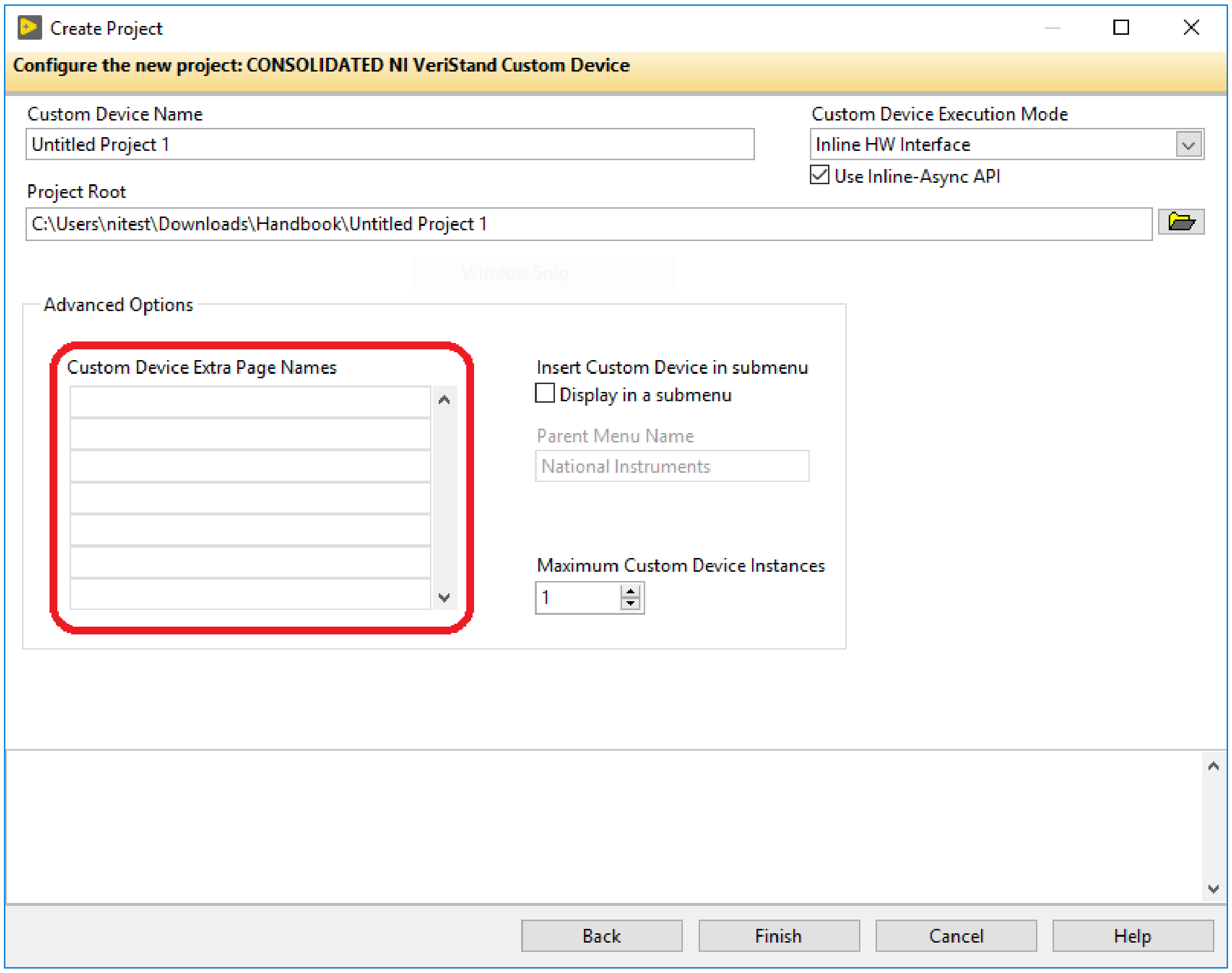
VeriStand requires four items to override a default page with an extra page.

1. Page
2. Globally Unique IDentifier (GUID)
3. XML Declaration
4. Build Specification

Page

A properly formed page VI must exist.

If you plan properly, you will be able to specify all the extra pages when you run the niveristand-custom-device-wizard. An extra page is created for each element in the **Custom Device Extra Page Names** control.



Create Project

Configure the new project: CONSOLIDATED NI VeriStand Custom Device

Custom Device Name: Untitled Project 1

Custom Device Execution Mode: Inline HW Interface

[✓] Use Inline-Async API

Project Root: C:\Users\nitest\Downloads\Handbook\Untitled Project 1

Advanced Options

Custom Device Extra Page Names

Insert Custom Device in submenu

[ ] Display in a submenu

Parent Menu Name: National Instruments

Maximum Custom Device Instances: 1

The *niveristand-custom-device-wizard* generates the page, GUID, and XML Declaration. The wizard then includes the page in the build specification. You will find the extra page template in the *Page Template.vit*. This file is located at `Custom Device API.lvlib\Templates\Subpanel Page VI`.

If you do not use the wizard to create extra pages, you must manually add and configure them. Manually adding extra pages to a custom device after running the wizard is difficult. Avoid this issue by creating a few extra pages beyond what you think is necessary.

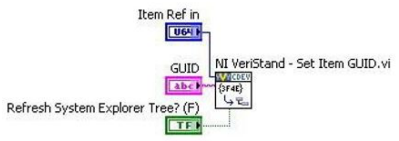
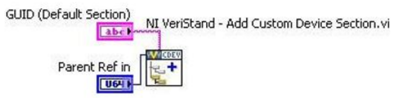
Note: Unused extra pages are not executed, but they do consume marginal space on disk.

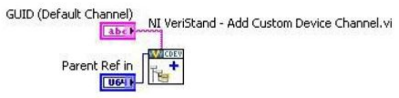
GUID

When you associate an extra page with a channel or section, you override the default page for that item.

This can be do done in two ways.

1. Specify the GUID when the item is created.
2. Set the item's GUID with the NI VeriStand - Set Item GUID VI. You can access this VI in LabVIEW by navigating to **NI VeriStand** » **Custom Device API** » **Configuration VIs** » **Item Properties**.


Item Ref in
GUID
NI VeriStand - Set Item GUID.vi
Refresh System Explorer Tree? (F)




The *niveristand-custom-device-wizard* generates a GUID for each extra page in the Custom Device Extra Page Names control.

XML Declaration

The custom device API associates a channel or section with a GUID while the custom device XML associates the GUID with the page VI.

The page and its GUID must be declared in the custom device XML `<Pages>` section within a `<Page>` schema.

```
<Page>
  <Name>
    <eng>Extra Page 1</eng>
    <loc>Extra Page 1</loc>
  </Name>
  <GUID>36481013-A447-6517-7D1C-FBB21CAE1E9F</GUID>
  <Glyph>
    <Type>To Application Data Dir</Type>
    <Path>System Explorer\Glyphs\default fpga category.png</Path>
  </Glyph>
  <Item2Launch>
    <Type>To Common Doc Dir</Type>
    <Path>Custom Devices\Extra Page Demo\Demo Configuration.llb\Extra Page 1.vi</Path>
  </Item2Launch>
</Page>
```

If the developer planned for the extra pages before running the *niveristand-custom-device-wizard*, the tool makes the appropriate entries in the custom device XML file for each extra page.

Build Specification

Extra pages are dynamically called VIs. Since they are not a part of the custom device VI hierarchy, they must be explicitly included in the Build Specification.

If the developer planned for the required extra pages before running the *niveristand-custom-device-wizard*, the wizard configures the build specifications to include the extra pages into the initialization library.

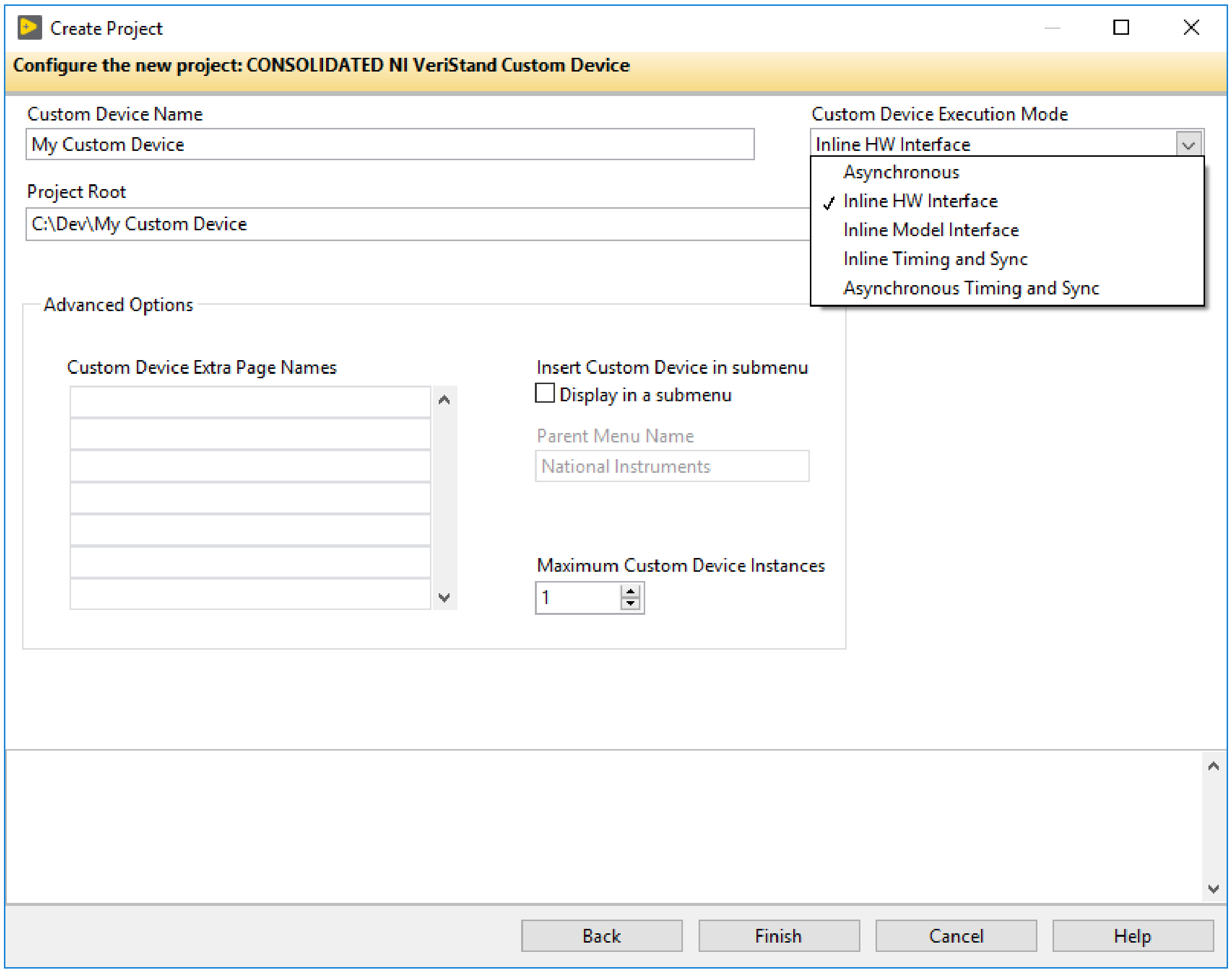
If a page must be added to the custom device after the tool runs, you must edit the configuration Build Specification to include the extra page. You must also include any dynamically called dependencies.

1.4 Custom Device Types

The custom device type refers to its execution mode. The mode defines how the device interacts with the VeriStand Engine.

The *VeriStand Engine* is the non-visible mechanism that controls system timing and communication between the Target and Host Computer. While deployed to the Target, all custom devices run inside the engine.

The niveristand-custom-device-wizard generates a new LabVIEW Project containing one of five device frameworks. The framework is determined by the **Custom Device Execution Mode**.


Create Project

Configure the new project: CONSOLIDATED NI VeriStand Custom Device

Custom Device Name: My Custom Device

Custom Device Execution Mode: Inline HW Interface

- Asynchronous
- ✓ Inline HW Interface
- Inline Model Interface
- Inline Timing and Sync
- Asynchronous Timing and Sync

Project Root: C:\Dev\My Custom Device

Advanced Options

Custom Device Extra Page Names

Insert Custom Device in submenu: ☐ Display in a submenu

Parent Menu Name: National Instruments

Maximum Custom Device Instances: 1

The selected mode determines when the device will run with respect to the other operations performed by the VeriStand Engine. There are five device frameworks available. Three of the frameworks are for custom devices, and the other two are for custom timing and synchronization devices.

Custom timing and synchronization devices are the same as regular custom devices, but they can be configured as the hardware synchronization master to drive RTSI0. For more information, refer to Real-Time System Integration (RTSI) and Configuration Explained.

Custom timing and synchronization devices are not covered in detail in this document. For more information about custom timing and synchronization devices, refer to the *VeriStand Help* topic Adding and Configuring Timing and Sync Devices. Multi- chassis synchronization may also be accomplished using built-in features. For more information, refer to Creating a Distributed System With NI VeriStand.

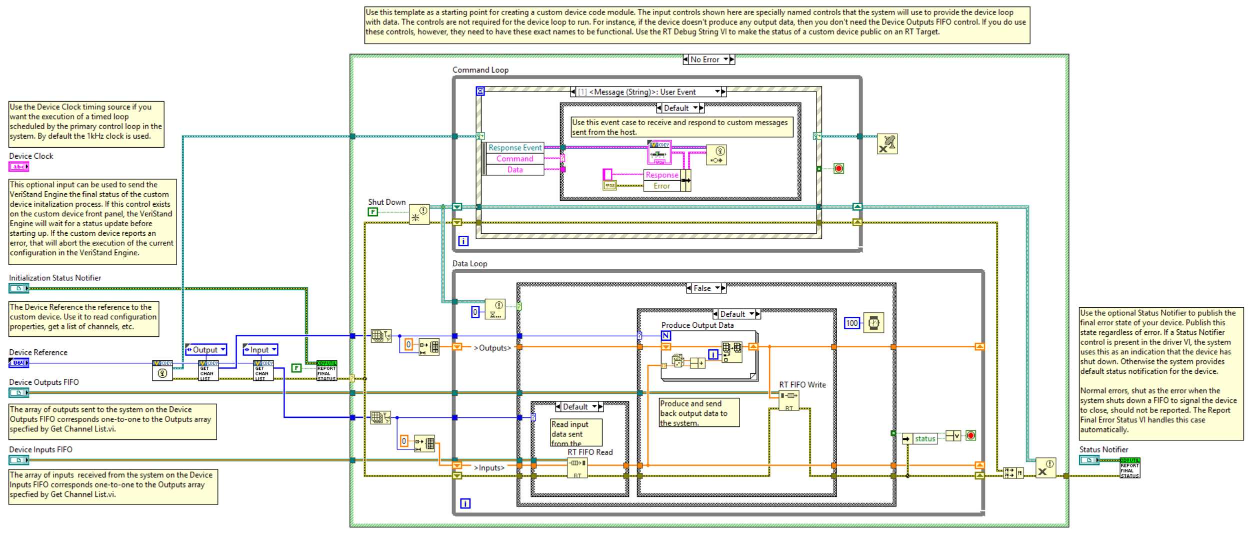
Two of the regular custom devices run in-line with the Primary Control Loop (PCL), the other runs in parallel with the PCL. A custom device is not limited to using just one type of framework. Some developers have built both in-line and parallel engines for a single custom device and allow the operator to select which mode to deploy.

Depending on your needs, you can alter the code within the framework. However, you must maintain the connector pane, controls, and indicators provided by the niveristand-custom-device-wizard. VeriStand uses these objects to interface with the custom device. If they are changed, the custom device will experience errors.

1.4.1 Asynchronous

The asynchronous custom device framework provides a simple, single-loop architecture. There are sections for initialization and cleanup before and after the loop.

Note: The asynchronous template provides a While Loop that can be exchanged for a Timed Loop.



Use this template as a starting point for creating a custom device code module. The input controls shown here are specially named controls that the system will use to provide the device loop with data. The controls are not required for the device loop to run. For instance, if the device doesn't produce any output data, then you don't need the Device Outputs FIFO control. If you do use these controls, however, they need to have these exact names to be functional. Use the RT Debug String VI to make the status of a custom device public on an RT Target.

Use the Device Clock timing source if you want the execution of a timed loop scheduled by the primary control loop in the system. By default the 1kHz clock is used.

This optional input can be used to send the VeriStand Engine the final status of the custom device initalization process. If this control exists on the custom device front panel, the VeriStand Engine will wait for a status update before starting up. If the custom device reports an error, that will abort the execution of the current configuration in the VeriStand Engine.

The Device Reference the reference to the custom device. Use it to read configuration properties, get a list of channels, etc.

The array of outputs sent to the system on the Device Outputs FIFO corresponds one-to-one to the Outputs array specfied by Get Channel List.vi.

The array of inputs received from the system on the Device Inputs FIFO corresponds one-to-one to the Outputs array specfied by Get Channel List.vi.

Use this event case to receive and respond to custom messages sent from the host.

Read input data sent from the

Produce and send back output data to the system.

Use the optional Status Notifier to publish the final error state of your device. Publish this state regardless of error. If a Status Notifier control is present in the driver VI, the system uses this as an indication that the device has shut down. Otherwise the system provides default status notification for the device.

Normal errors, shut as the error when the system shuts down a FIFO to signal the device to close, should not be reported. The Report Final Error Status VI handles this case automatically.

The loop runs in parallel to the PCL. If proper real-time development practices are adhered to, it is unlikely to block or slow the PCL. The rest of the VeriStand system will continue to execute as expected even if the asynchronous custom device is latent or stalls.

The loop can be synchronized to the PCL's timing source, making it pseudo-synchronous. This applies to asynchronous devices that use a Timed Loop. While Loops cannot be used for this purpose.

The benefit of an asynchronous custom device synchronized to the PCL is that it will not cause the PCL to be late if the device is late. VeriStand ticks the device clock for all Timed Loops that have **Use Device Clock** set to true.

The asynchronous device can run at a different rate than the PCL. You can define the rate using any execution timing method available in LabVIEW. The rate can iterate faster than the PCL or be a decimation of the PCL rate specified using the **NI VeriStand - Set Custom Device Decimation** VI. This VI can be found in LabVIEW by navigating to **NI VeriStand** » **Custom Device API** » **Configuration** » **Item Properties** » **Device Properties**.

The asynchronous template uses RT FIFOs, specifically Device Inputs FIFO and Device Outputs FIFO, to exchange channel data with the rest of VeriStand. Since the asynchronous device runs in parallel to the PCL and passes channel data through RT FIFOs, there is a minimum of one cycle delay from when data travels back and forth from the PCL to the custom device. These FIFOs correspond to those in the VeriStand Engine.

The asynchronous device is not guaranteed to execute at the same time as other components of the system. For example, the first iteration may execute before the PCL processes alarms.

The input controls are specially named controls that the system will use to provide the device loop with data. The controls are not required for the device loop to run. For example, if the device doesn't produce any output data, then you don't need the Device Outputs FIFO control. If you do need these controls, they must have these exact names to be functional.

The optional *notifier* status element is used to notify the engine of the last state of the custom device and to indicate the device has completed execution. If this control is not used, a default No Error value is returned to the system when the device finishes execution. This error state is not checked until the system shuts down. Use an output channel to send more immediate status values to the system.

The asynchronous framework includes VIs from the VeriStand Asynchronous Device Properties VIs subpalette.

1.4.2 Inline Hardware Interface

The inline hardware interface template is similar to a state machine architecture.

Note: Some developers will recognize it as an action-engine. For a discussion on action engines, refer to the NI Discussion Forums post Community Nugget 4/08/2007 Action Engines.

The PCL specifies the case to execute. An uninitialized **Feedback Node** is used for iterative data transfer.

There are five cases defined by the Operation enumerated control.

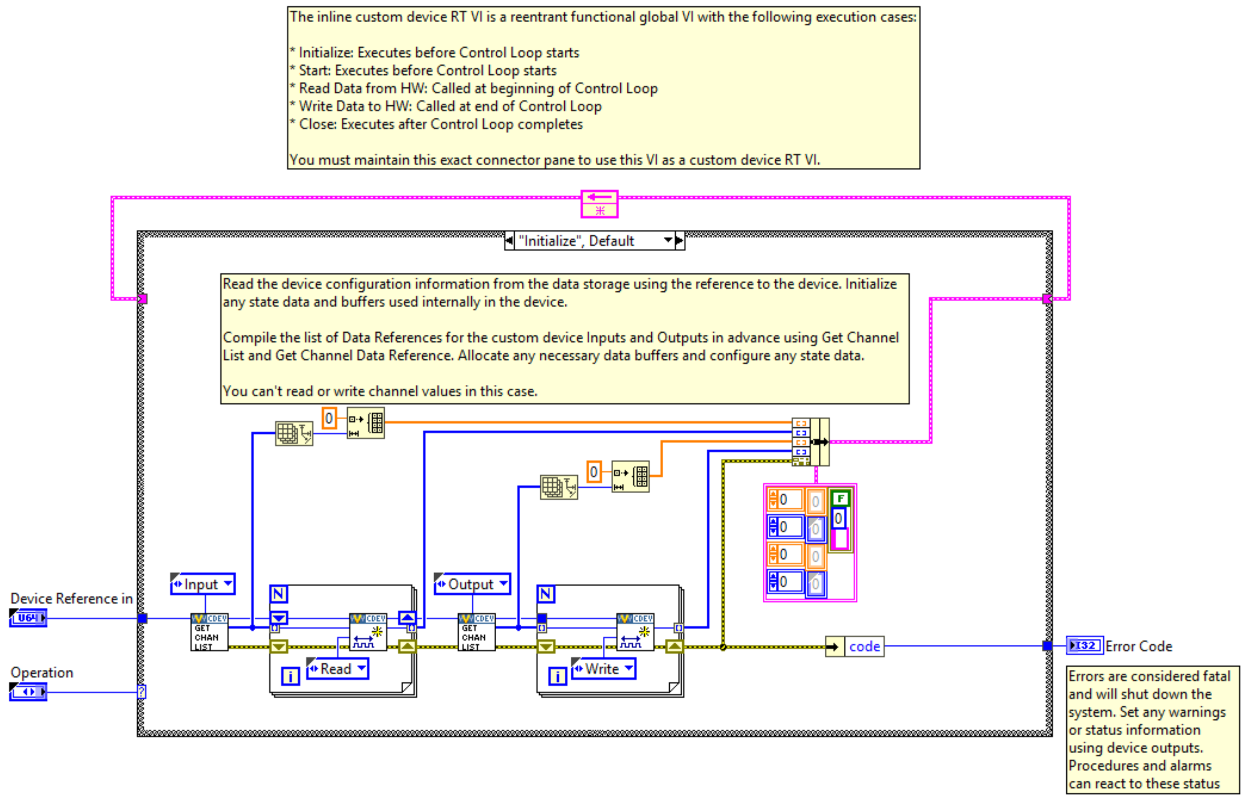
1. Initialize
2. Start
3. Read Data from Hardware
4. Write Data to Hardware
5. Close

This custom device runs in-line with the PCL, which calls each case at a specific time with respect to the other components in the VeriStand Engine. The PCL will not proceed until the custom device case has completed.

Initialize Case

The *Initialize* case executes before the PCL starts. Inside the case, you can use a device reference to extract configuration information from device properties. Initialize data and buffers are used internally in the device.

The framework compiles the list of data references for the custom device Inputs and Outputs. This is done in advance using VeriStand Data References VIs, specifically the NI VeriStand - Get Channel Data Reference VI.


The inline custom device RT VI is a reentrant functional global VI with the following execution cases:

* Initialize: Executes before Control Loop starts
* Start: Executes before Control Loop starts
* Read Data from HW: Called at beginning of Control Loop
* Write Data to HW: Called at end of Control Loop
* Close: Executes after Control Loop completes

You must maintain this exact connector pane to use this VI as a custom device RT VI.

Read the device configuration information from the data storage using the reference to the device. Initialize any state data and buffers used internally in the device.

Compile the list of Data References for the custom device Inputs and Outputs in advance using Get Channel List and Get Channel Data Reference. Allocate any necessary data buffers and configure any state data.

You can't read or write channel values in this case.

Errors are considered fatal and will shut down the system. Set any warnings or status information using device outputs. Procedures and alarms can react to these status

Note: Because the PCL has not started, channel values cannot be read or written in the Initialize case.

Start Case

The *Start* case executes after initialization but before the PCL runs. There is no difference between what code you can place in the Initialize and Start states.

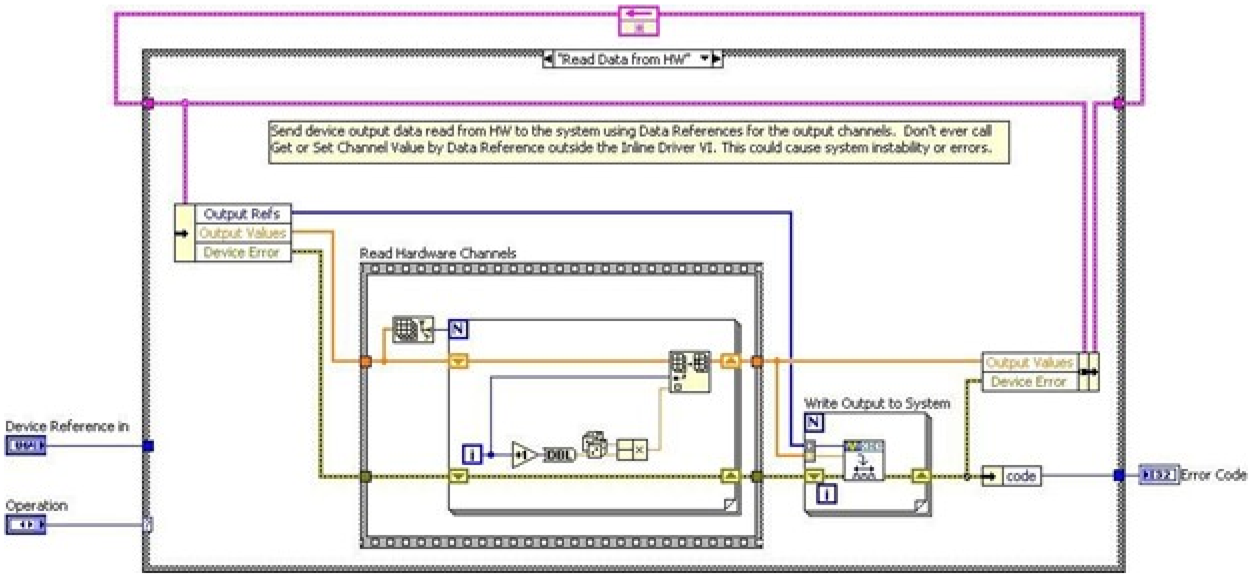
Note: Because the PCL has not started, channel values cannot be read or written in the Start case.

Read Data from HW Case

The *Read Data from HW* case executes at the beginning of the PCL, before other components, such as alarms, and procedures, execute.

After processing system mappings, the data obtained in this case is available to the other components of the system for the remainder of the PCL iteration. For a detailed timing diagram, refer to Outline of PCL Iteration.

The template contains a Flat Sequence frame named Read Hardware Channels. You can replace the code inside the frame with the API calls necessary to obtain data from a hardware API.

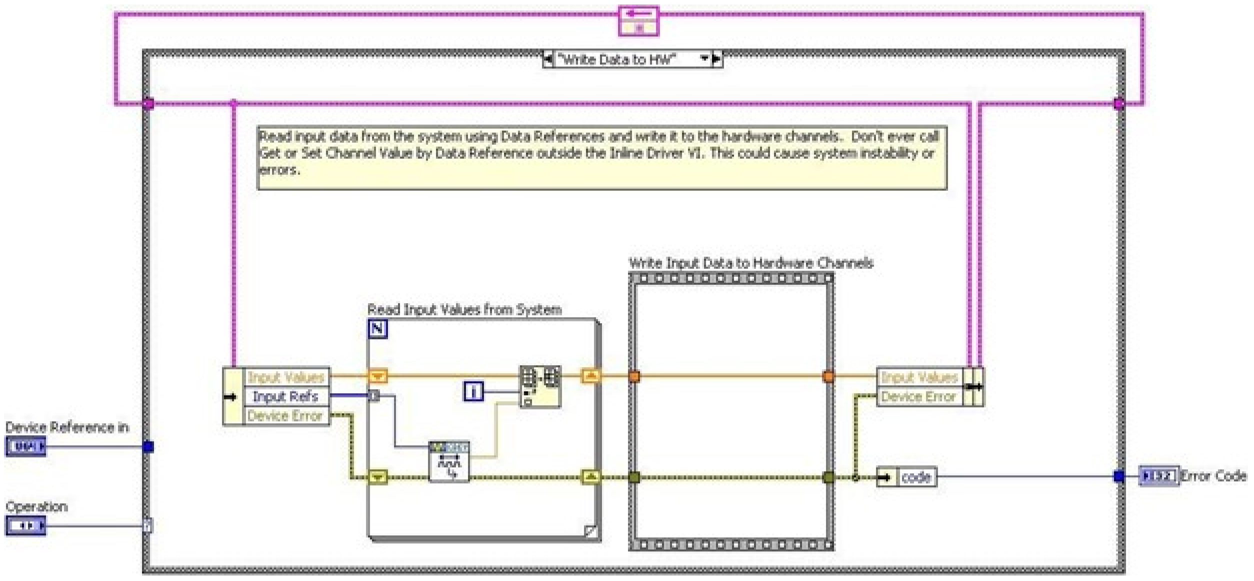


Send device output data read from HW to the system using Data References for the output channels. Don't ever call Get or Set Channel Value by Data Reference outside the Inline Driver VI. This could cause system instability or errors.

Write Data to HW Case

The *Write Data to HW* case executes at the end of the PCL, after the other components have executed.

The case contains a Flat Sequence frame named Write Input Data to Hardware Channels. You can replace the code inside the frame with the API calls necessary to send data to a hardware device.



Read input data from the system using Data References and write it to the hardware channels. Don't ever call Get or Set Channel Value by Data Reference outside the Inline Driver VI. This could cause system instability or errors.

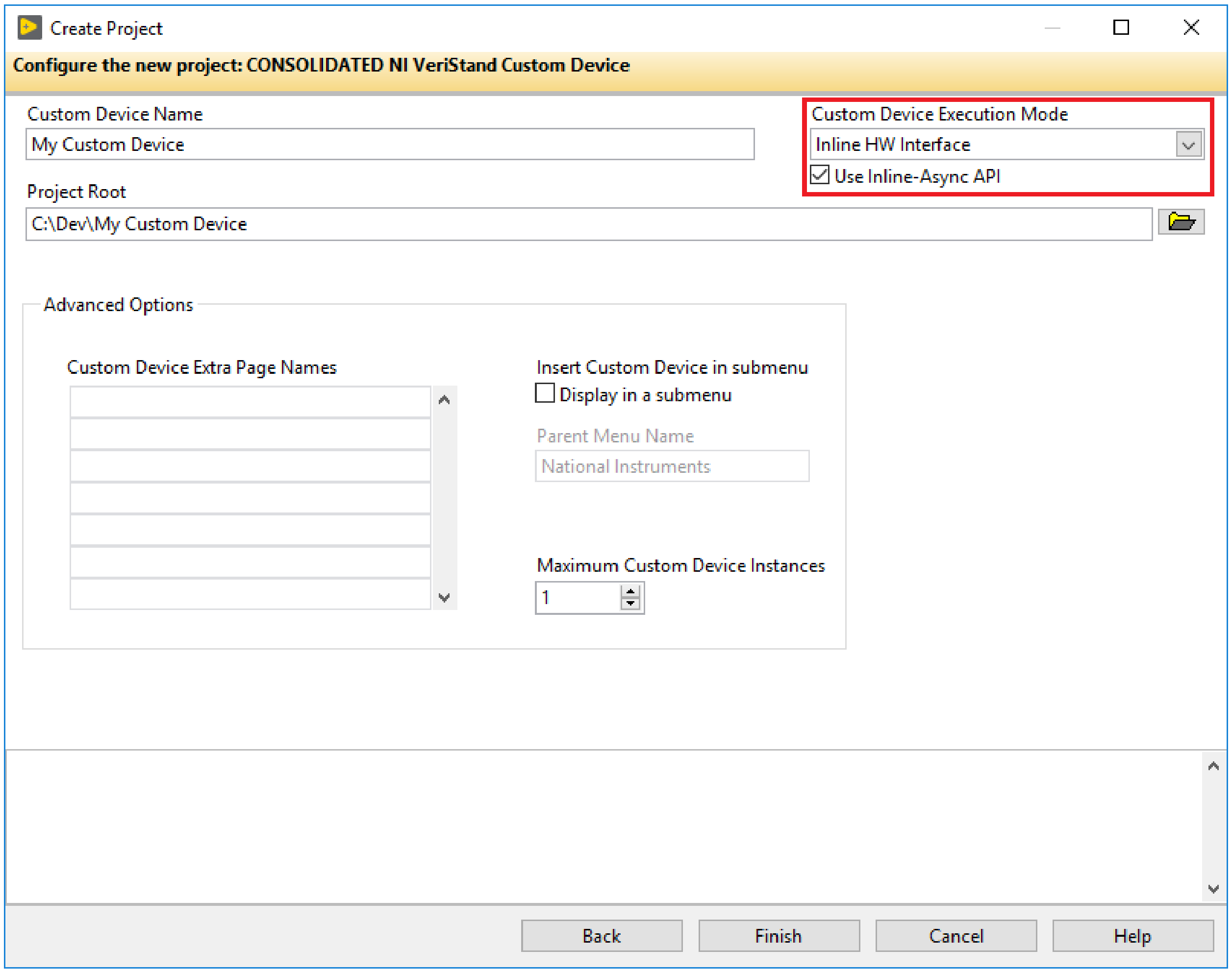
Close Case

The *Close* case executes after the PCL has finished executing. You should close references and release resources in this state.

Note: Because the PCL has terminated, channel values cannot be read or written in the Close case.

1.4.3 Inline-Async Hardware Interface

You can generate an inline-async custom device template from the niveristand-custom-device-wizard. When creating a project, set the execution mode to **Inline HW Interface** and enable **Use Inline-Async API**.



Create Project

Configure the new project: CONSOLIDATED NI VeriStand Custom Device

Custom Device Name: My Custom Device

Custom Device Execution Mode: Inline HW Interface

[✓] Use Inline-Async API

Project Root: C:\Dev\My Custom Device

Advanced Options

Custom Device Extra Page Names

Insert Custom Device in submenu

[ ] Display in a submenu

Parent Menu Name: National Instruments

Maximum Custom Device Instances: 1

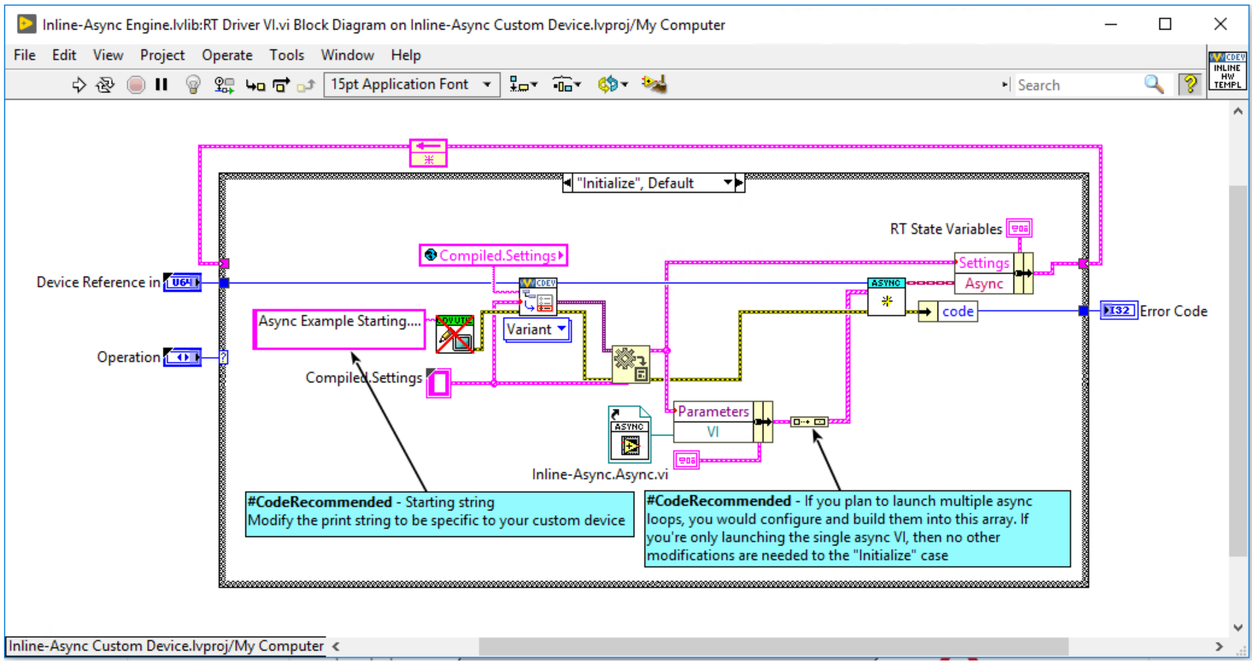
By using the Inline-Async-API, the inline-async template framework can perform the following actions.

- Initializing asynchronous VIs.
- Launching asynchronous VIs.
- Cleaning up asynchronous VIs.
- Handling and reporting errors.
- Transferring data between inline and asynchronous VIs.

The RT Driver VI of an inline custom device can communicate channel data with VeriStand. While doing so, the VI can also launch an asynchronous loop(s) to handle nondeterministic operations.

One example of a nondeterministic operation is log file data writing. The RT Driver VI of the inline custom device communicates with the asynchronous loop(s) using RT FIFOs.

Note: While the RT Driver VI is using RT FIFOs, data may be lost if elements are not read at a fast enough rate.


Inline-Async Engine.lvlib:RT Driver VI.vi Block Diagram on Inline-Async Custom Device.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
"Initialize", Default
RT State Variables
Compiled.Settings
Settings
Async
Device Reference in
Error Code
code
Async Example Starting....
Variant
Operation
Compiled.Settings
Parameters
VI
Inline-Async.Async.vi
Inline-Async Custom Device.lvproj/My Computer

#CodeRecommended - Starting string
Modify the print string to be specific to your custom device

#CodeRecommended - If you plan to launch multiple async loops, you would configure and build them into this array. If you're only launching the single async VI, then no other modifications are needed to the "Initialize" case

1.4.4 Inline Model Interface

The Inline Model Interface custom device template also has a state machine/action engine architecture. The template uses an uninitialized Feedback Node for iterative data transfer.

There are four cases defined by the Operation enumerated control.

1. Initialize – Same configuration as the Inline HW Interface.
2. Start – Same configuration as the Inline HW Interface.
3. Execute Model
4. Close – Same configuration as the Inline HW Interface.

This custom device is run in-line with the PCL. The device calls each case at a specific time with respect to the other components in the system. The PCL will not proceed until the custom device case has completed.

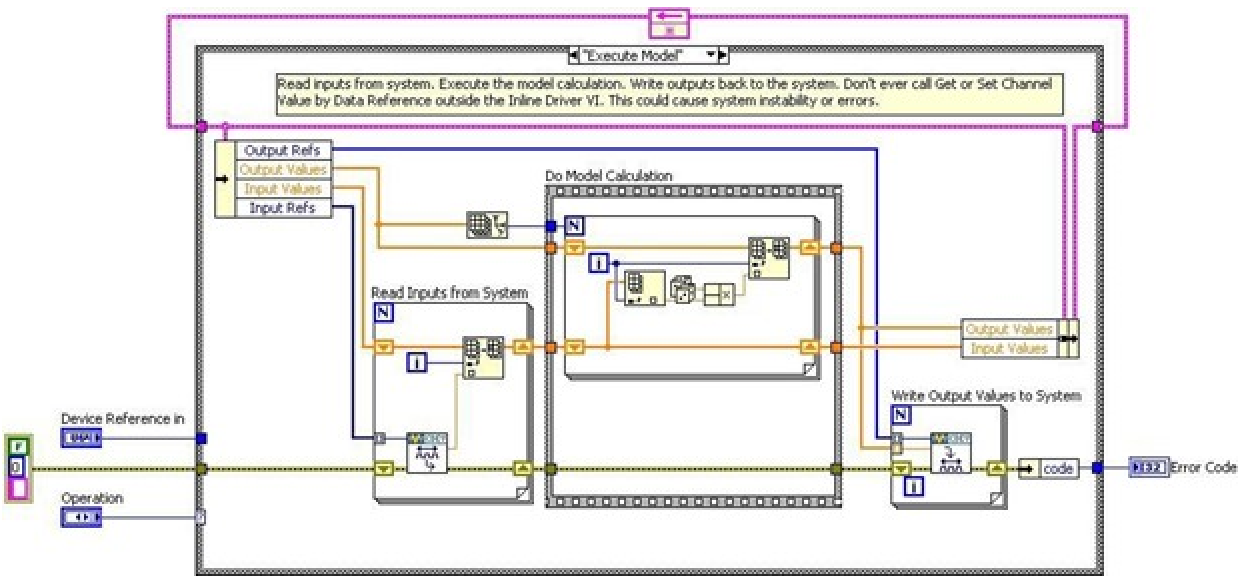
Execute Model Case

The *Execute Model* case is called and runs in the middle of the PCL.

This state takes the following steps.

1. Reads input data.
2. Performs a calculation.
3. Writes output data to VeriStand.

Using the Inline Model Interface mode enables you to process data acquired from hardware inputs and send the processed values to hardware outputs with no latency.



Read inputs from system. Execute the model calculation. Write outputs back to the system. Don't ever call Get or Set Channel Value by Data Reference outside the Inline Driver VI. This could cause system instability or errors.

1.4.5 Inline Timing and Sync

The inline timing and sync custom device is similar to the inline hardware interface custom device. The major difference between the two custom device types is that the inline timing and sync custom device can also function as a hardware synchronization master device to drive the RTSI 0 line.

1.4.6 Asynchronous Timing and Sync

The asynchronous timing and sync custom device is similar to the asynchronous custom device. The major difference between the two custom device types is that the asynchronous timing and sync custom device can also function as a hardware synchronization master device to drive the RTSI 0 line.

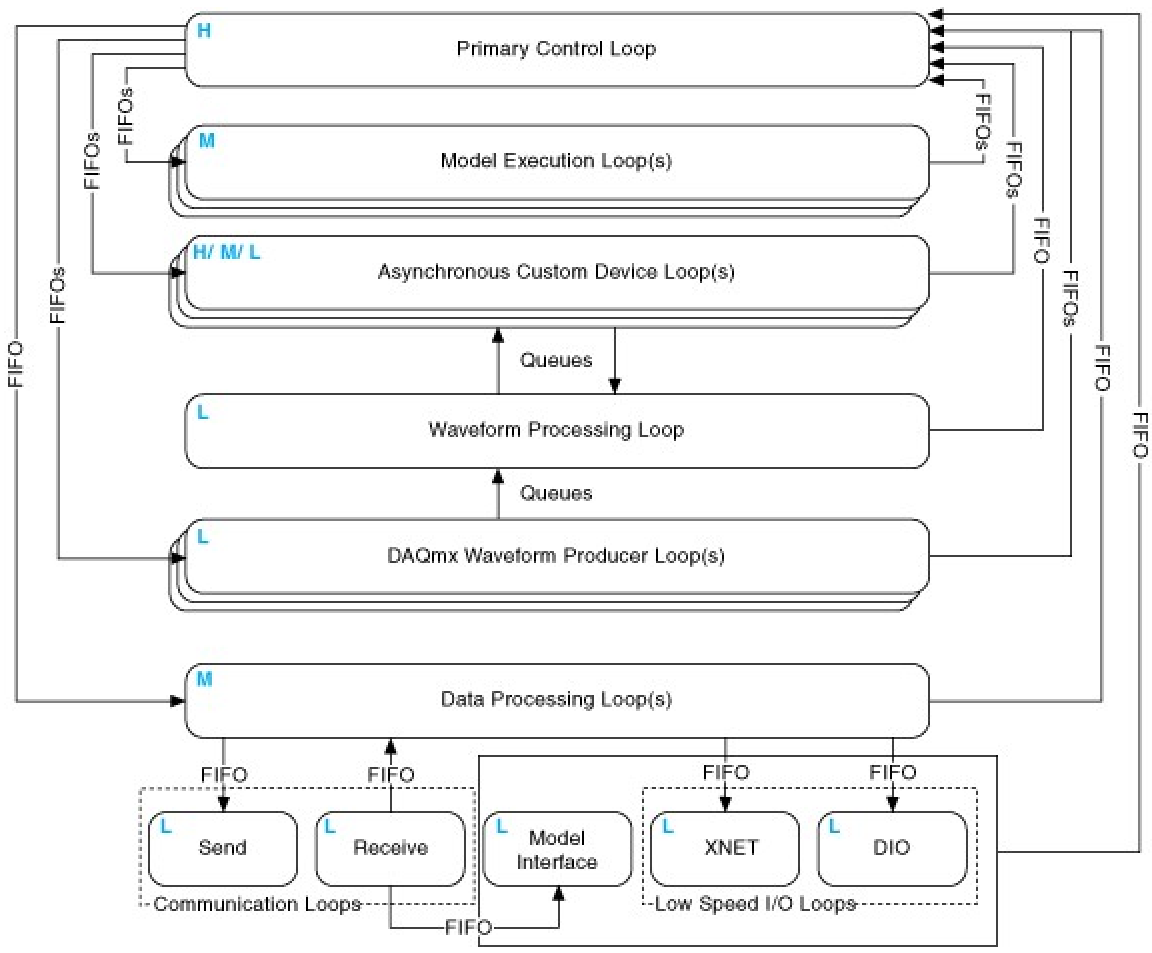
1.4.7 Outline of PCL Iteration

The order of operations in the Primary Control Loop varies with respect to the execution mode of the controller.

Note: You can adjust the settings in *System Explorer* by navigating to **Targets** » **Controller** » **Other Settings** » **Execution Mode**.

The Data Processing Loop is responsible for executing procedures, alarms, and calculated channels. For more information on hardware timing in VeriStand, refer to the *KnowledgeBase* topic Hardware I/O Latency Times in NI VeriStand.

The following diagram displays the operation of the VeriStand Engine.



Parallel Mode

In *Parallel* mode, the PCL initiates execution of models and continues to its next iteration without waiting for models to finish executing. This causes a one-cycle delay between when a model executes and when the data it produces is available to the system.

The following are the steps that the PCL takes while in parallel mode.

1. Gets inputs from hardware devices in the system definition. **Note:** If the system includes an inline hardware interface custom device, the PCL reads DAQ digital lines and counters after the Read Data from HW case of the custom device executes in step 3.
2. Reads asynchronous custom device FIFOs from the previous iteration.
3. Runs the **Read Data From HW** case of inline hardware interface custom devices. **Note:** If you configured hardware scaling, VeriStand applies the scaling after acquiring all hardware inputs.
4. Reads previous iteration data from models in the system definition. **Note:** This step executes on the second and subsequent iterations.
5. Reads data from the previous iteration of the Data Processing Loop.
6. Processes system mappings. **Note:** VeriStand components and custom devices cannot read data from a previous step until the PCL processes system mappings. This is true even if the previous step acquired the data the component needs.
7. Runs the **Execute Model** case of inline model interface custom devices.

8. Executes steps of running real-time sequences. **Notes:**
   - VeriStand executes real-time sequences after input operations but before output operations. VeriStand continues to run every step of the real-time sequence until the sequence is complete, reaches a Yield step, or completes an iteration of a loop with Auto Yield set to `TRUE`. If a sequence takes longer than the given time for an iteration of the PCL, the PCL runs late.
   - To avoid errors, break up the timing of the steps by placing Yield steps throughout the sequence and enabling the Auto Yield property for any loops in the sequence.
9. Processes system mappings.
10. Writes data to models.
11. Initiates asynchronous execution of models.
12. Writes data to the Data Processing Loop.
13. Writes output data to hardware devices.
14. **(Optional)** For inline hardware interface custom devices, runs the **Write Data to HW** case.
15. Writes data to asynchronous custom device FIFOs.

Low Latency Mode

In *Low Latency* mode, the PCL waits for the Model Execution Loop(s) to finish writing data to models before it reads and publishes model values to the system. This occurs during every iteration of the system.

When the model completes execution, the PCL provides data from the model to other loops during the same iteration that the model generated the data.

The following are the steps that the PCL takes while in low latency mode.

1. Gets inputs from hardware devices in the system definition. **Note:** If the system includes an inline hardware interface custom device, the PCL reads DAQ digital lines and counters after the Read Data from HW case of the custom device executes in step 3.
2. Reads asynchronous custom device FIFOs from the previous iteration.
3. Runs the **Read Data from HW** case of inline hardware interface custom devices. **Note:** If you configured hardware scaling, VeriStand applies the scaling after acquiring all hardware inputs.
4. Reads data from the previous iteration of the Data Processing Loop.
5. Processes system mappings. **Note:** VeriStand components (including custom devices) cannot read data from a previous step until the PCL processes system mappings, even if the previous step acquired the data the component needs.
6. Runs the **Execute Model** case of inline model interface custom devices.
7. Executes steps of running real-time sequences. **Notes:**
   - VeriStand executes real-time sequences after input operations but before output operations. VeriStand continues to run every step of the real-time sequence until the sequence is complete, reaches a Yield step, or completes an iteration of a loop with Auto Yield set to `TRUE`. If a sequence takes longer than the given time for an iteration of the PCL, the PCL runs late.
   - To avoid errors, break up the timing of the steps by placing Yield steps throughout the sequence and enabling the Auto Yield property for any loops in the sequence.
8. Processes system mappings.
9. Writes data to models.

10. Initiates execution of models and waits for them to complete execution.
11. Reads data from models.
12. Processes system mappings.
13. Writes data to the Data Processing Loop.
14. Writes output data to hardware devices.
15. **(Optional)** For inline hardware interface custom devices, runs the **Write Data to HW** case.
16. Writes data to asynchronous custom device FIFOs.

CHAPTER

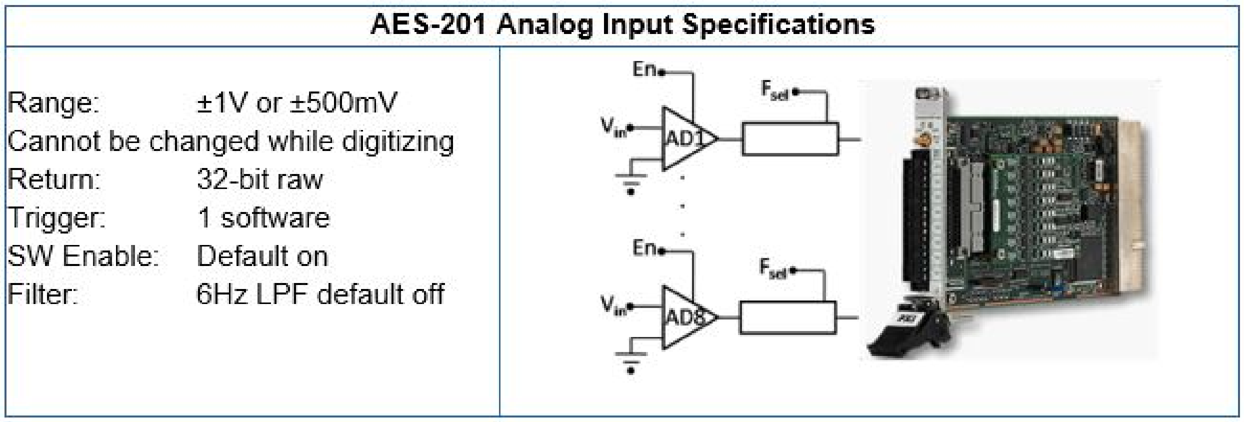
TWO

IMPLEMENT THE CUSTOM DEVICE

2.1 Implementing a Custom Device

We will now walk through the implementation of a hypothetical third-party custom device for the AES-201. This example will focus on the custom device process. For more information on implementing a custom device, refer to Implementing a Custom Device.

The specifications of this custom device are displayed in the following image.



AES-201 Analog Input Specifications

Range:	±1V or ±500mV
Cannot be changed while digitizing	
Return:	32-bit raw
Trigger:	1 software
SW Enable:	Default on
Filter:	6Hz LPF default off

En
F_{sel}
V_{in}
AD1
En
F_{sel}
V_{in}
AD8

2.1.1 Determine Custom Device Feasibility

Before we begin, let us consider the customer needs, analyze any risks, and create specifications for the custom device.

Customer Needs

Our customer requires 32-bits of resolution for their real-time (RT) test system. This is the only PXI digitizer that fulfills their requirements. After checking with NI.com and the manufacturer, we did not find an existing AES-201 custom device. We determine that a new custom device is necessary.

Risk Analysis

The AES-201 ships with a hardware driver that's compatible with LabVIEW Real-Time and a LabVIEW API. We have a real-time desktop target that is identical to our customer's platform. At our request, the customer has provided their model DLL. This will allow us to test and benchmark on a similar structure to our customer's system.

Development Specifications

Based on the AES-201, we create the following specifications.

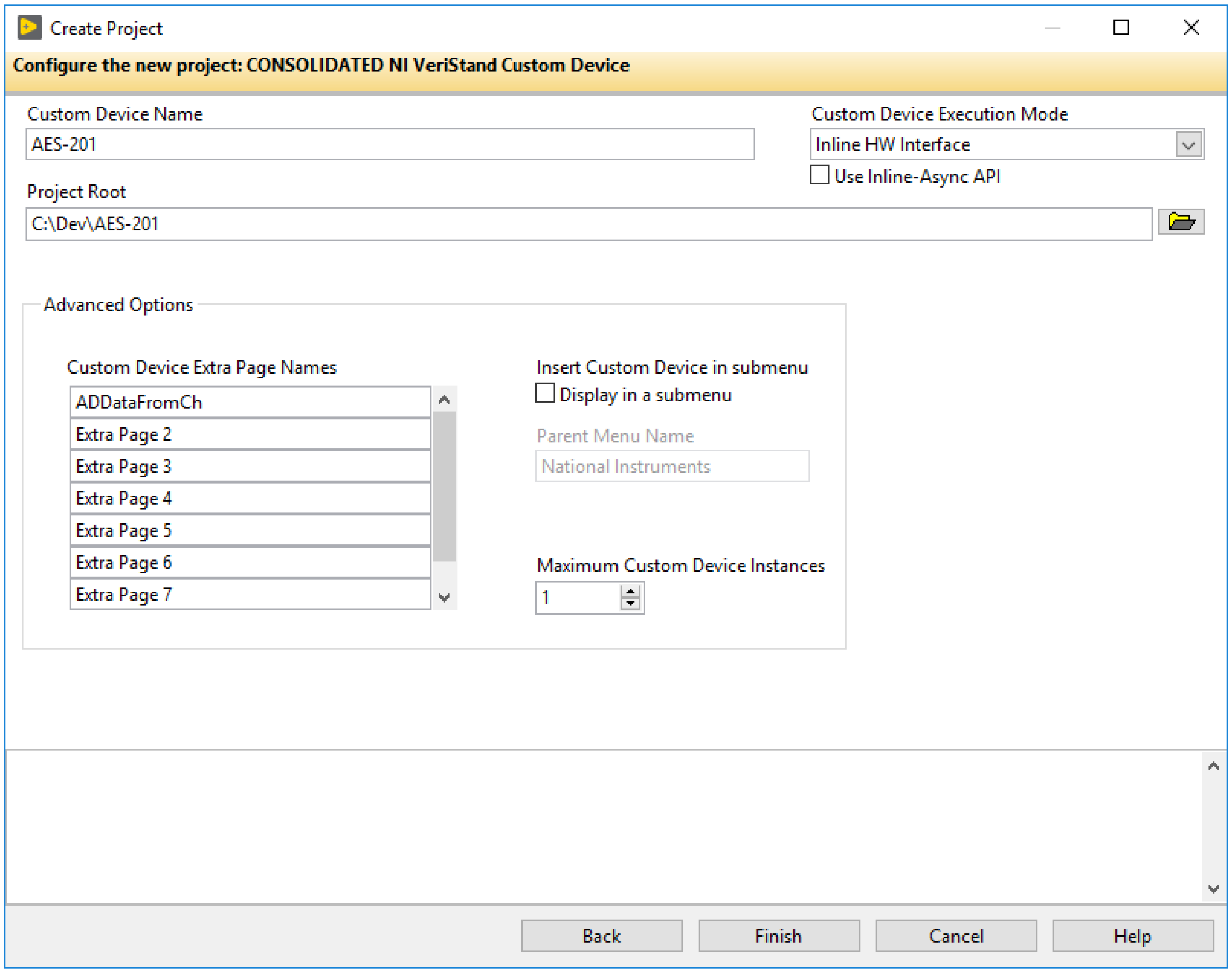
- Have eight output channels: *ADDataFromCh<1..8>*.
- Have nine input channels: *ADEnCh<1..8>*, *SWTrig*.
- Have nine properties: *FilterEn<1..8>* and *Range*.
- Use a nested two-level hierarchy.
- Override the default channel page for *ADDataFromCh<1..8>*.
- Use the default page for everything else. **Note:** We only need one extra page, but we will create several just in case requirements change.
- Use the Hardware Inline custom device to avoid FIFO latency.

2.1.2 Build the Template Project

We will begin by building a template project.

1. Open the niveristand-custom-device-wizard from LabVIEW by navigating to **Create Project** » **NI VeriStand** » **CONSOLIDATED NI VeriStand Custom Device**.
2. Enter the **Custom Device Name**. This will serve as the sub folder name.
3. Select the **Custom Device Execution Mode**.
4. Select the **Project Root**. The wizard creates the new LabVIEW Project in a sub folder inside the project root. We do not need to specify a sub folder for the device because the wizard creates one.
5. Click **Finish**.

The following image displays a configured wizard that will generate a LabVIEW Project for the AES-201 custom device.


Create Project

Configure the new project: CONSOLIDATED NI VeriStand Custom Device

Custom Device Name: AES-201

Custom Device Execution Mode: Inline HW Interface

☐ Use Inline-Async API

Project Root: C:\Dev\AES-201

Advanced Options

Custom Device Extra Page Names:
- ADDataFromCh
- Extra Page 2
- Extra Page 3
- Extra Page 4
- Extra Page 5
- Extra Page 6
- Extra Page 7

Insert Custom Device in submenu

☐ Display in a submenu

Parent Menu Name: National Instruments

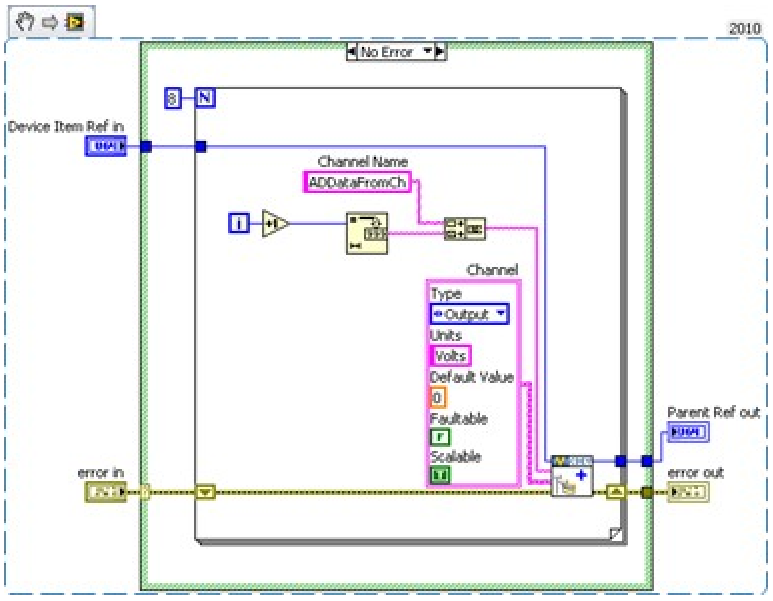
Maximum Custom Device Instances: 1

2.1.3 Build the Configuration

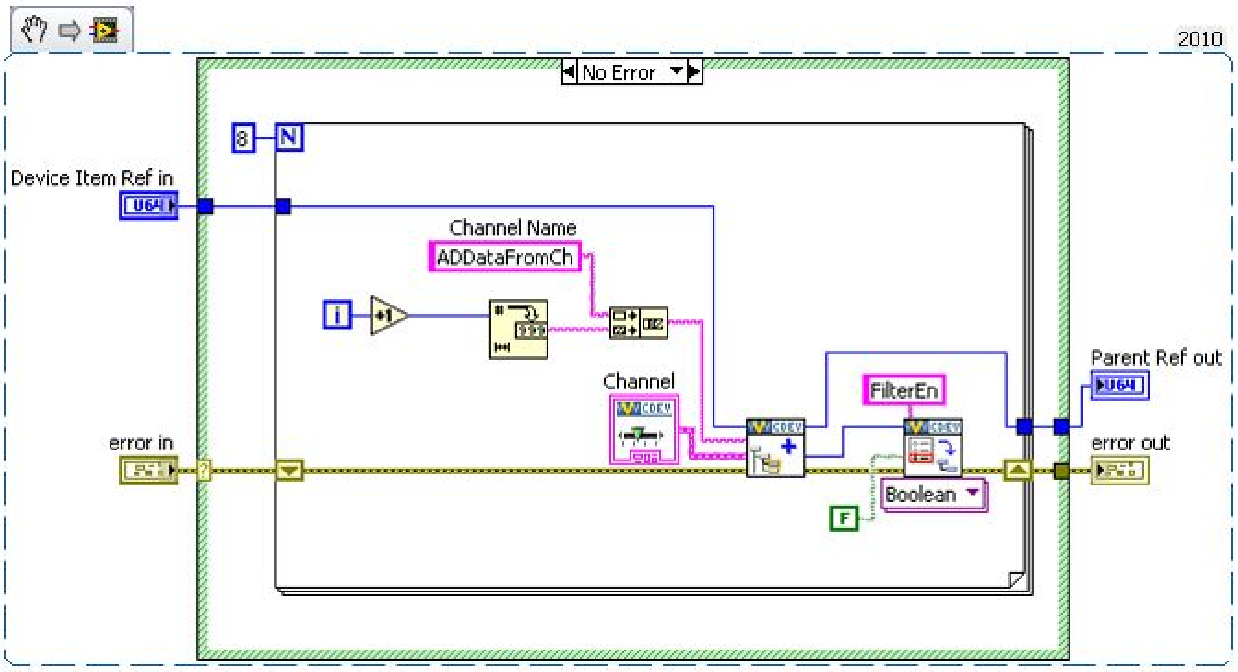
We will now modify the LabVIEW Project VIs generated by the niveristand-custom-device-wizard.

Edit the Initialization VI

We will start by opening the Initialization VI. Locate the VI by navigating to **AES-201 System Explorer.lvlib** » **System Explorer** » **Dynamically Called**. In the initialization VI, we will build-up the default channel list. For more information, refer to Add Custom Device Channels and Waveforms.

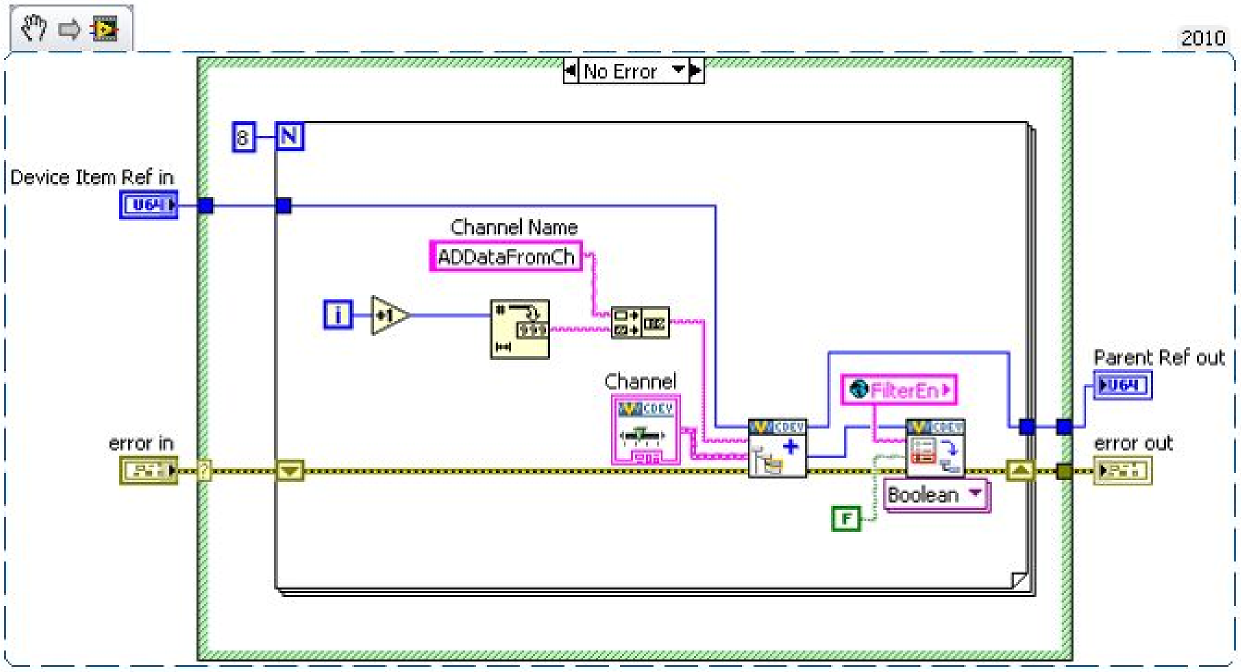


Add a Boolean property to each channel. This property will indicate the state of the filter on the channel. For more information, refer to Adding Custom Device Item Properties.



Replace the string constant with a global variable that has the same default value as the constant. Use the global variable *Constants.vi* by navigating to **<Custom Device Name> Shared.lvlib** » **Shared**.

Note: You should use global variables or enum type definitions for any constants that will be reused throughout the custom device.

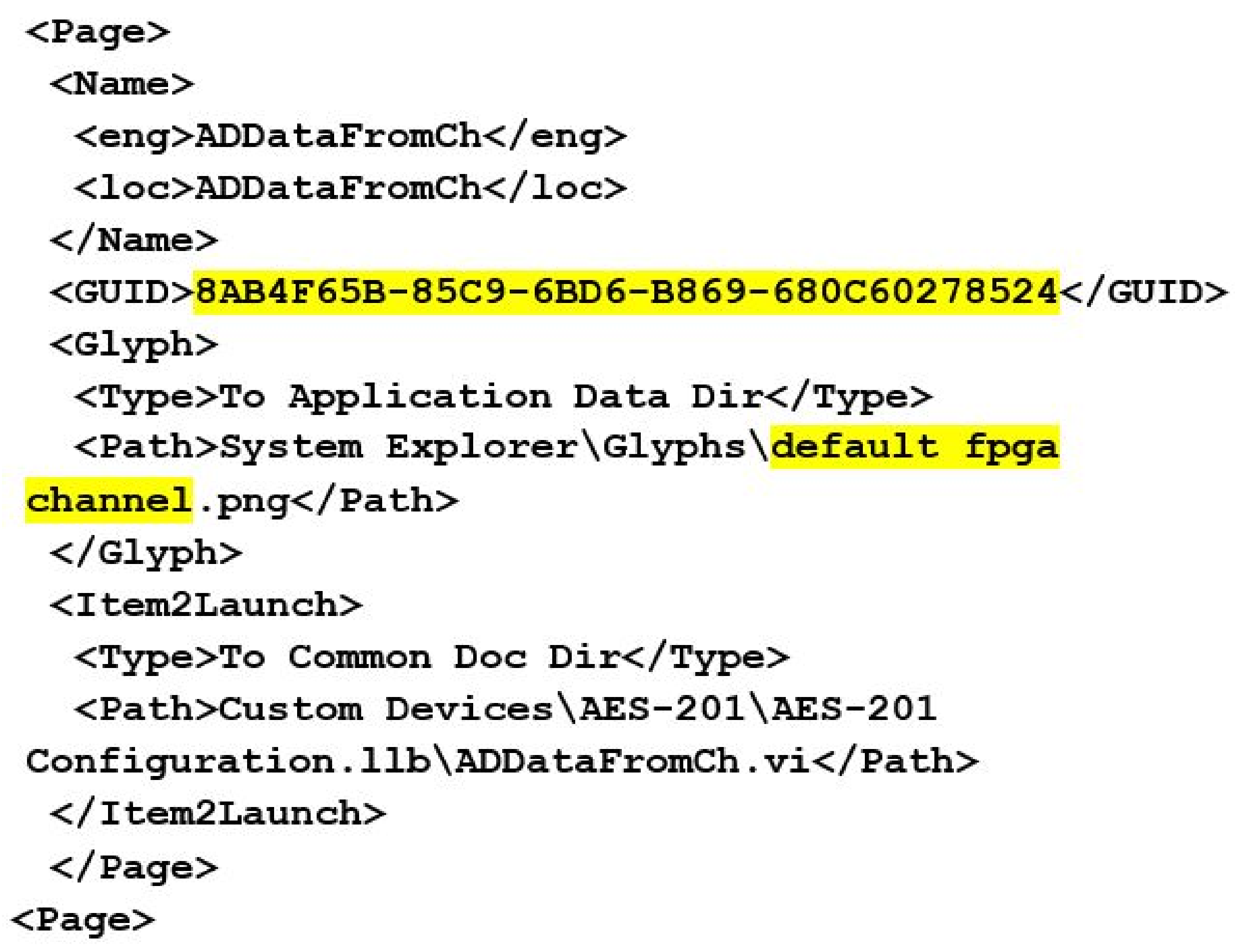


Override the Default Page

We want to override the default channel page to add a control that allows the operator to set the filter. We created an extra page called *ADDataFromCh.vi* for this purpose.

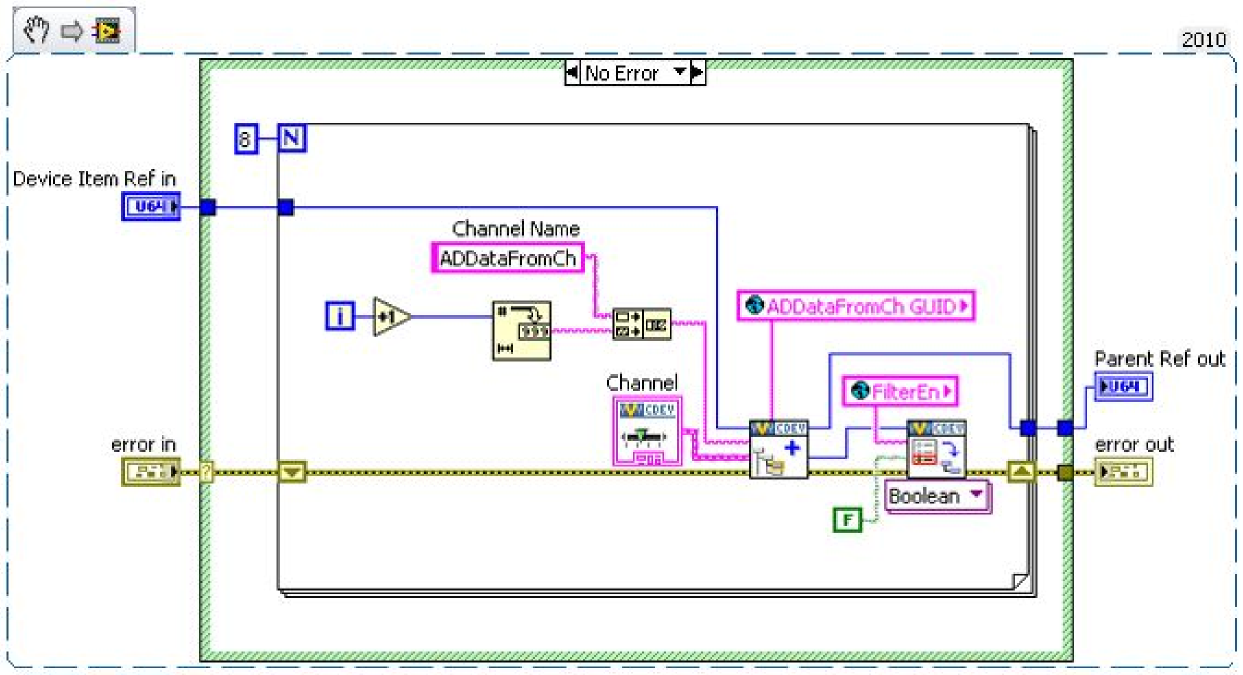
Open the custom device XML to find the GUID associated with the extra page. You should update the glyph of the channel page to *default fpga channel*. Operators are used to having channels associated with this glyph.

Note: You can also change the glyph of the main page to *daq device*.



```
 <Page>
  <Name>
   <eng>ADDataFromCh</eng>
   <loc>ADDataFromCh</loc>
  </Name>
  <GUID>8AB4F65B-85C9-6BD6-B869-680C60278524</GUID>
  <Glyph>
   <Type>To Application Data Dir</Type>
   <Path>System Explorer\Glyphs\default fpga
 channel.png</Path>
  </Glyph>
  <Item2Launch>
   <Type>To Common Doc Dir</Type>
   <Path>Custom Devices\AES-201\AES-201
 Configuration.llb\ADDataFromCh.vi</Path>
  </Item2Launch>
  </Page>
<Page>
```

Add the GUID to the global variable and wire the global into the GUID terminal of Add Custom Device Channel. This will associate the channel with the VI.

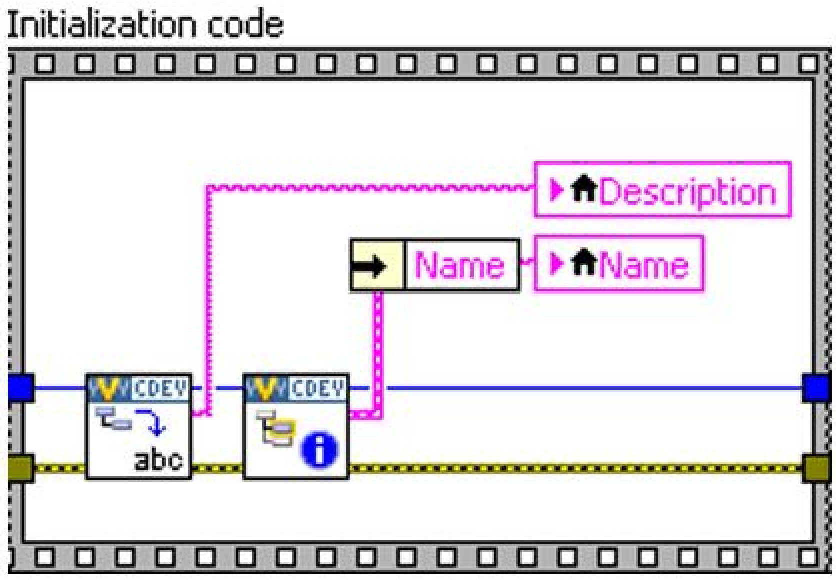


When the operator clicks on **ADDataFromCh<1..8>** in the configuration tree, *ADDataFromCh.vi* runs as a sub panel in System Explorer instead of the default channel page.

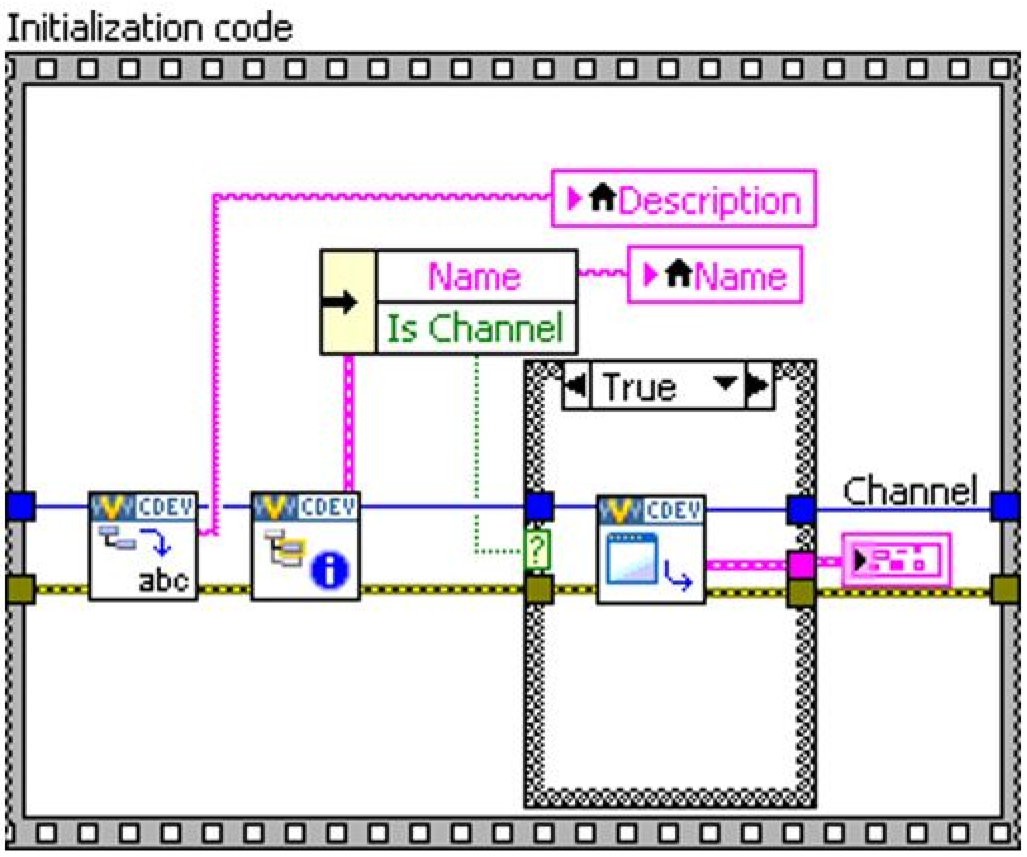
From here-on, we will set properties when we create the item rather than using the Set Item Property VI to set them on the item reference.

Edit the Extra Page

Now that we have linked the channels to the extra page, we will make edits to the extra page, *ADDataFromCh.vi*. In the Initialization frame, we will add code to display the channel information.



Operators are used to seeing channel data when they click on a channel. If the device is a channel, we will send the channel data to an indicator on the front panel.

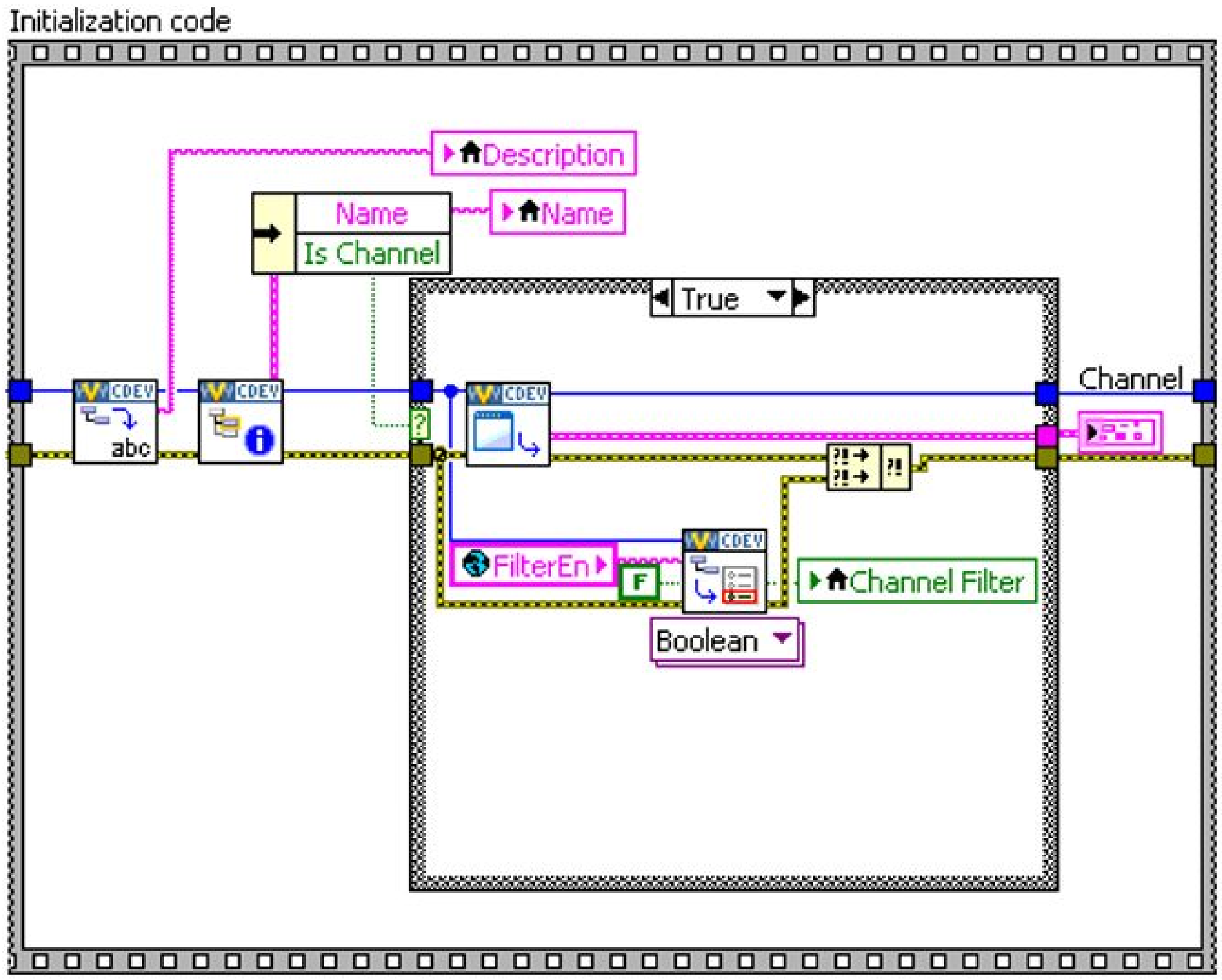


You should use the Boolean outputs from functions in the API to make sure that you are operating on a valid reference. In this case, we will only retrieve the channel data if we have a valid channel reference.

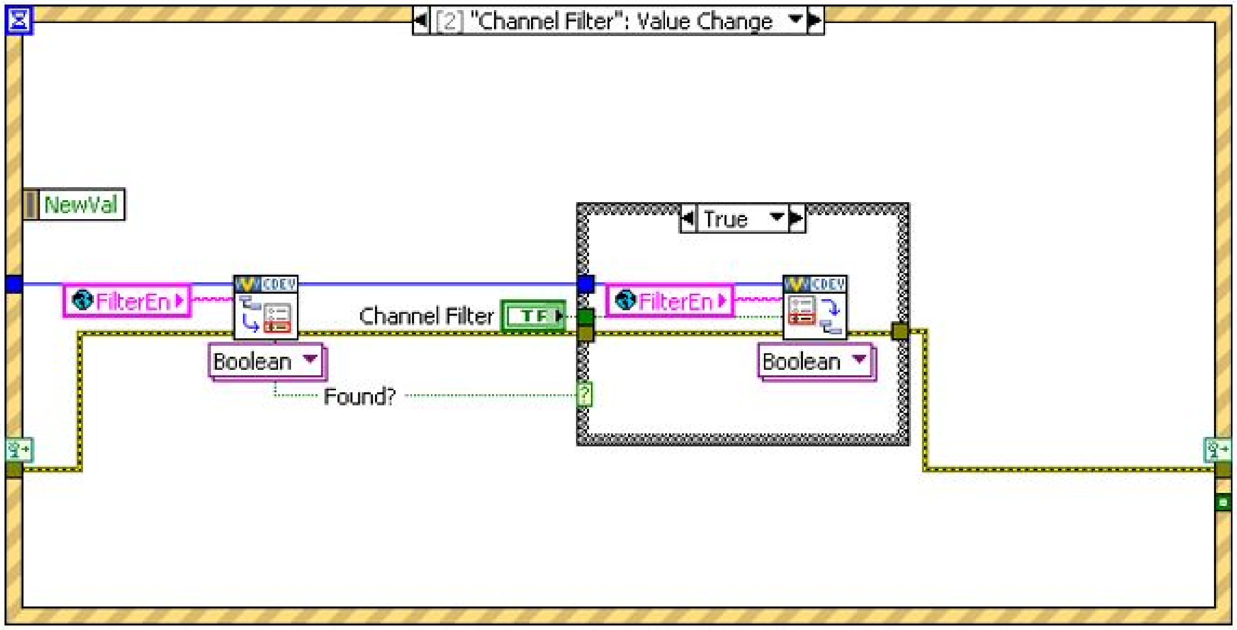
Another option is to specify the default property value. The default property value is returned if the property is not found. Using the default property value does not set the property.

The initialization frame will read the name and description from the device reference. Do the same thing for the FilterEn property so the operator can see the state of the channel's filter setting.

VeriStand is responsible for passing the correct channel reference to the custom device and storing state data for all the controls and indicators. The developer is responsible for acting on the reference and displaying the state.

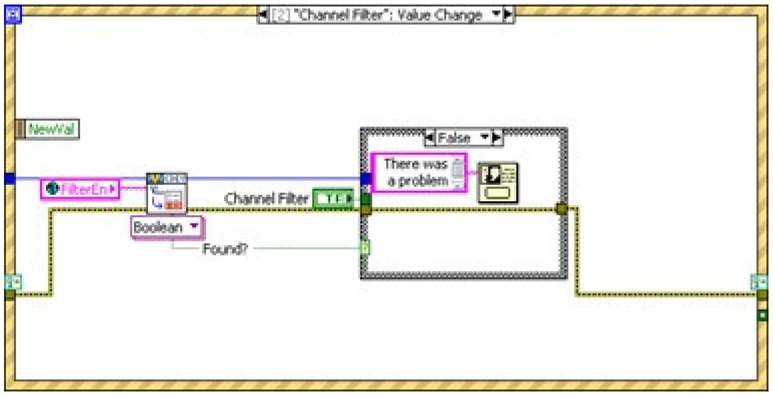


Add a Boolean control to the front panel called *Channel Filter*. Create a case in the Event Structure for the control's value change. If the FilterEn property is found, set the property according to the value of the control. If the FilterEn property is not found, show a dialog box with debugging information.



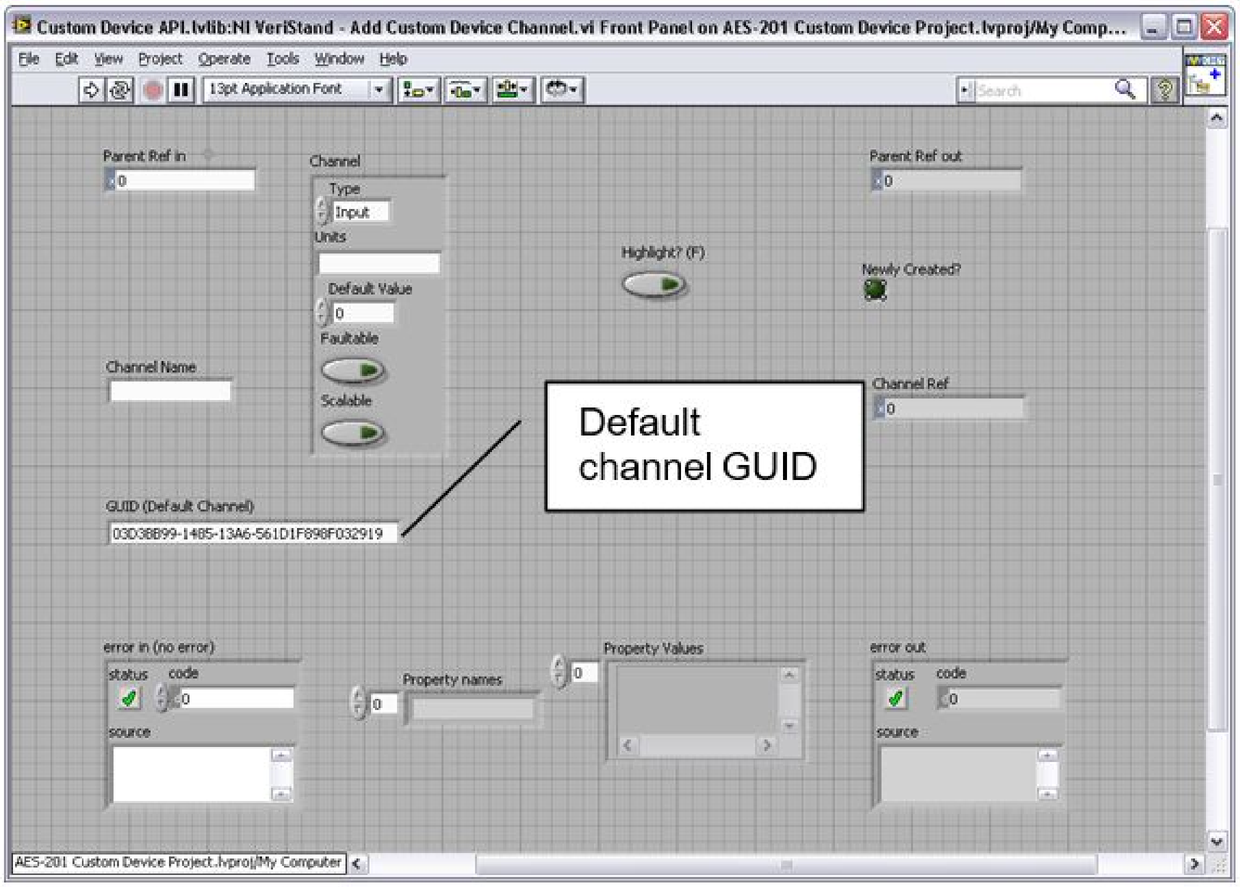
If the operator does not change this control, the property is never created. To rectify this, you can initialize the property in the Initialization VI or you can assume a default value when you read the property.

Remember, this VI runs on the host computer. We can launch a pop-up dialog box to assist with debugging.



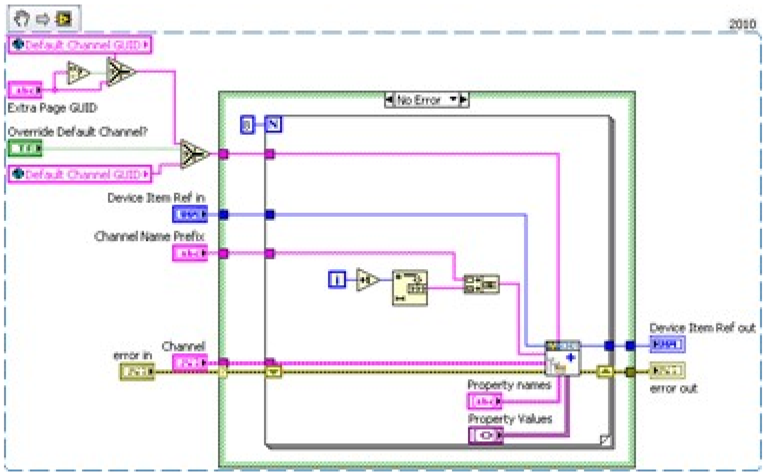
Now we will build a subVI that creates channels so we can reuse it for the enabled channels.

Add the default channel GUID to the *Constants.vi*. You can get the GUID from the front panel of **Add Custom Device Channel**.



For reference, the GUID is `03D3BB99-1485-13A6-561D1F898F032919`.

If the **Override Default Channel?** terminal of our subVI is true, the VI takes a GUID from the caller. If not, the VI uses the default channel GUID.



The properties are set from the Add Custom Device Channel VI directly. You can use this subVI in many custom device projects.

Allow Simultaneous Calls to the Same Extra Page

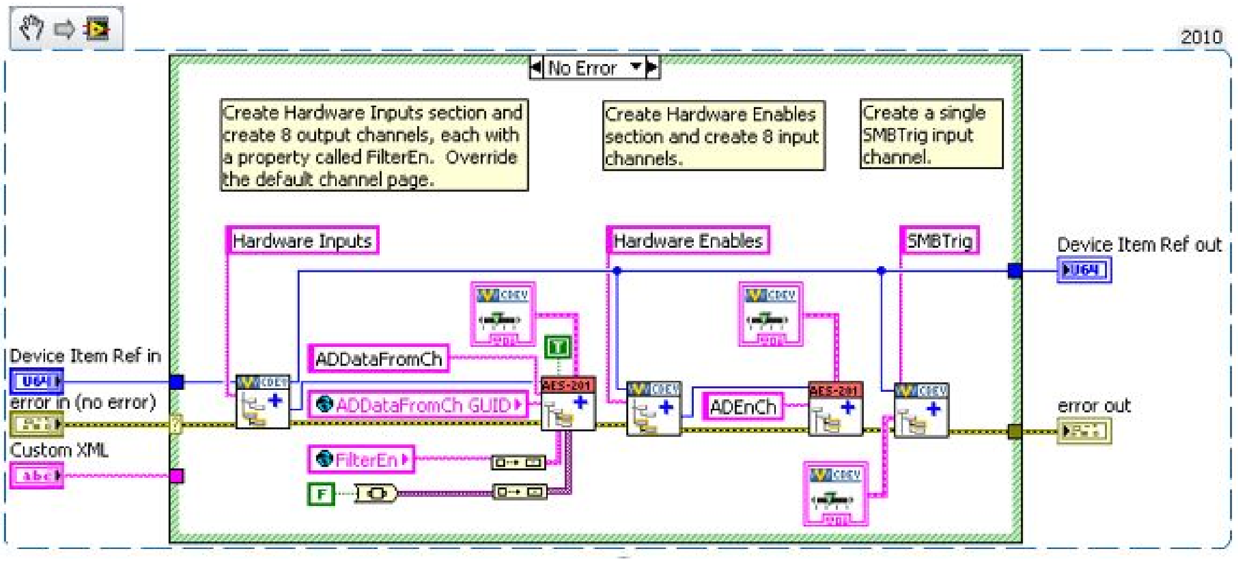
Custom devices execute as reentrant on the target. This enables the operator to run multiple independent instances of the same custom device. This would be useful if the operator has several AES-201 cards.

To preserve this capability, enable **Preallocated clone reentrant execution** from the subVI by navigating to **File** » **VI Properties** » **Execution**. For more information, refer to the LabVIEW Help topic **Reentrancy: Allowing Simultaneous Calls to the Same SubVI**.

Final Initialization VI

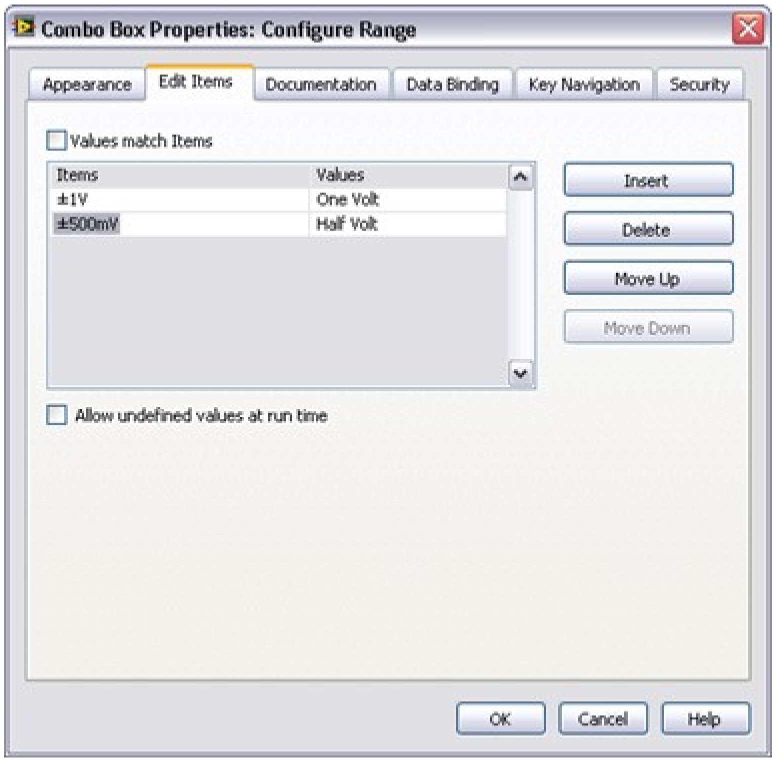
The final Initialization VI creates two sections.

1. Hardware Inputs, with eight output channels.
2. Hardware Enables, with eight input channels. **Note:** We will also create an input channel for the software trigger.


2010
No Error
Create Hardware Inputs section and create 8 output channels, each with a property called FilterEn. Override the default channel page.
Create Hardware Enables section and create 8 input channels.
Create a single SMBTrig input channel.
Hardware Inputs
Hardware Enables
SMBTrig
Device Item Ref out
Device Item Ref in
error in (no error)
Custom XML
ADDataFromCh
ADDataFromCh GUID
FilterEn
ADEnCh
AES-201
error out

Configure the Main Page

Now that the initialization routine is complete, we should configure the main page. We will use a type definition combo box to set the range of the AES-201. Add the type definition to the *AES-201 System Explorer.lvlib*.



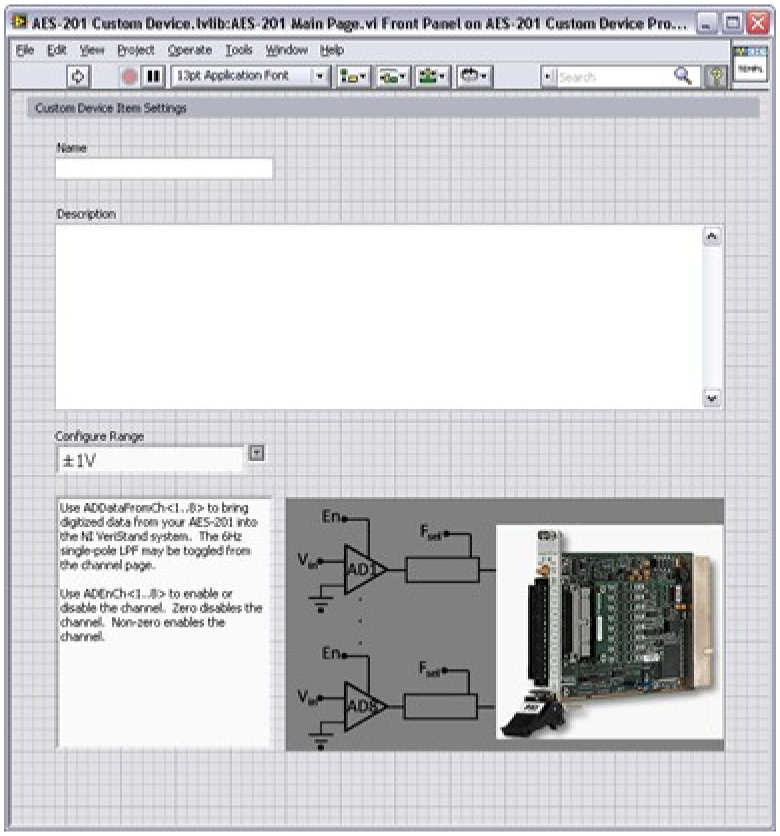
☐ Values match Items

Items	Values
±1V	One Volt
±500mV	Half Volt

☐ Allow undefined values at run time

Modify the main page so the operator can set the range of the device.

Note: You do not have to override the main page with a custom page. You can modify the main page directly.


AES-201 Custom Device.lvlib:AES-201 Main Page.vi Front Panel on AES-201 Custom Device Pro...
File
Edit
View
Project
Operate
Tools
Window
Help
13pt Application Font
Search
En
F
V
AD1
AD8

Custom Device Item Settings

Name

Description

Configure Range

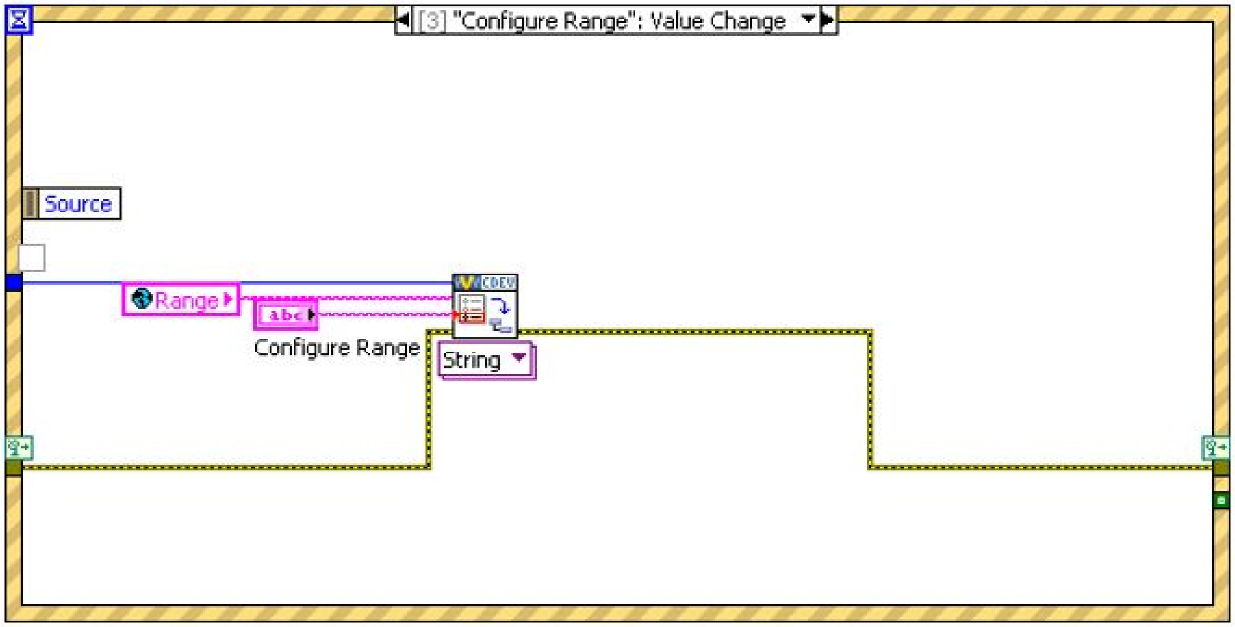
±1V

Use ADDataFromCh<1..8> to bring digitized data from your AES-201 into the NI VeriStand system. The 6Hz single-pole LPF may be toggled from the channel page.

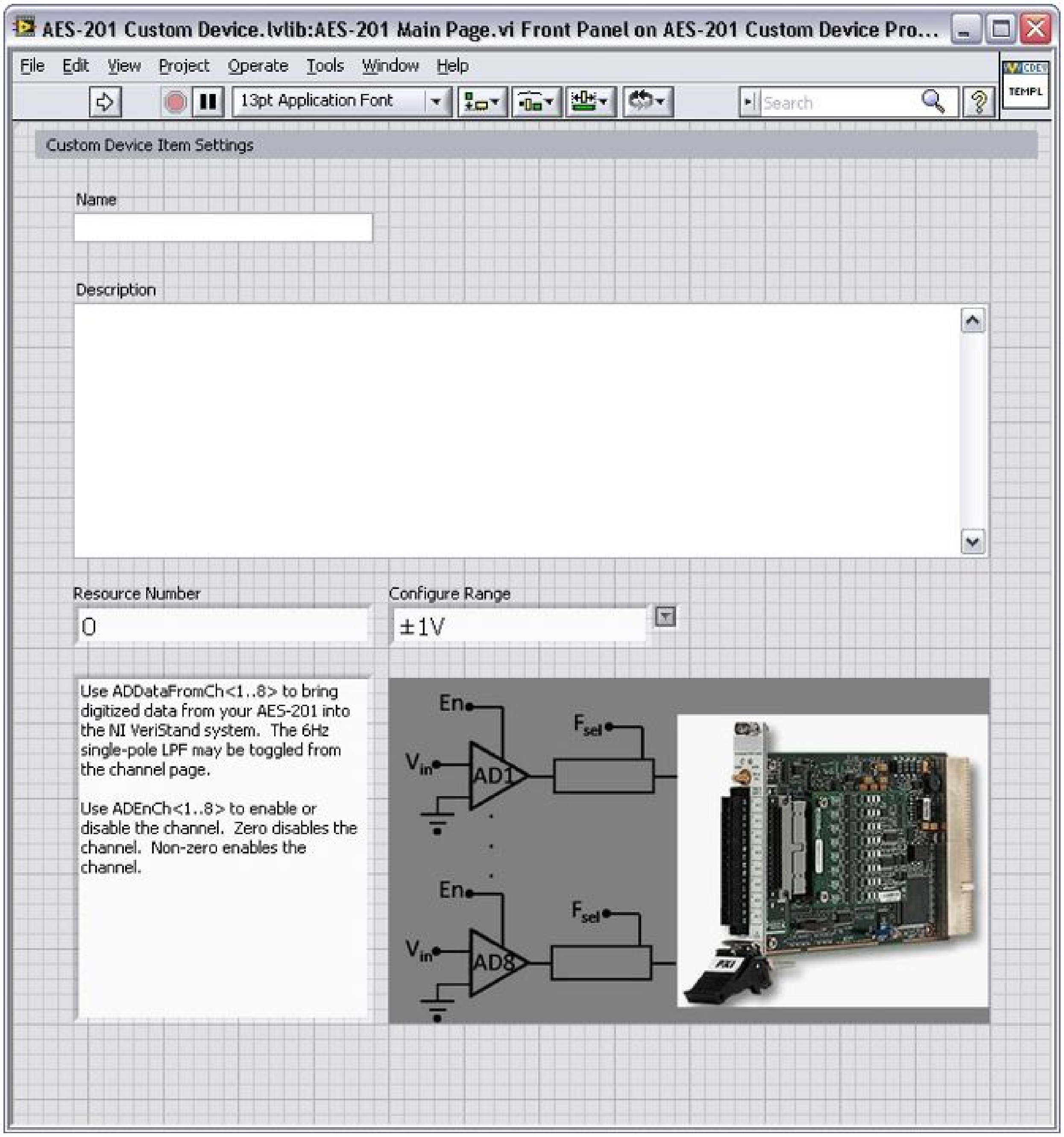
Use ADEnCh<1..8> to enable or disable the channel. Zero disables the channel. Non-zero enables the channel.

Add another string to the global variable for the range property.

Add an event case to the main page that sets the range property when the operator changes the value of the control.



The engine will need to know how to address the board. Add another control so the operator can configure the **Resource Number**.


AES-201 Custom Device.lvlib:AES-201 Main Page.vi Front Panel on AES-201 Custom Device Pro...
File Edit View Project Operate Tools Window Help
13pt Application Font
Search
TEMPL
En
Fsel
Vin
AD1
AD8

Custom Device Item Settings

Name

Description

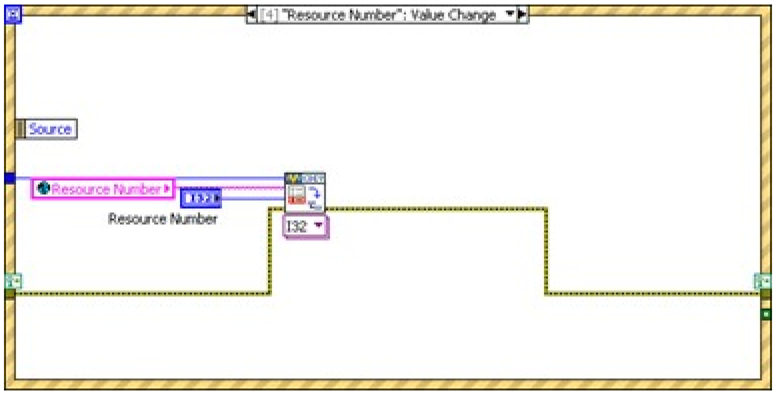
Resource Number: 0

Configure Range: ±1V

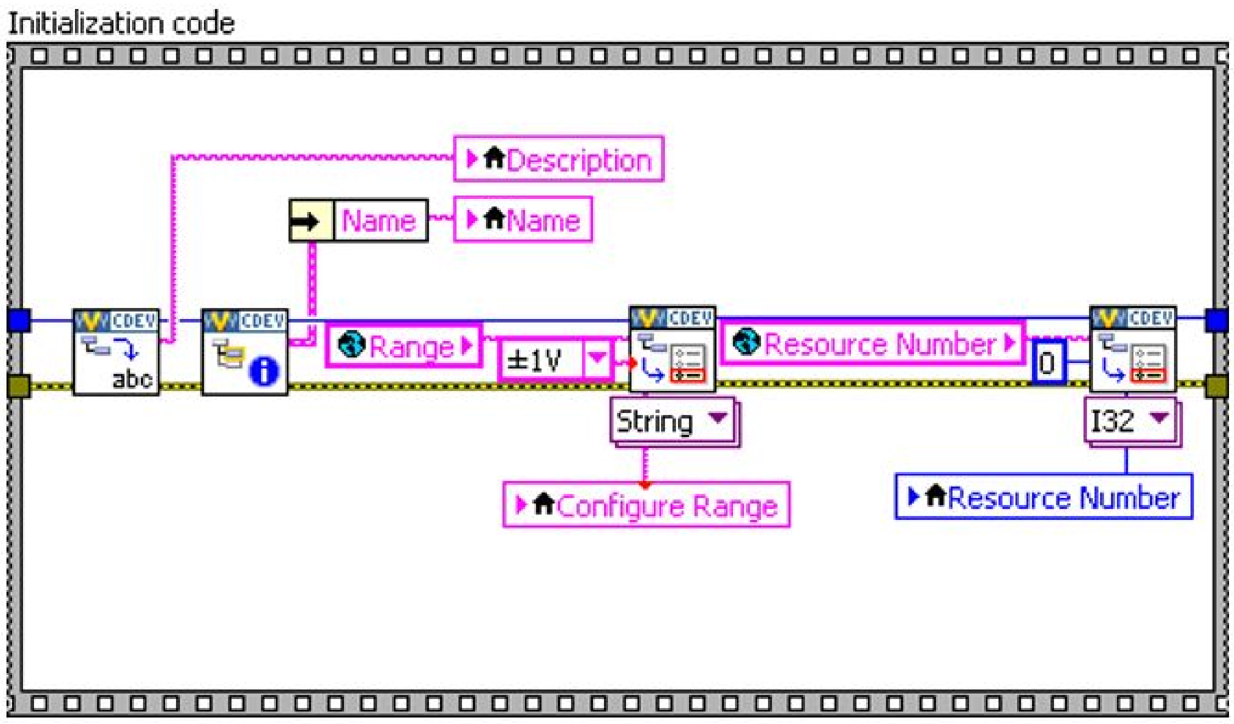
Use ADDataFromCh<1..8> to bring digitized data from your AES-201 into the NI VeriStand system. The 6Hz single-pole LPF may be toggled from the channel page.

Use ADEnCh<1..8> to enable or disable the channel. Zero disables the channel. Non-zero enables the channel.

Add the **Resource Number** as an event case to set the resource number property.

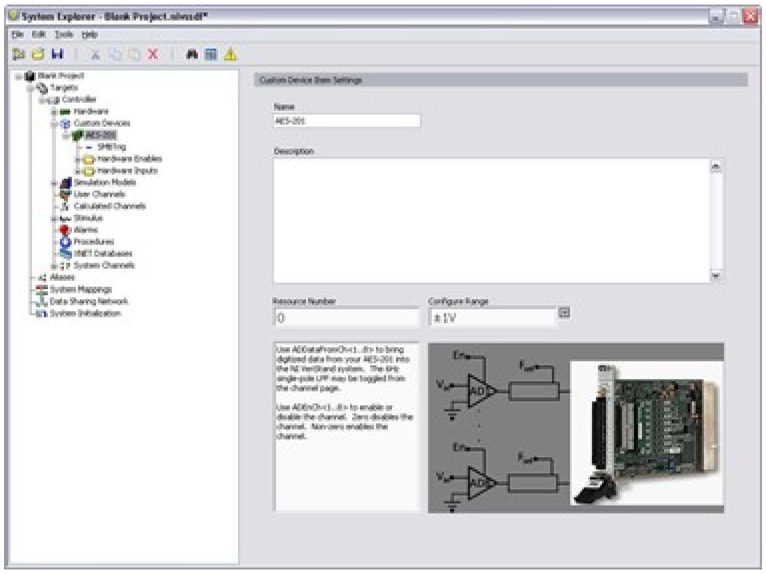


Read the device's resource name and range into the corresponding controls in the initialization frame similarly to the extra channel page's filter property.



Build the Final Configuration

Build the custom device and inspect the hierarchy, sections, channels, main page, and extra pages.



Name
AES-201

Description

Resource Number
0

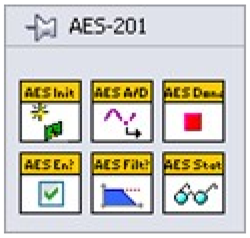
Configure Range
± 1V

Use ADDataFromCh<1...8> to bring digitized data from your AES-201 into the NI VeriStand system. The 8Hz single-pole LPF may be toggled from the channel page.

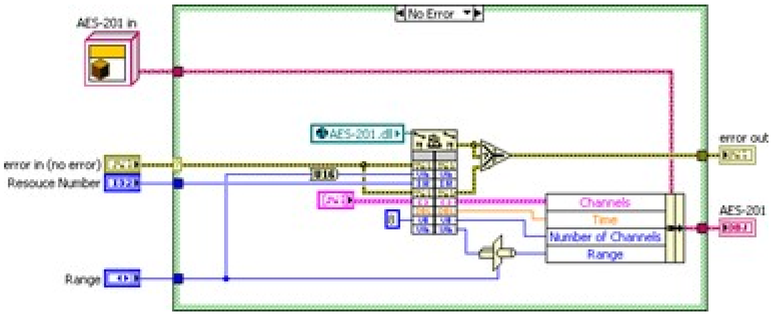
Use ADEnCh<1...8> to enable or disable the channel. Zero disables the channel. Non-zero enables the channel.

2.1.4 Build the Driver

The AES-201 comes with a simple LabVIEW API. We will use the API to build the RT driver portion of the custom device.



Functions in the API call into the hardware DLL. This is typical of a LabVIEW API. This paradigm requires the developer to post the DLL to the execution host.

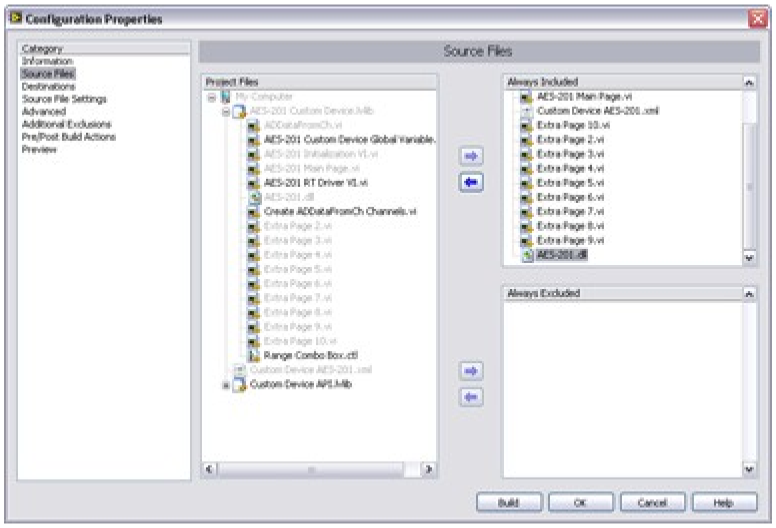


Modify the custom device to package the DLL with the custom device and deploy it to the execution host.

2.1.5 Add Custom Device Dependencies

Shared libraries are typically *.dll* files on Windows operating systems and *.so* files on Linux systems. If you're building a custom device for a PXI target, you will be working with *.so* files.

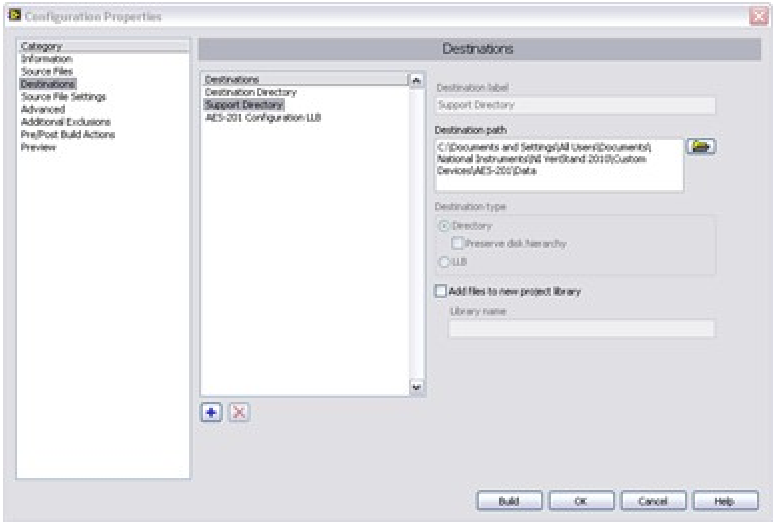
There are two parts to packaging dependencies. First, you need to incorporate the dependency into the LabVIEW Project.



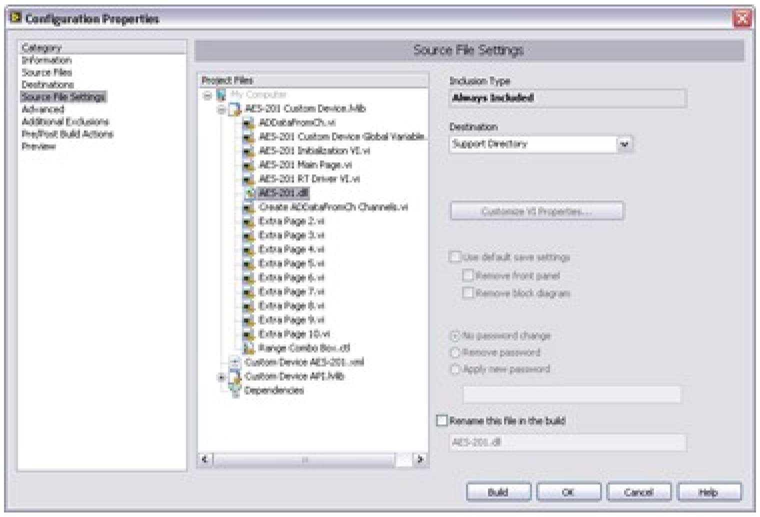
Add the DLL to the custom device LabVIEW library.

Modify the configuration's **Source Distribution** by adding the DLL to the **Always Included** list.

Be sure to note the location of the support directory. In this case it's `C:\Documents and Settings\All Users\Documents\National Instruments\NI VeriStand 2018\Custom Devices\AES- 201\Data`.



Set the destination directory for the DLL to the support directory.



Project Files
- My Computer
  - AES-201 Custom Device.lvlib
    - ADDataFromCh.vi
    - AES-201 Custom Device Global Variable...
    - AES-201 Initialization VI.vi
    - AES-201 Main Page.vi
    - AES-201 RT Driver VI.vi
    - AES-201.dll
    - Create ADDataFromCh Channels.vi
    - Extra Page 2.vi
    - Extra Page 3.vi
    - Extra Page 4.vi
    - Extra Page 5.vi
    - Extra Page 6.vi
    - Extra Page 7.vi
    - Extra Page 8.vi
    - Extra Page 9.vi
    - Extra Page 10.vi
    - Range Combo Box.ctl
  - Custom Device AES-201.xml
  - Custom Device API.lvlib
  - Dependencies

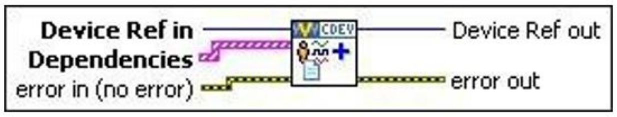
Inclusion Type: Always Included
Destination: Support Directory
☐ Use default save settings
☐ Remove front panel
☐ Remove block diagram
● No password change
○ Remove password
○ Apply new password
☐ Rename this file in the build: AES-201.dll

When you build the configuration, LabVIEW sends the DLL to the support directory.

The second part to packaging dependencies is in incorporating the dependency into the custom device. Use the Add Custom Device Dependencies VI to deploy the library to the execution host.

NI VeriStand – Add Custom Device Dependencies VI
Owning Palette: Dependencies VIs

Adds dependencies to a custom device.

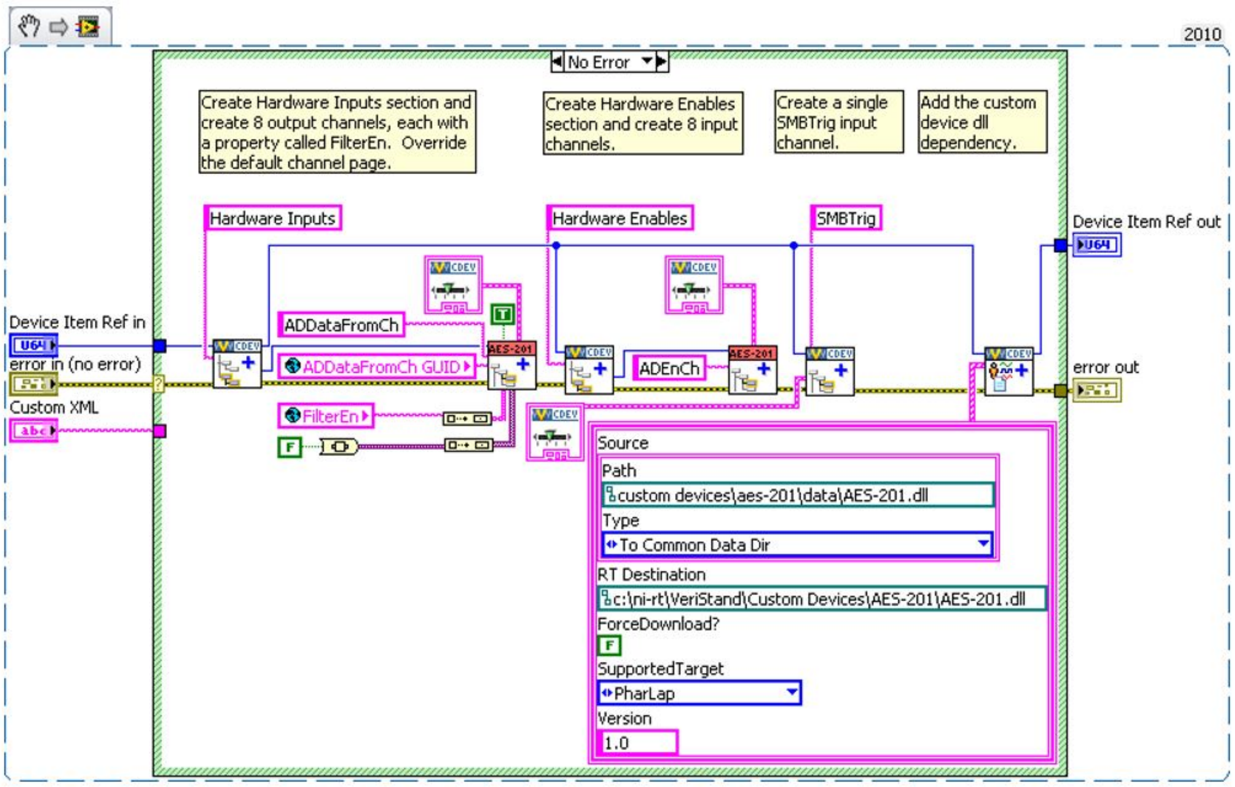


U64	**Device Ref in** is the NI VeriStand reference to the custom device.	
[▪]	**Dependencies** is an array of Custom Device File Dependency controls.	
	▭	**Path** is the path and file name of the dependency
	◂▸	**Type** is absolute or relative to one of NI VeriStand's built-in file paths. See the What is a Custom Device for a list of built-in file paths.
	▭	**RT Destination** specifies the directory and file name of the dependency on the execution host
	TF	**Force Download** terminal must be false.
	◂▸	**SupportedTarget** specifies the target operating systems that will receive the dependency
	abc	**Version**
	error in describes error conditions that occur before this node runs. This input provides standard error in functionality.	
U64	**Device Ref out** is a duplicate of the Device Ref in.	
	error out contains error information. This output provides standard error out functionality.	

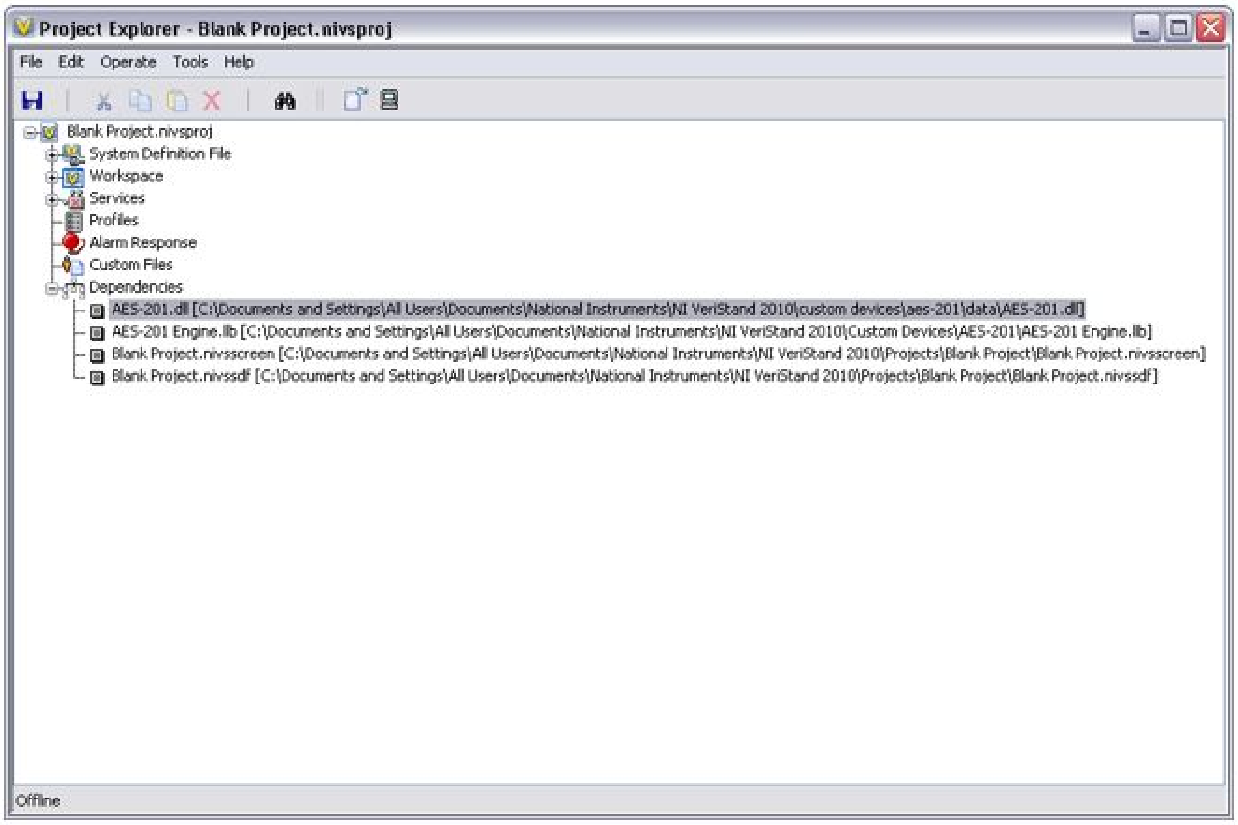
There are several other VIs in the VeriStand Dependencies VIs palette that operate on custom device dependencies. They can be found by navigating to the following locations.

- **Dependencies VIs** » **Get Custom Device Dependencies**
- **Dependencies VIs** » **Reset Custom Device Dependencies**

Add the custom device dependency to the Initialization VI.


2010
No Error
Create Hardware Inputs section and create 8 output channels, each with a property called FilterEn. Override the default channel page.
Create Hardware Enables section and create 8 input channels.
Create a single SMBTrig input channel.
Add the custom device dll dependency.
Hardware Inputs
Hardware Enables
SMBTrig
Device Item Ref in
error in (no error)
Custom XML
ADDataFromCh
ADDataFromCh GUID
FilterEn
ADEnCh
Device Item Ref out
error out
Source
Path
custom devices\aes-201\data\AES-201.dll
Type
To Common Data Dir
RT Destination
c:\ni-rt\VeriStand\Custom Devices\AES-201\AES-201.dll
ForceDownload?
F
SupportedTarget
PharLap
Version
1.0

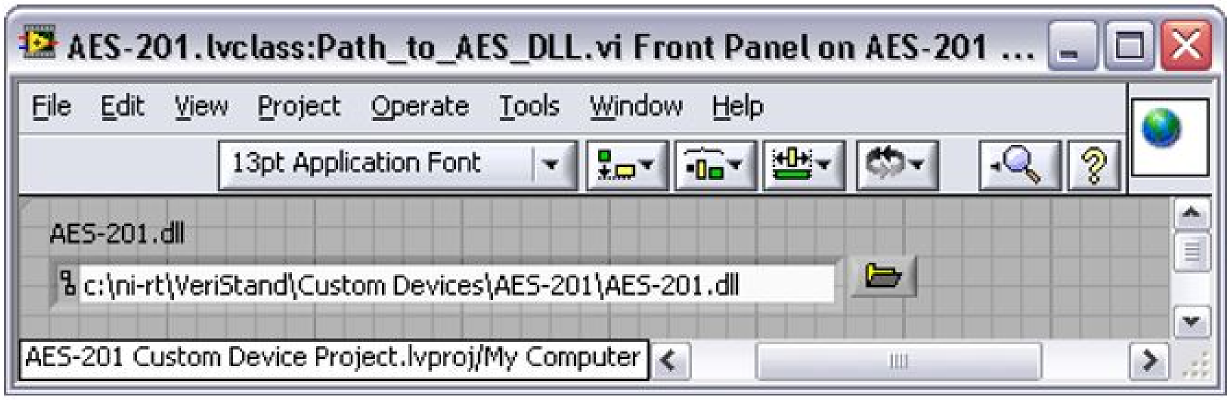
As a result, the Initialization VI adds the DLL to the project dependency list while running.


Project Explorer - Blank Project.nivsproj
File Edit Operate Tools Help
Offline

Blank Project.nivsproj
- System Definition File
- Workspace
- Services
- Profiles
- Alarm Response
- Custom Files
- Dependencies
  - AES-201.dll [C:\Documents and Settings\All Users\Documents\National Instruments\NI VeriStand 2010\custom devices\aes-201\data\AES-201.dll]
  - AES-201 Engine.llb [C:\Documents and Settings\All Users\Documents\National Instruments\NI VeriStand 2010\Custom Devices\AES-201\AES-201 Engine.llb]
  - Blank Project.nivsscreen [C:\Documents and Settings\All Users\Documents\National Instruments\NI VeriStand 2010\Projects\Blank Project\Blank Project.nivsscreen]
  - Blank Project.nivssdf [C:\Documents and Settings\All Users\Documents\National Instruments\NI VeriStand 2010\Projects\Blank Project\Blank Project.nivssdf]

You must direct the engine to the *.so* file on the target. Use either of the following methods.

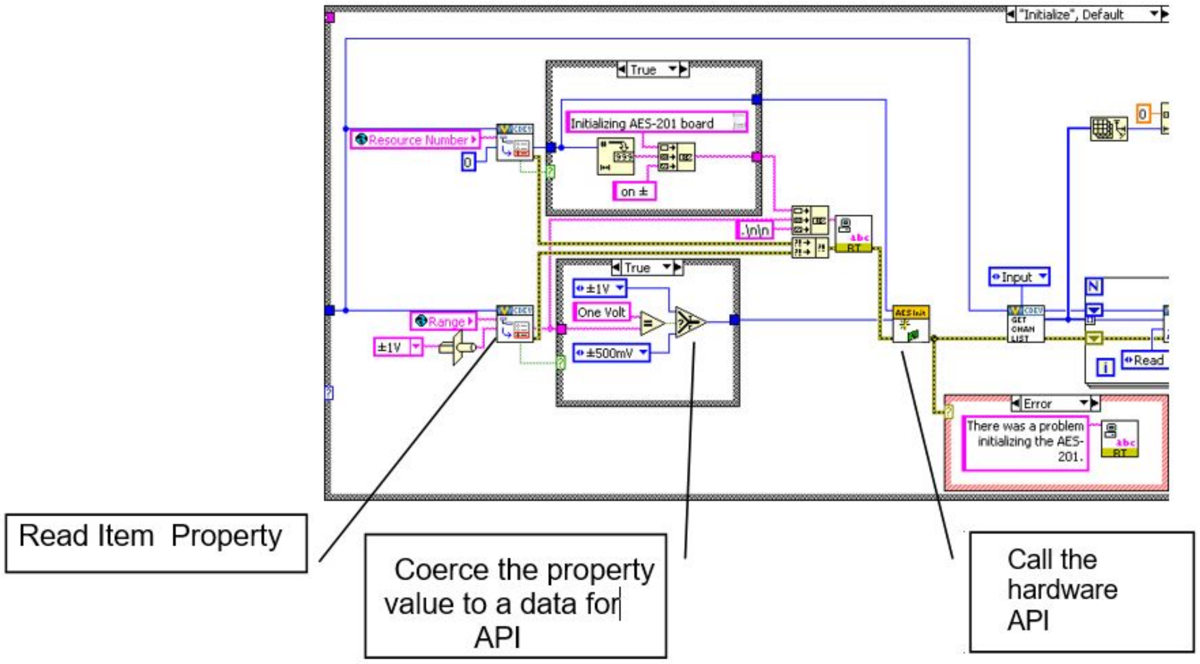
- Deploy the *.so* file to a folder in RT's search path. By default, the path is `C:\ni-rt\system`.
- **(Recommended)** Use a global variable that points to the absolute path of the DLL on the target.


AES-201.lvclass:Path_to_AES_DLL.vi Front Panel on AES-201 ...
File Edit View Project Operate Tools Window Help
13pt Application Font
AES-201.dll
c:\ni-rt\VeriStand\Custom Devices\AES-201\AES-201.dll
AES-201 Custom Device Project.lvproj/My Computer

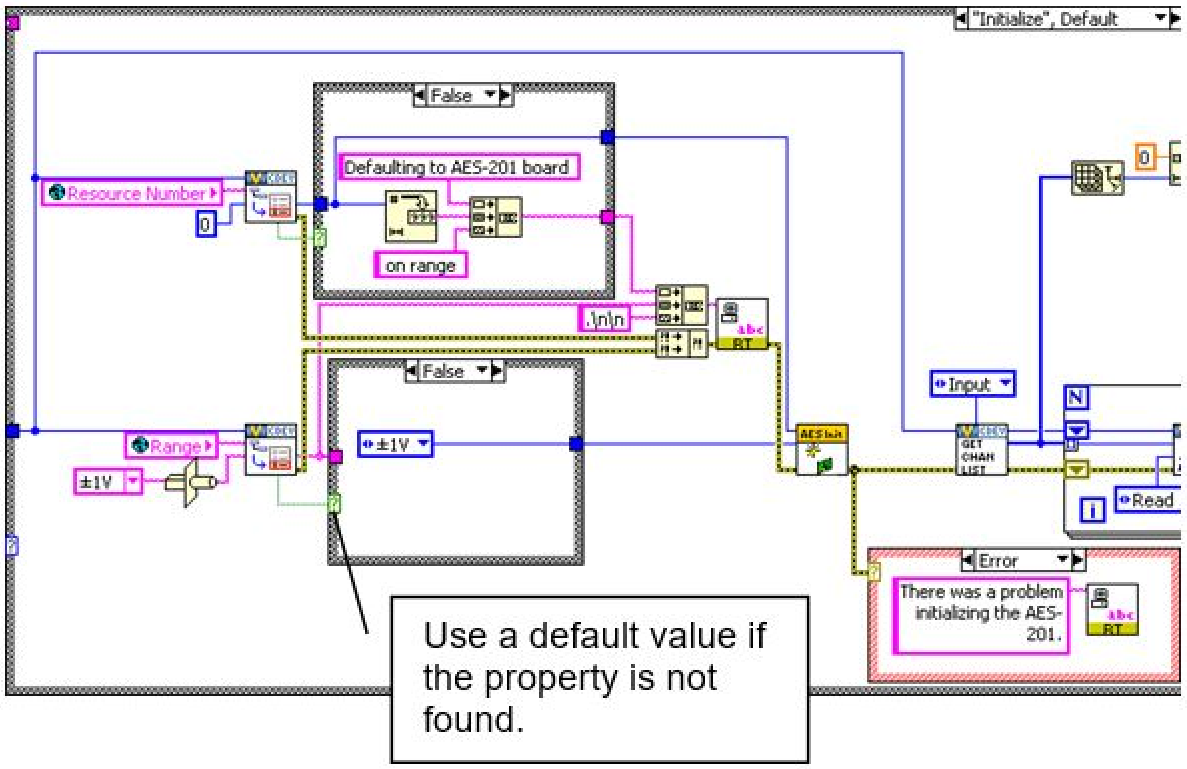
For ease of maintenance, deploy the *.so* file to `C:\ni-rt\VeriStand\Custom Devices\<Custom Device Name>\<library>.so`.

Read the range and resource number properties from the custom device reference. Recall that you must read the property from the correct item and that we set these properties to the top-level device reference.

Call the AES-201 API to initialize the board according to the property values.



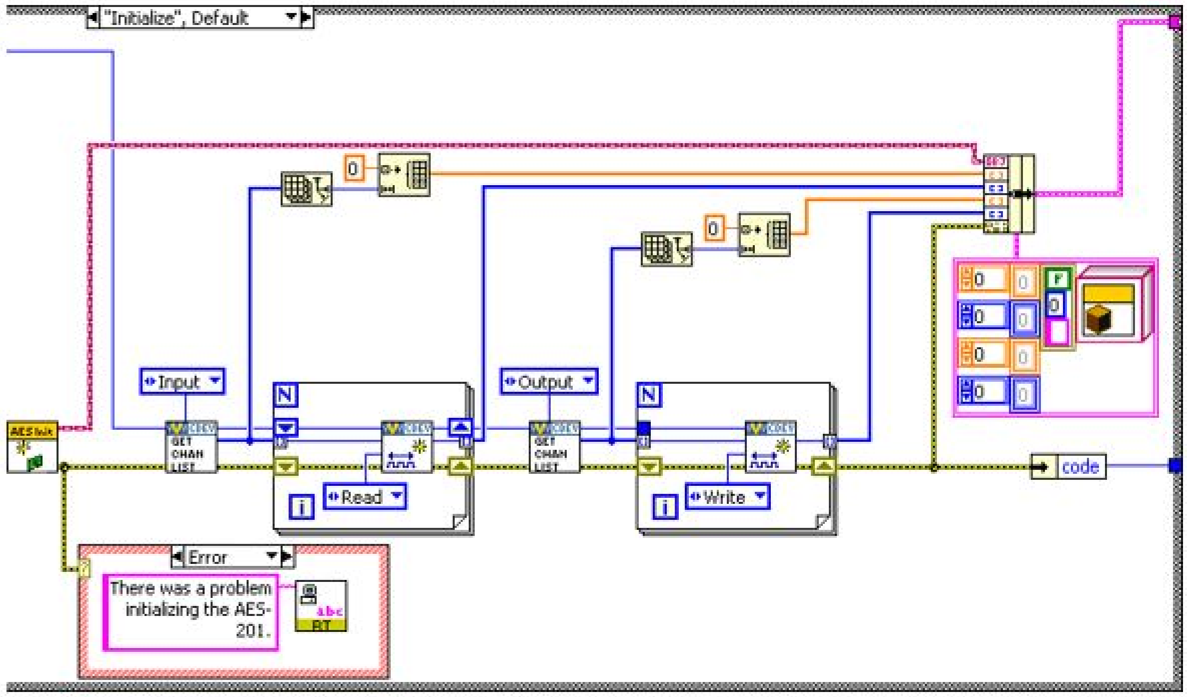
If the operator didn't trigger the event to set the property, there won't be a property to read. Instead of throwing an error, default to the value of your choice and call the API accordingly.


"Initialize", Default
False
Defaulting to AES-201 board
on range
Resource Number
0
.\n\n
RT
False
Range
±1V
±1V
AES Init
Input
GET CHAN LIST
N
Read
Error
There was a problem initializing the AES-201.
RT
Use a default value if the property is not found.

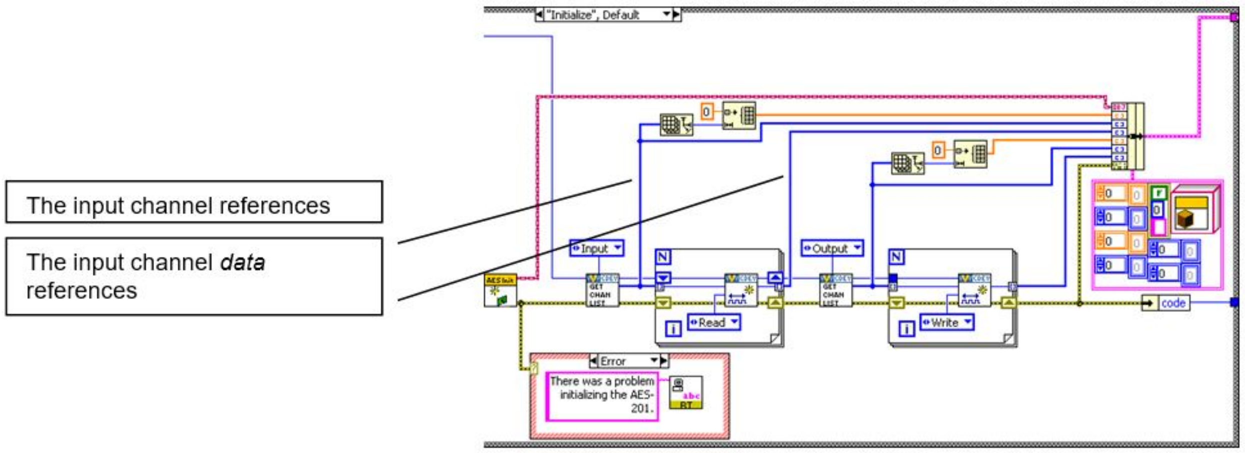
Note: You should print a few strings to the console to tell the operator what is happening.

The inline HW custom device uses a feedback node to pass state data between states. Add the AES-201 state data to the feedback node's cluster.

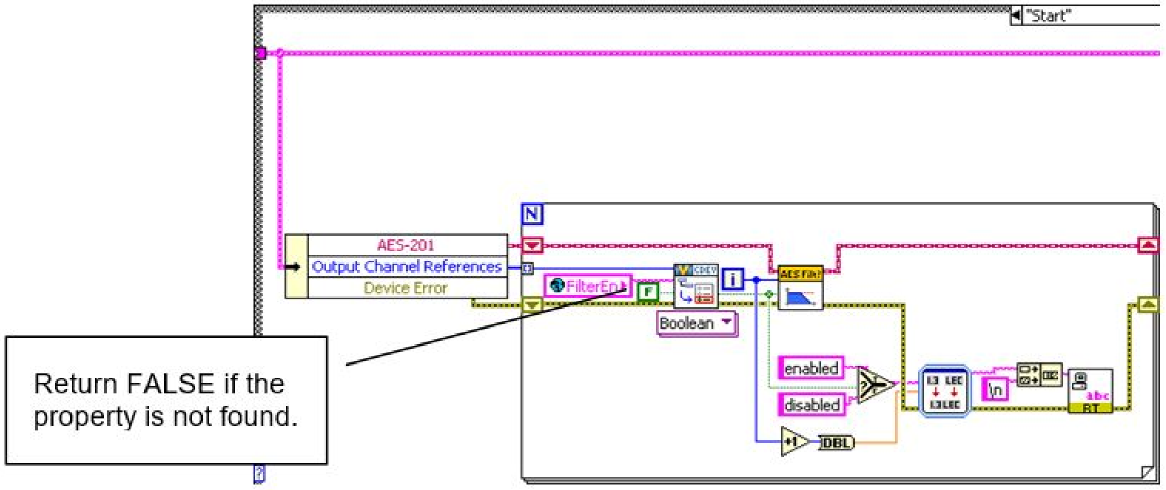
This LabVIEW object represents all the state data needed to use the AES-201 in subsequent states.



Add the input and output channel references to the state data cluster.


"Initialize", Default
The input channel references
The input channel *data* references
Input
Output
Read
Write
Error
There was a problem initializing the AES-201.
code

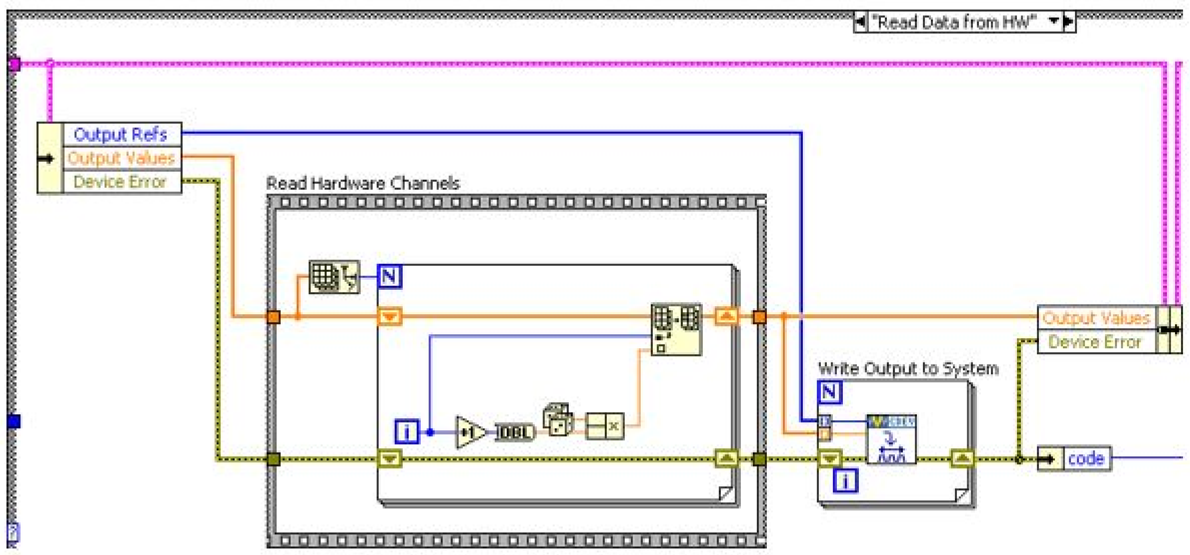
The output channels are for *ADDataFromCh<1..8>*. Check the filter property on each output channel reference and call the AES-201 API to set the filter accordingly.


"Start"
AES-201
Output Channel References
Device Error
FilterEn
Boolean
enabled
disabled
DBL
Return FALSE if the property is not found.

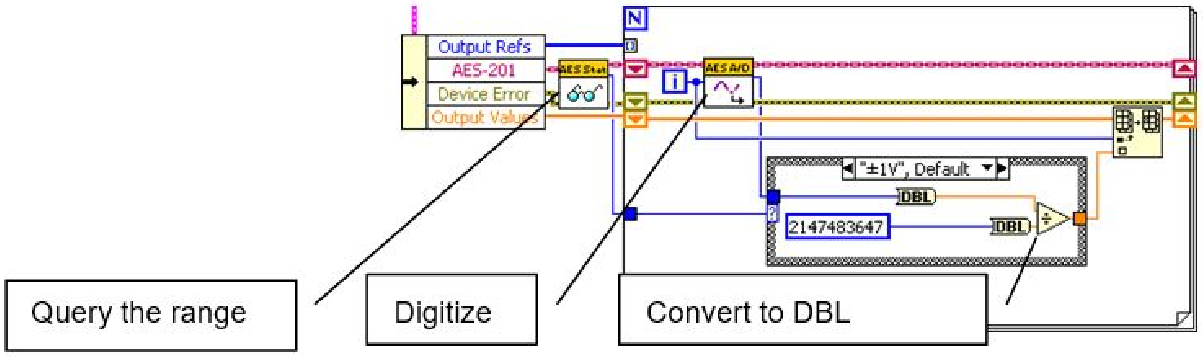
After the custom device has been configured and deployed, VeriStand will no longer exchange property information between the host computer and execution host.

Since we implemented the filter as a property, we will call the AES-201 API in the **Start** case. If the operator wants to toggle the filter, they must reconfigure the device in System Explorer.

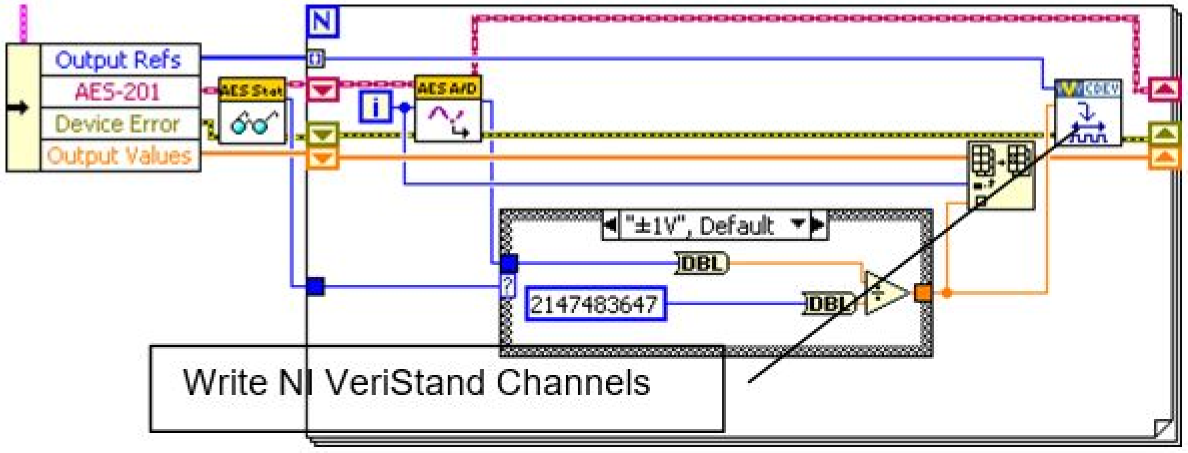
After configuring the hardware, we will request an A/D sample. For this custom device, the *Read Data from HW* case will be useful for this operation.



Replace the Read Hardware Channels frame with the API call to digitize. Convert the 32-bit raw data to DBL data, depending on the range of the AES-201.



Send the channel data to the rest of the VeriStand system by writing to the Output Reference.

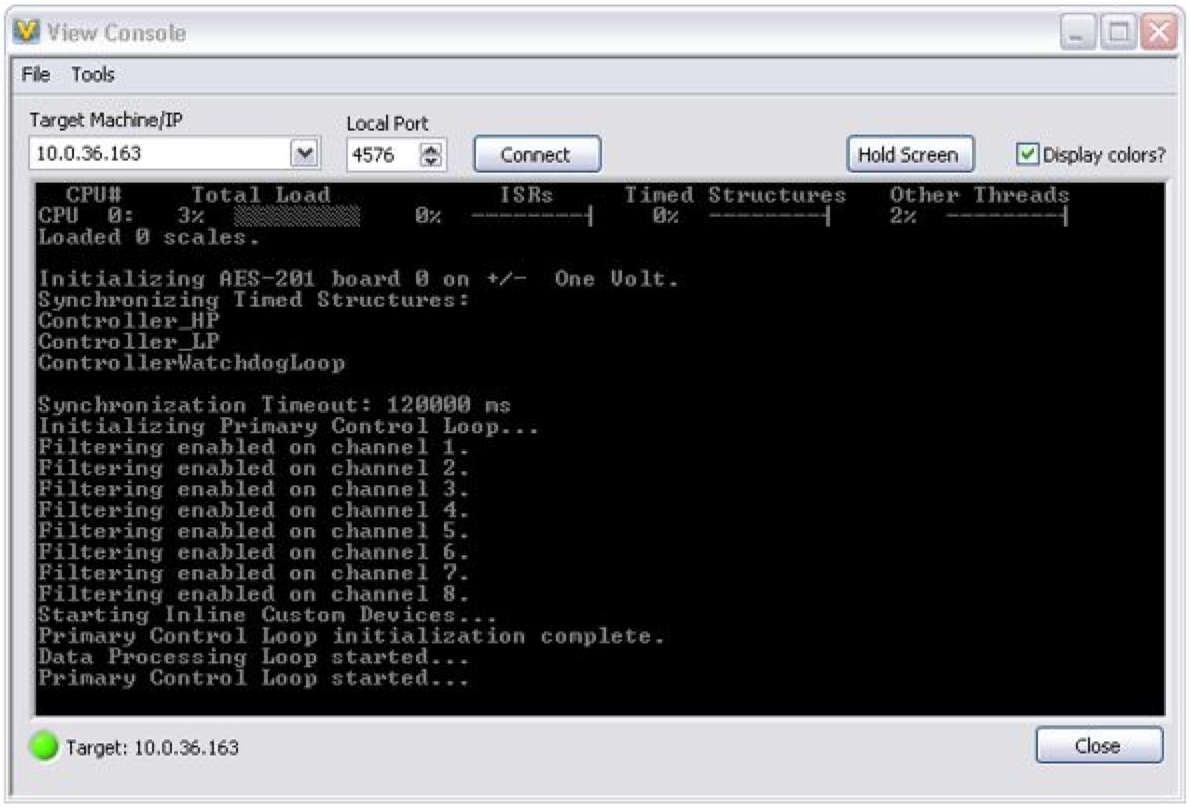


For flat hierarchies, the reference array corresponds one-to-one with channels. This is because they were created on the host computer. The first channel created is the 0th element of the array.

For non-flat hierarchies, the reference array corresponds top-down and one-to-one with channels as they were created. Channels at the highest level of the hierarchy appear first in the array, then subsequent levels channels appear in the order they were created.

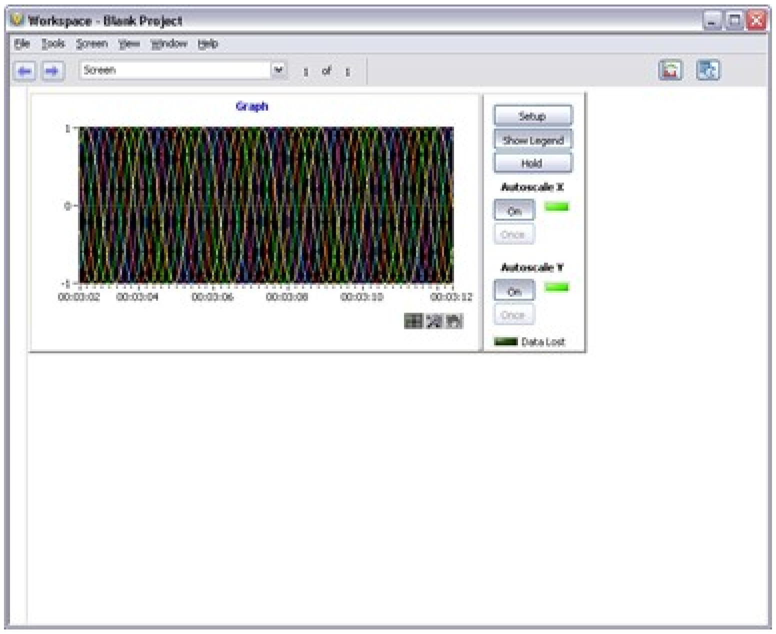
Robust custom devices do not depend on any particular order of channel references. Unique properties or GUIDs should be used to ensure the driver VI operates on the correct channel.

By default, the AES-201 inputs are enabled. You must build the custom device, enable filtering on all channels, add it to a new system definition, and deploy the project.

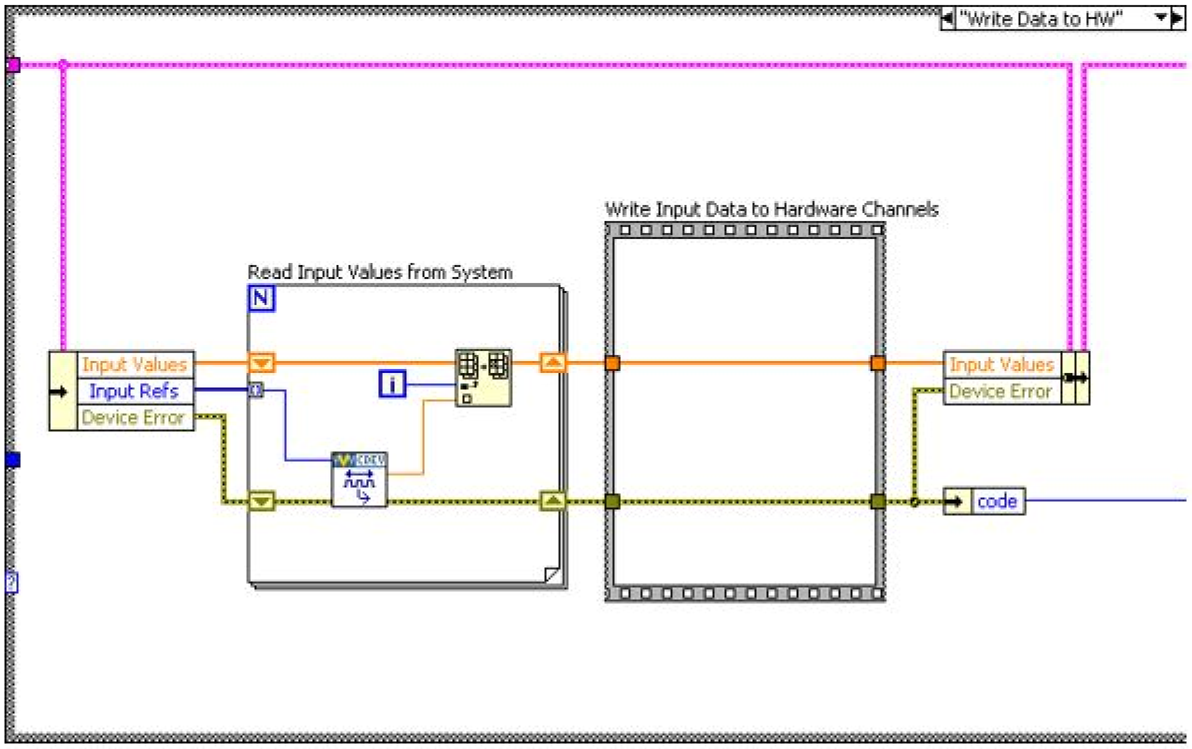


```
  CPU#     Total Load          ISRs      Timed Structures   Other Threads
CPU  0:   3%             0%  ---------|    0%  ---------|    2%  ---------|
Loaded 0 scales.

Initializing AES-201 board 0 on +/-  One Volt.
Synchronizing Timed Structures:
Controller_HP
Controller_LP
ControllerWatchdogLoop

Synchronization Timeout: 120000 ms
Initializing Primary Control Loop...
Filtering enabled on channel 1.
Filtering enabled on channel 2.
Filtering enabled on channel 3.
Filtering enabled on channel 4.
Filtering enabled on channel 5.
Filtering enabled on channel 6.
Filtering enabled on channel 7.
Filtering enabled on channel 8.
Starting Inline Custom Devices...
Primary Control Loop initialization complete.
Data Processing Loop started...
Primary Control Loop started...
```

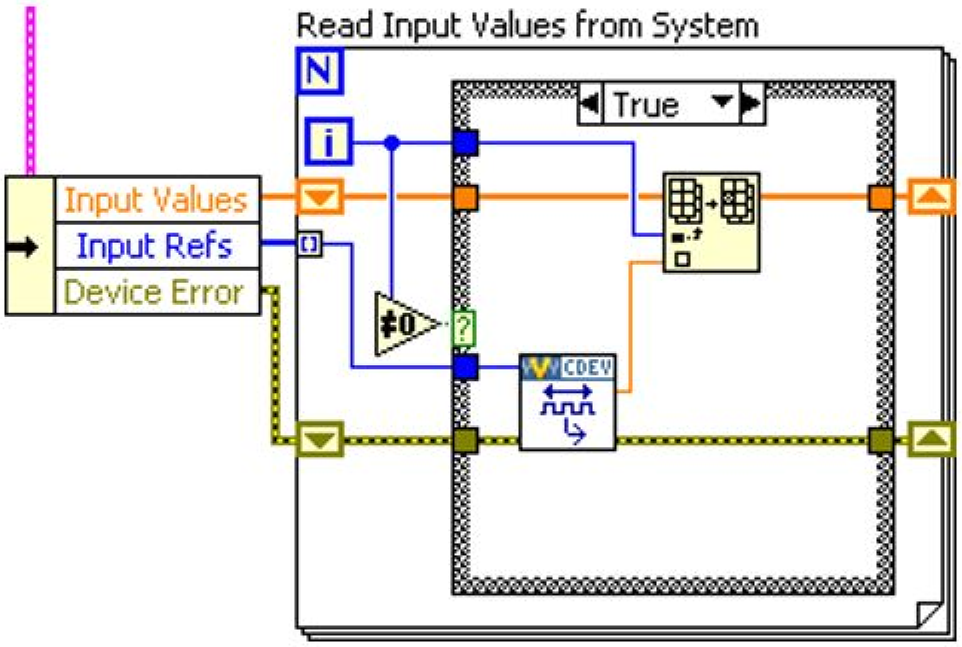
Check the console for messages to determine if the non-default configuration is active. You should also map the *ADDataFromCh<1..8>* channels to a simple graph to ensure they display the expected signals.



Now we will process the software enable channels. For this custom device, the *Write Data to HW* case is useful.

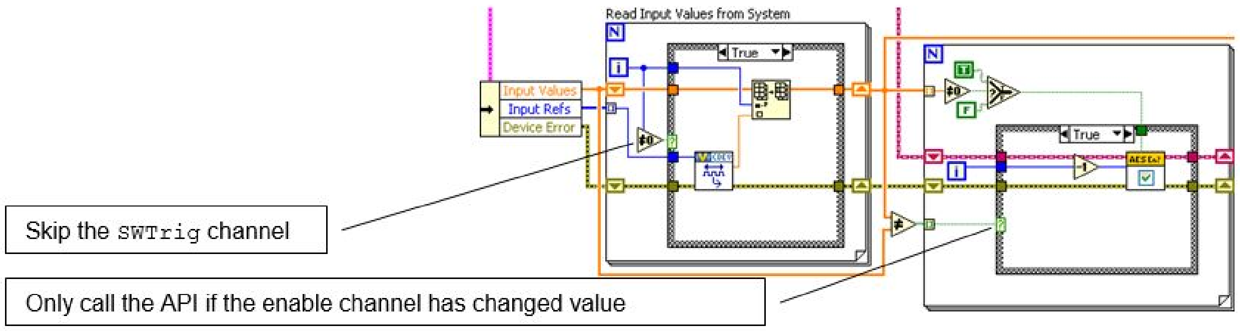


The *SWTrig* channel is higher than the *ADEnCh<1..8>* input channels in the hierarchy. Even though *SWTrig* was created last, it is the first channel in the input channel reference array.



We will skip the *SWTrig* channel reference for now and read the eight enable channels.

Make a call to the AES-201 only if the enable channel value has changed. Enable the hardware channel if the VeriStand channel does not equal zero.


Read Input Values from System
True
Input Values
Input Refs
Device Error
Skip the SWTrig channel
Only call the API if the enable channel has changed value
AES En?

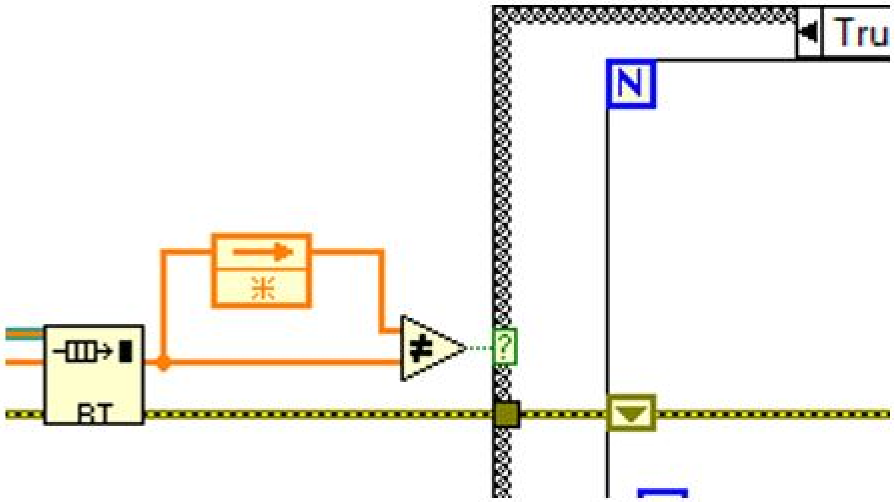
2.1.6 Channel Change Detection

You can build change detection into the custom device engine so it does not perform actions if the data has not changed. This will cause differing execution times depending on data.

Note: This is not considered jitter unless the code fails to meet determinism requirements.

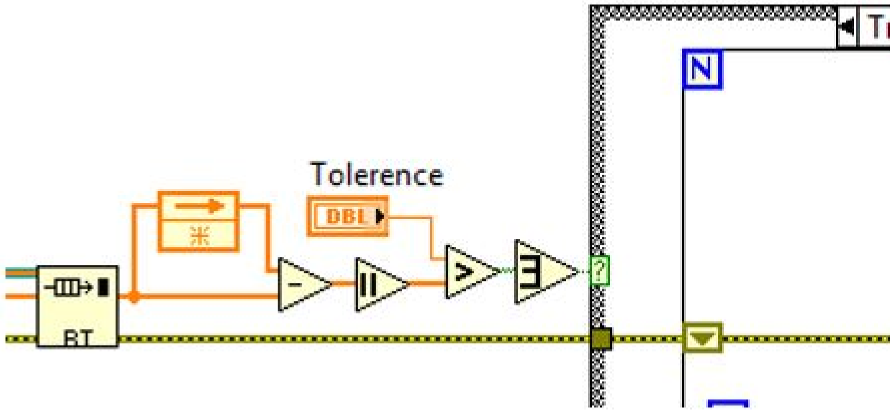
There are a variety of change detection methods. We will briefly discuss two of them. They are simple change detection and change detection with tolerance.

The following LabVIEW code is an example of simple change detection.

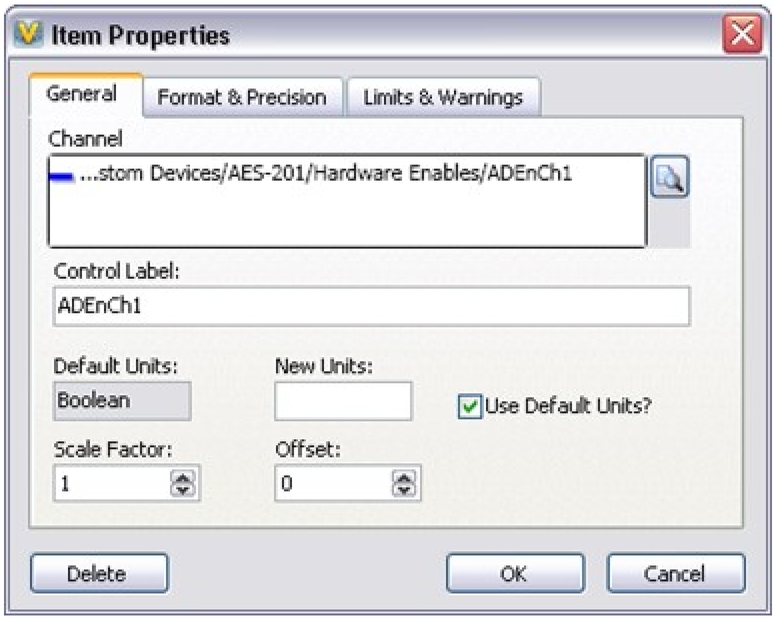


Simple change detection can fail due to floating point precision issues. Change detection with tolerance avoids these precision issues. You should use tolerances that avoid false triggers.

The following LabVIEW code is an example of change detection with tolerance.

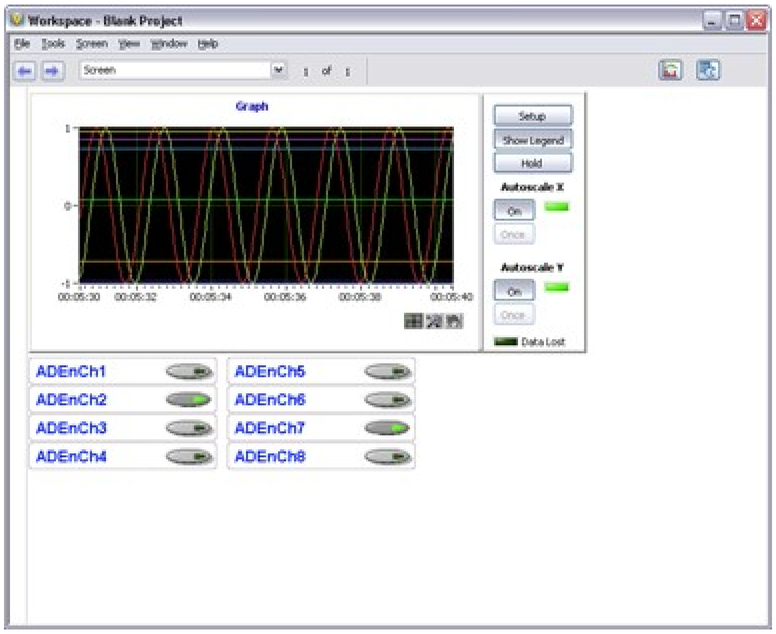


Rebuild the device and add eight Boolean controls to the workspace. Map each control to the corresponding *ADEnCh<1..8>* channel.



Channel: ...stom Devices/AES-201/Hardware Enables/ADEnCh1
Control Label: ADEnCh1
Default Units: Boolean
New Units:
Use Default Units? ✓
Scale Factor: 1
Offset: 0

You should now be able to toggle the channels on and off from the workspace. In this example, disabled channels hold the last sample.



Planning is very important. Because we thoroughly planned the AES-201 custom device before we started writing code, it was fairly straightforward to implement.

CHAPTER

THREE

DEBUGGING AND BENCHMARKING

3.1 Debugging and Benchmarking

Use LabVIEW and VeriStand to debug and benchmark custom devices as you would any other code during development.

3.1.1 LabVIEW Debugging Techniques

Custom devices are written in LabVIEW code. You can develop, test, and debug this code in LabVIEW before running the niveristand-custom-device-wizard. Use built-in LabVIEW debugging techniques during development and merge your resulting LabVIEW code into the custom device framework.

A custom device is one of many parts of the system definition. Behavior of LabVIEW code within the custom device framework, such as timing, will likely differ from a stand-alone LabVIEW application. You should benchmark the custom device inside of the VeriStand Engine.

Once custom devices are added to the system definition, they are fully integrated into the VeriStand context. Built-in LabVIEW debugging techniques will no longer be available.

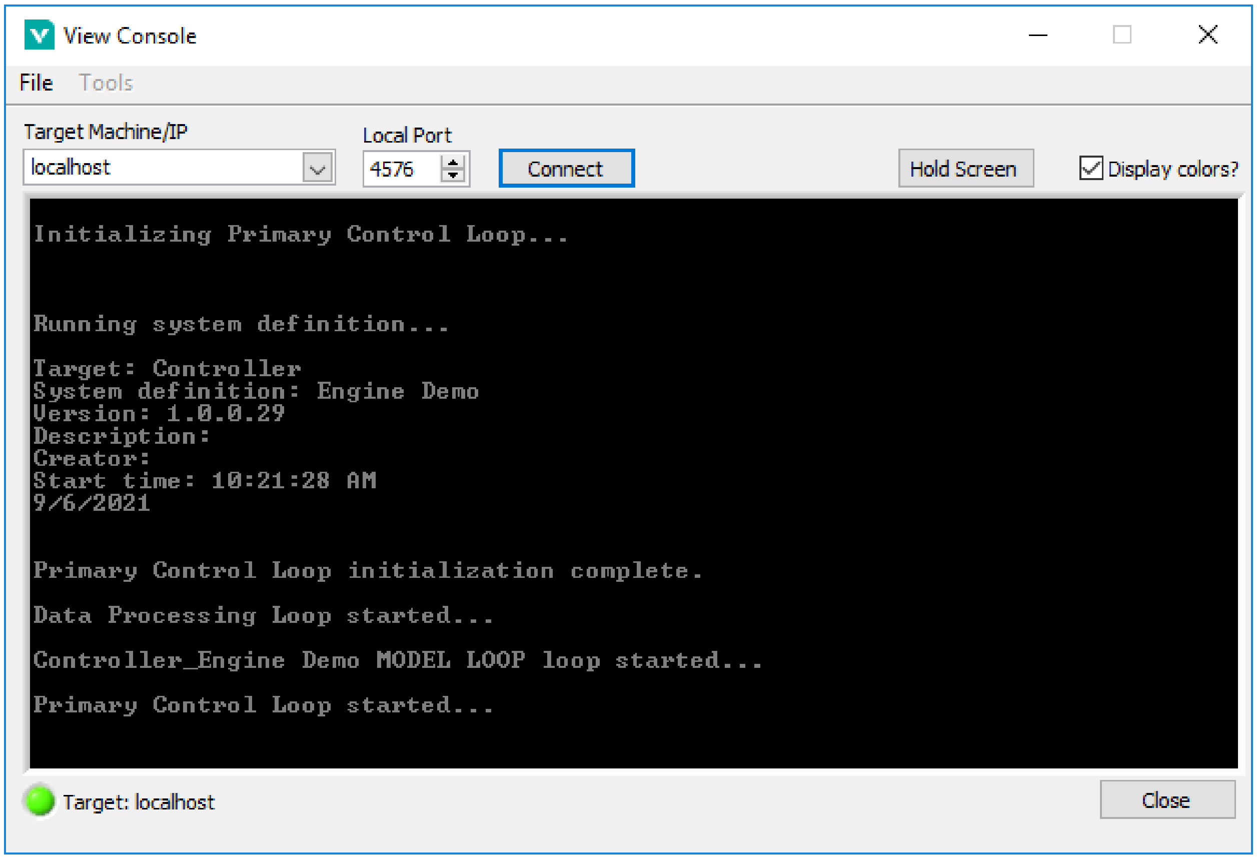
3.1.2 Console Viewer

The Console Viewer is a subcomponent of the VeriStand Real-Time (RT) Engine.

You can install the Console Viewer to the target with NI Measurement and Automation Explorer (MAX). Once installed, the component runs a small UDP daemon that allows the operator to view the console. You can access the Console Viewer from the VeriStand Editor by clicking **Tool Launcher** » **View Console**.

Note: You cannot use the Console Viewer on NI Linux Real-Time targets. Instead, connect your NI Linux Real-Time targets to a computer using a serial port to view the output.

The Console Viewer provides a periodic snapshot of the system definition and resulting CPU usage. The viewer can also display debugging messages.



```
Initializing Primary Control Loop...


Running system definition...

Target: Controller
System definition: Engine Demo
Version: 1.0.0.29
Description:
Creator:
Start time: 10:21:28 AM
9/6/2021


Primary Control Loop initialization complete.

Data Processing Loop started...

Controller_Engine Demo MODEL LOOP loop started...

Primary Control Loop started...
```

CPU spikes and transients may not be observable. If the system is busy, the Console Viewer may not update. You can use other debugging methods for a more accurate indication of resource utilization.

The Console Viewer is also available as a stand-alone add-on to LabVIEW Real-Time. For more information, refer to Remotely View Console Output of Real-Time Targets.

3.1.3 Custom Error Codes

You can define custom error codes in LabVIEW and distribute them to VeriStand with a custom device.

1. Copy a custom *errors.txt* file to VeriStand in the `<Base>\National Instruments\Shared\Errors\English` directory.
2. Add the file as a dependency in the custom device.
3. Add the file as a dependency in the custom device XML file.
4. **(Optional)** For RT targets, deploy the *errors.txt* file to the error directory on target. Error messages will display in Console Viewer.

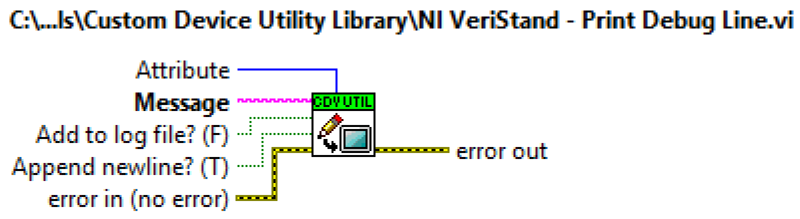
For more information, refer to Defining Custom Error Codes to Distribute throughout Your Application.

3.1.4 Printing With the Print Debug Line VI

The recommended method of printing to the console is to use the NI VeriStand - Print Debug Line VI.

This VI works on both Windows and RT execution hosts. Use the optional **Attribute** input to change the color of the text. You can also use the optional **Add to log file? (F)** input to append the string to the VeriStand log file.

To locate this VI in LabVIEW, navigate to **Custom Device API** » **Utilities**.

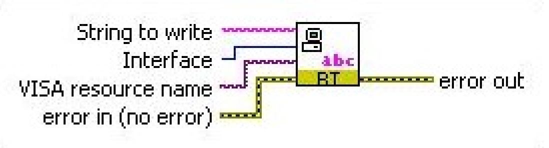


3.1.5 Printing with RT Debug String VI

The RT Debug String VI sends a string to the standard output device.

By default, this VI sends the debug string to the video port. If you have a device capable of serial redirection, the debug string is sent to the serial port.

To locate this VI in LabVIEW, navigate to **Real-Time** » **RT Utilities**.



3.1.6 Telemetry Custom Device

The Telemetry Custom Device supports VeriStand benchmarking by logging system channels and monitoring target resources. Usage data is logged to a TDMS file on the target that is running the VeriStand Engine.

3.1.7 System Channels

VeriStand includes system channels that provide information on internal processes. Several of these system channels are useful for benchmarking and debugging.

The following table contains examples of debugging and benchmarking system channels.

System Channel	Description
HP Count	The number of times the Primary Control Loop reported being late.
HP Loop Duration	The duration of the Primary Control Loop in nanoseconds.
LP Count	The number of times the Data Processing Loop reported being late.
Model Count	The number of times the models have not completed their execution in time.

If the value of the count channels increase over time, the target is not achieving the desired loop rates. You can use the system channels in conjunction with an alarm or procedure to handle events.

3.1.8 System Monitor Custom Device

The System Monitor Custom Device tracks memory resources and CPU usage on an RT target running the VeriStand Engine. Set the update rate (Hz) in System Explorer to determine how often the custom device checks CPU and memory usage and sends them to the corresponding channel.

Note: The VeriStand System Monitor can only be used on an RT target.

3.1.9 Distributed System Manager

You can use the NI Distributed System Manager (DSM) to monitor the CPU and memory resources of an RT target. You must install System State Publisher on the RT target.

This component runs a small daemon that publishes the system state to DSM. For more information, refer to Monitor RT target resources.

System State Publisher provides a periodic snapshot of utilization. CPU spikes and transients may not be observable. If the system is busy, DSM may not update. You can use other debugging methods for a more accurate indication of resource utilization.

3.1.10 Additional Debugging Options for VeriStand

Upon request, NI can provide advanced debugging tools to help you resolve certain custom device issues. These tools are a last resort when all other debugging options have been exhausted. For more information, contact NI.

CHAPTER

FOUR

DISTRIBUTING THE CUSTOM DEVICE

4.1 Distributing the Custom Device

After building, debugging, validating, and benchmarking the custom device, you need to package the device for others to use.

As with most applications, you can streamline distribution in two ways.

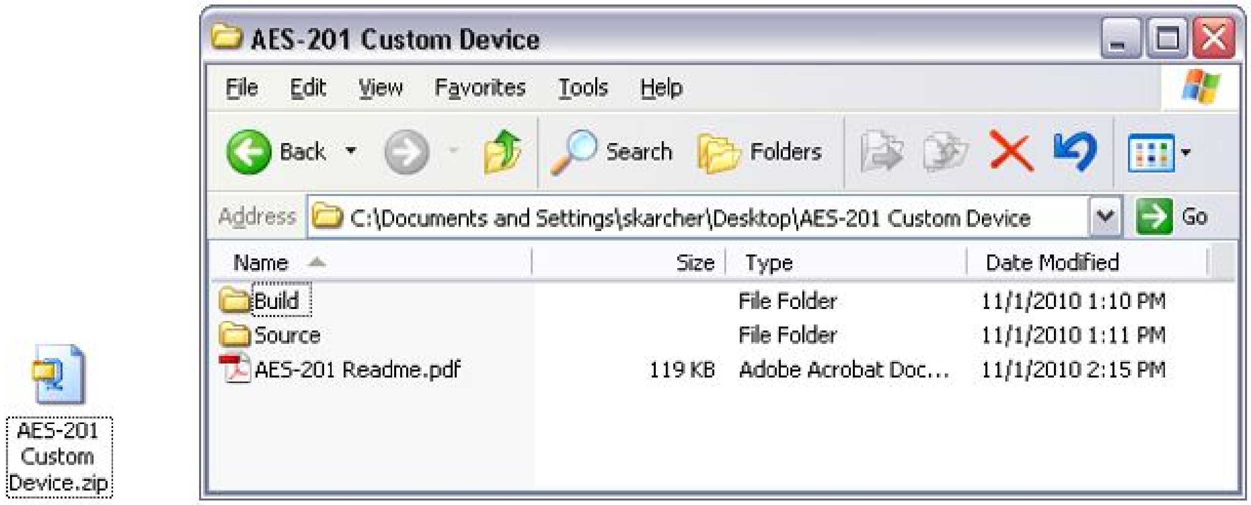
- With an Installer—For more information, refer to the *LabVIEW Help* topic (Windows) Installer Properties.
- With a Package for Distribution—For more information, refer to the *LabVIEW Help* topic Creating Packages for Distribution.

NI recommends distributing the custom device by copying the necessary files into a simple folder hierarchy. The top-level folder should contain the following items.

1. Build folder - Contains files for operator to copy to the `<Common Data>\Custom Devices\` VeriStand directory. This will add the custom device to the system definition.
2. Source folder - Contains the LabVIEW Project used to create the custom device and any supporting libraries and dependencies required to build the custom device. For example, the AES-201 custom device would ship with the LabVIEW API and hardware DLL.
3. Readme file - Contains instructions for installing, licensing, using, and modifying the custom device. The file should also contain contact information if you plan to support the device or a disclaimer if you do not plan to support the device. The Readme file is a good place to store any benchmarking information.

Note: Do not include the *Custom Device API.lvlib* files with the distribution. You do not want to replace the library on the operator's machine or change the library linking on your machine.

The following image demonstrates the distribution folder hierarchy for the AES-201 custom device.


AES-201 Custom Device.zip
AES-201 Custom Device
File Edit View Favorites Tools Help
Back Search Folders Go
Address C:\Documents and Settings\skarcher\Desktop\AES-201 Custom Device

Name	Size	Type	Date Modified
Build		File Folder	11/1/2010 1:10 PM
Source		File Folder	11/1/2010 1:11 PM
AES-201 Readme.pdf	119 KB	Adobe Acrobat Doc...	11/1/2010 2:15 PM

CHAPTER

FIVE

CUSTOM DEVICE TIPS AND TRICKS

5.1 Custom Device Tips and Tricks

Use the following information to help you develop custom devices.

5.1.1 Using Custom Device Engine Events

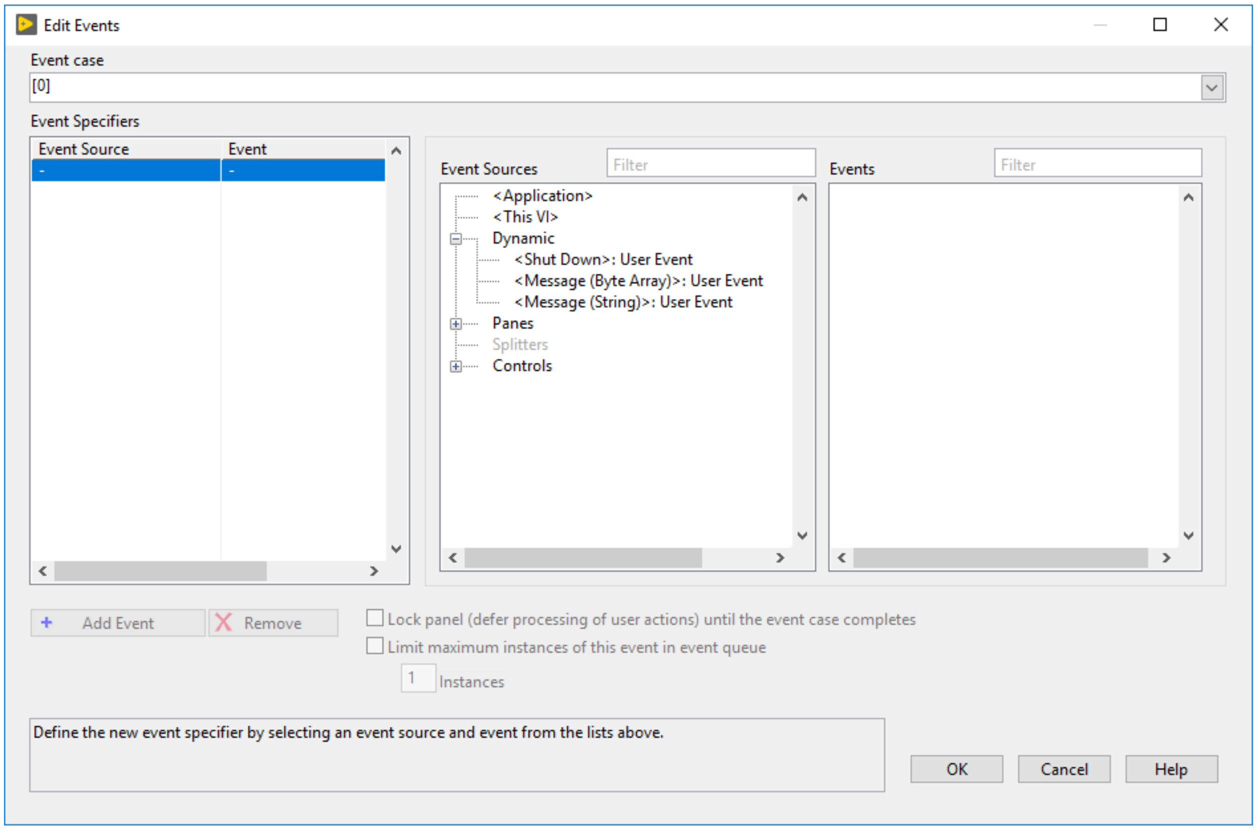
After deploying a custom device, internal channels exchange data. If the channels are insufficient or overly cumbersome, you can implement your own communication mechanism.

VeriStand provides access to its own TCP pipe. You do not have to maintain the connection. This pipe facilitates readable text and byte array data.

In LabVIEW, navigate to **NI VeriStand** » **Custom Device API** » **Driver Functions** to find *NI VeriStand - Register Custom Device Engine Events.vi*. This VI provides three dynamic events that can be registered in any VI with a reference to the custom device.

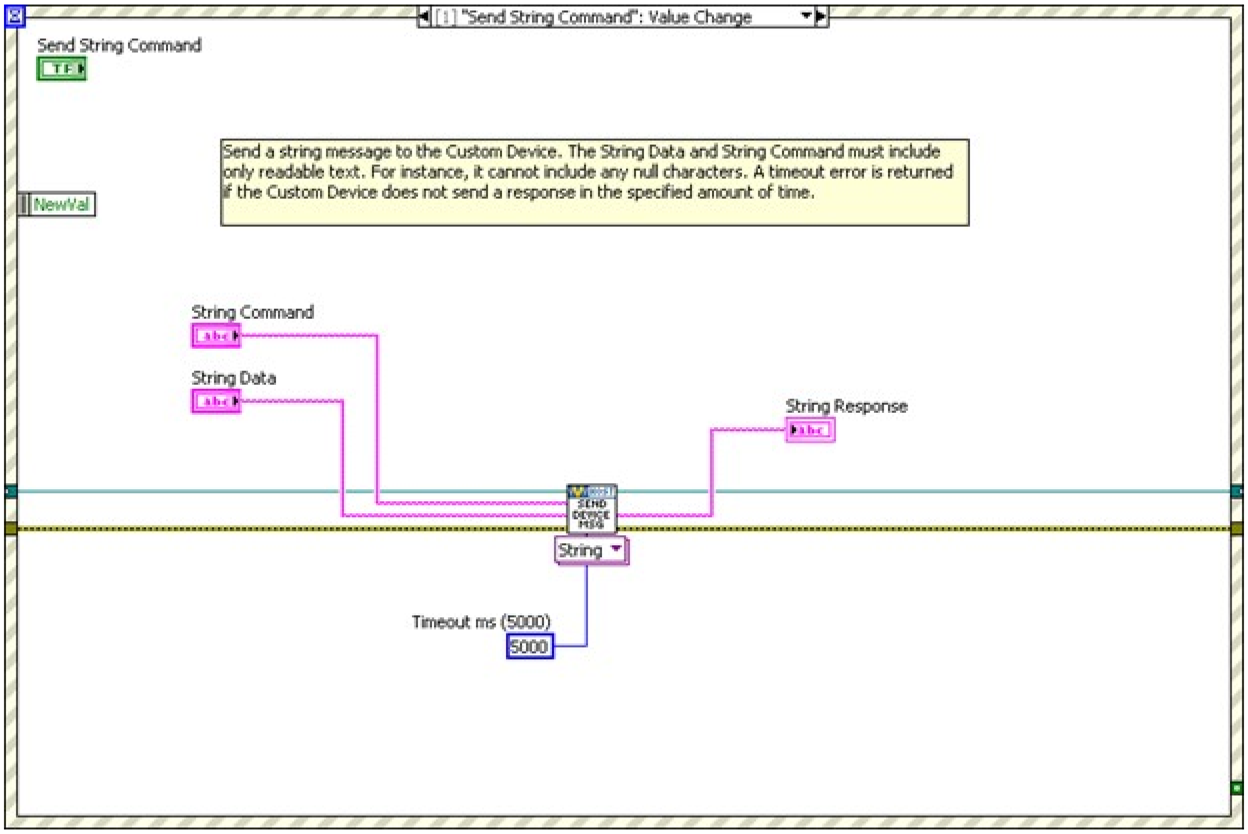
1. Shut Down
2. Message (Byte Array)
3. Message (String)

The following image displays the VI interface for registering VeriStand Dynamic Events.



Define the new event specifier by selecting an event source and event from the lists above.

The two message events activate when *NI VeriStand – Send Custom Device Message.vi* is called. The following LabVIEW code displays how to send information to VeriStand's Dynamic Message Events.



Send a string message to the Custom Device. The String Data and String Command must include only readable text. For instance, it cannot include any null characters. A timeout error is returned if the Custom Device does not send a response in the specified amount of time.

To view an example of the dynamic event pipe, navigate to the `<LabVIEW>\examples\NI VeriStand\Custom Devices\Communication Example\` directory and open *Communication Example Custom Device Project.lvproj*.

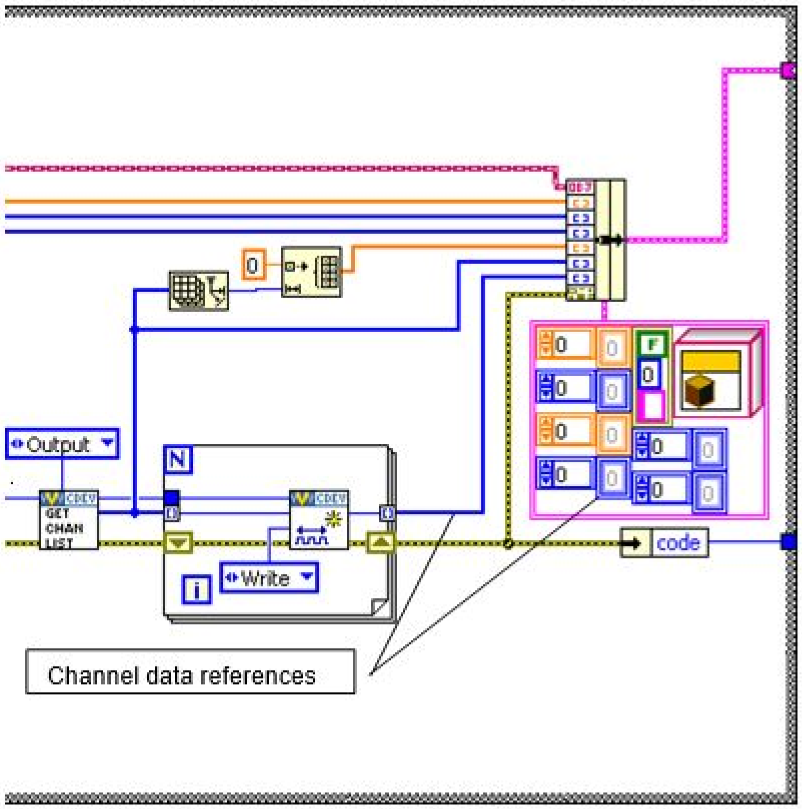
5.1.2 Processing Channel Data in Blocks

For inline hardware and inline model custom devices with many channels, it is more efficient to read and write channel data using block data references.

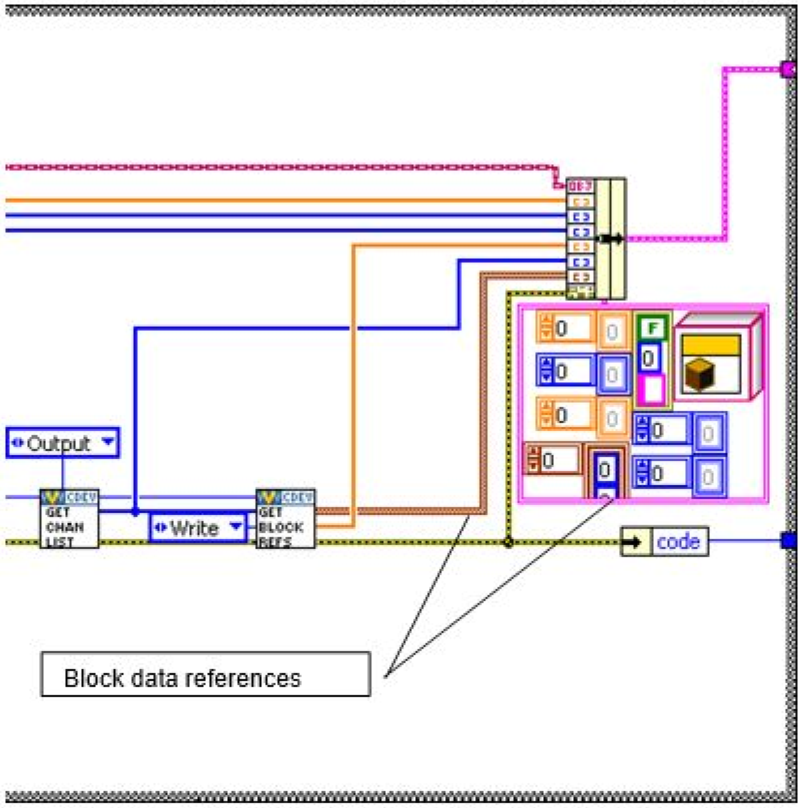
In LabVIEW, navigate to **NI VeriStand** » **Custom Device API** » **Driver Functions** » **Data References** to find the following VIs to work with block data references.

- Get Channel Block Data References
- Get Channel Values by Block Data Reference
- Set Channel Values by Block Data Reference

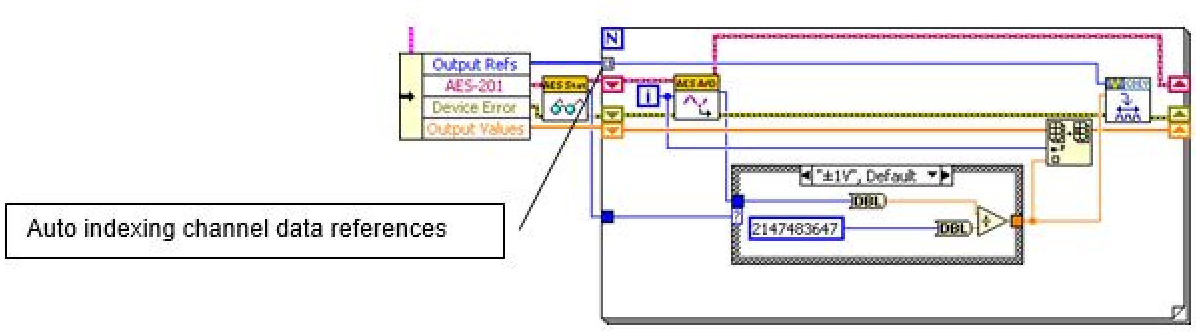
The following initialization code generates a list of output channel references.



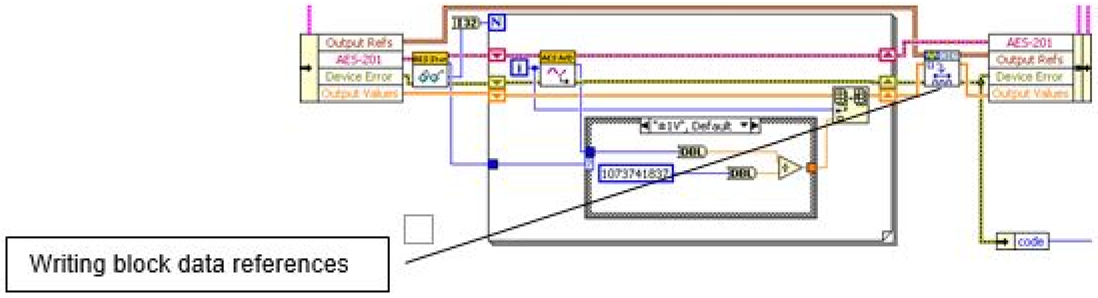
Instead of output channel references, modify the state data cluster to obtain block references to the output channels.



The original version of the custom device automatically creates an index of each channel data reference.



You should modify this code to write the block reference. In the following example, the channel block data references are written together outside the loop rather than channel-by-channel within the loop.



5.1.3 Working with String Constants

While developing custom devices, property names and GUIDs are represented as strings.

These case-sensitive strings can be difficult to use. GUIDs in particular are long and likely to produce typo errors. Use LabVIEW global variables or a type definition combo box control instead.

These alternatives have the following considerations.

- Global Variable - Ensure that you have set the correct default value for the control.
- Type Definition Combo Box - On the Properties dialog box, use the Edit Items tab to disable **Values match Items**. This control type does not auto-update from its type definition. You must completely populate the control before using it on a block diagram.

5.1.4 Creating Custom Error Codes

You can define custom error codes in LabVIEW and distribute them to VeriStand with a custom device.

1. Copy a custom *errors.txt* file to VeriStand in the `<Base>\National Instruments\Shared\Errors\English` directory.
2. Add the file as a dependency in the custom device.
3. Add the file as a dependency in the custom device XML file.
4. **(Optional)** For real-time targets, deploy the *errors.txt* file to the error directory on target. Error messages will display in Console Viewer.

For more information, refer to Defining Custom Error Codes to Distribute throughout Your Application.

5.1.5 Using Utility VIs

In LabVIEW, navigate to **NI VeriStand** » **Custom Device API** » **Utilities** for useful custom device development VIs.

For information on the VIs in this palette, refer to the context help documentation.

5.1.6 Sort Channels by FIFO Location

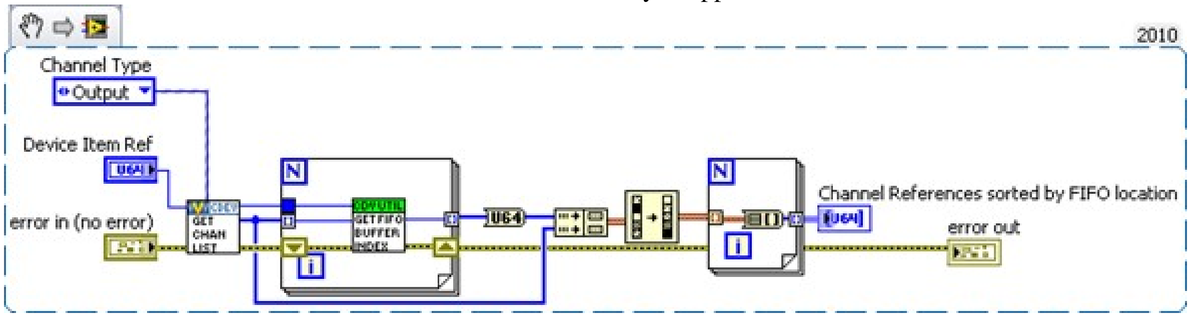
In LabVIEW, navigate to **NI VeriStand** » **Custom Device API** » **Utilities** to find the *Get Channel FIFO Buffer Index.vi*. This VI returns the FIFO buffer index for the input or output channel reference.

Use this function for Asynchronous Custom Device channels to determine what index to read or write in the FIFO arrays. The VI also returns which FIFO Buffer (Input or Output) the channel will be located in.

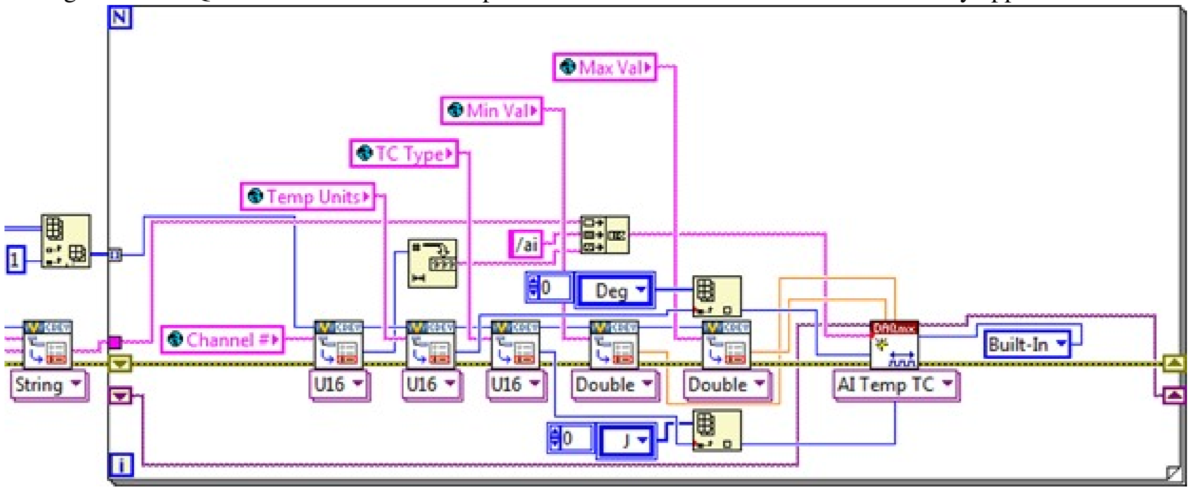
Note: This function is only intended for Asynchronous Custom Devices.

You can use this VI in a custom device to read a list of DAQmx thermocouple inputs.

1. Sort the channel references in the order they appear in the custom device FIFO.

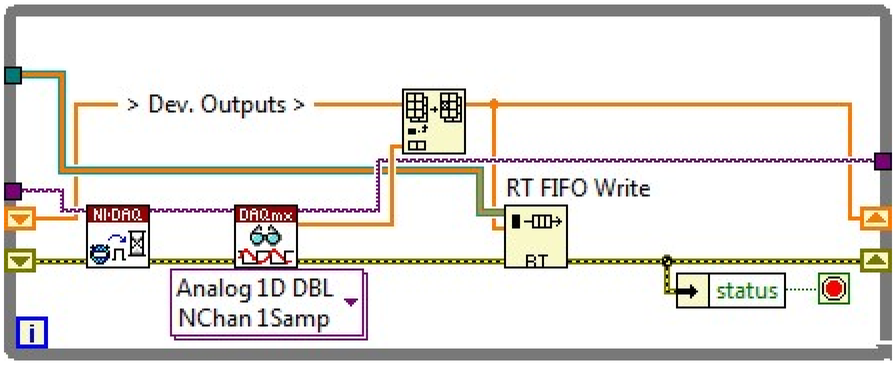


2. Configure the DAQmx task so the thermocouple channels are read in the same order as they appear in the FIFO.



There are several advantages to this architecture. The operator is free to add, remove, and reorder channels. Only the desired channels are configured. This makes writing data to the custom device FIFO efficient.

The following code writes multiple hardware channels directly to the custom device FIFO.



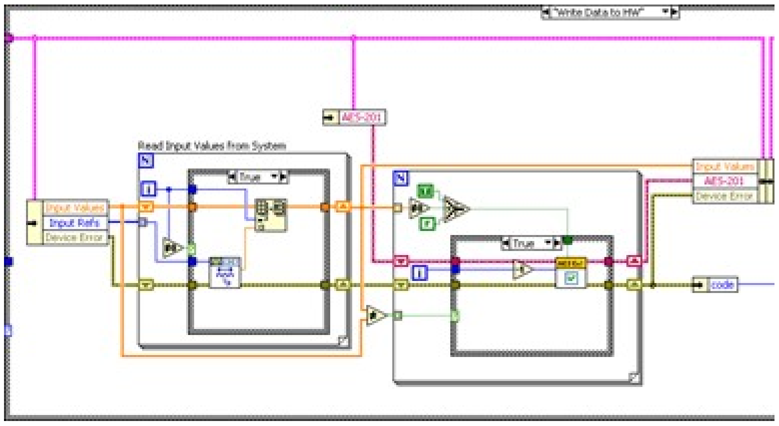
The hardware data returns from the DAQmx driver in the same order as the channel references in the asynchronous custom device FIFO.

5.1.7 Triggering Within the Custom Device

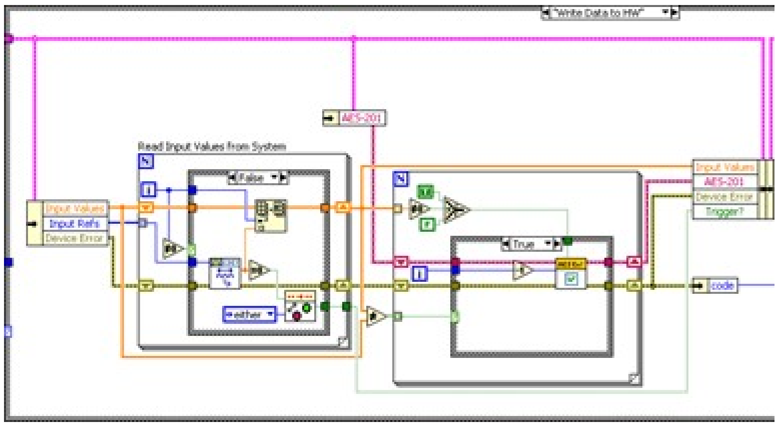
You can set up a custom device to run code when a specified event occurs. Implementing value-triggering is as simple as comparing *AEEnCh<1..8>* channel values to the previous iteration.

In LabVIEW, navigate to **Signal Processing** » **Point by Point** » **Other Functions** to find the Boolean Crossing Point by Point VI. This VI is useful for triggering events.

Recall the *Write Data to HW* state that reads VeriStand Channels. Add the following code to check the software trigger.



Check the SWTrig channel and handle any transition accordingly.



The triggering VI is useful in asynchronous custom devices that do not execute in line with the primary control loop (PCL). An asynchronous device might iterate multiple times in a single iteration of the PCL. This triggering VI will only assert on the desired edge of the transition.

5.1.8 Adding Extra Pages After Creating the Custom Device Project

If your Custom Device requires additional pages for sections or channels, you can specify their names before generating the LabVIEW project for the device.

Use the niveristand-custom-device-wizard's **Custom Device Extra Page Names** control. This tool ensures the following.

- The appropriate references are available to the page.
- The necessary declarations go into the Custom Device XML file.
- The Build Specification deploys the page to the correct location.

There are two signs that an extra page has not been added correctly to a custom device.

1. The default section or channel page loads into System Explorer instead of the expected extra page.
2. A System Explorer error message appears. `Custom Device Page Error: The following Custom Device page VI is not executable. The VI might not be found at the correct location, or it is missing dependencies that it requires to run. Please contact the Custom Device vendor for more information on this problem.`

To add a new page after the framework has been generated, you must manually perform all steps the niveristand-custom-device-wizard takes. Use LabVIEW Project Explorer to perform the operations.

1. Ensure the device gets the appropriate device reference.
2. Create the page section in the custom device XML file.
3. Modify the configuration build specification.

Note: Incorrect changes to the Custom Device's XML file can corrupt the system definition.

Ensure Device Has Appropriate Reference

The VeriStand API requires the correct Node Reference input. The VeriStand system is responsible for passing this reference to the page.

Navigate to `Custom Device API.lvlib\Templates\Subpanel Page VI\` for a Page Template VI.

Note: You can copy a page generated by the niveristand-custom-device-wizard, such as the Main page.

Create a XML Page Section

The Custom Device's XML file tells System Explorer how to load the device files.

1. In Project Explorer, open the custom device XML file.
2. Copy the information between Main Page's `<Pages>` and `</Pages>` declarations.
3. Paste the section immediately below the `</Page>` declaration that closes the *Main Page* section.
4. Update the `<eng>`, `<loc>`, and `<Path>` tags for the new page.
5. Update the `<GUID>` to match the GUID of the extra page you created.
6. Save and close the XML file.

Modify the Configuration Build Specification

The niveristand-custom-device-wizard scripts two Build Specifications that put the custom device files in the necessary format and location for System Explorer.

1. Open the Configuration Release's build specification dialog box.
2. In *Source Files*, expand the LabVIEW library for your device.
3. Make sure the new page is part of the Always Included section.
4. In *Source Files Settings*, ensure the new page in the *Project Files* tree has the destination set to **Custom Device <Name> Folder**.
5. Click **OK**.
6. Save the LabVIEW project.

You must rebuild the *Configuration Release* and *Engine Release* build specifications to deploy changes. You can then use the extra page as if it were generated by the niveristand-custom-device-wizard.

Note: The **niveristand-custom-device-wizard** is open source. You can examine the code like any other VI.

5.1.9 Updating Custom Device XML

XML Tags define the settings for a custom device. These elements, and non-standard element types, are defined in the *Custom Device.xsd* schema. You can locate this file by navigating to the `<Common Data>\Custom Devices` directory. Open the file in an XML or text editor to view the schema hierarchy.

The following example line is from the *Custom Device.xsd* file.

```
<xs:element minOccurs="0" name="ActionVIOnDelete" type="Path" />
```

The name of this tag is *ActionVIOnDelete*. Adding the tag to the custom device XML runs a VI when the operator deletes the item from System Explorer.

Note: Experimenting with an XML schema is easier in an empty custom device.

For examples on implementing XML features, refer to VeriStand's built-in components. These components are found in the `<Application Data>\System Explorer\System Explorer Definition Files` directory.

If a tag is opened, use the format `</tag_name>` to close the tag. If a tag must be specified but has no value, use the format `<tag_name />` to open and close the tag at the same time. This format has the same effect as `<tag_name>tag value</tag_name>`.

Delete Protection

Add `<DeleteProtection>true<DeleteProtection>` to any section in the custom device XML to block users from deleting that item from the System Explorer configuration tree.

Limit Max Custom Device Occurrences

Add `<MaxOccurrence>N</MaxOccurrence>` to the XML underneath the device type to limit the number of instances of a custom device in a single System Definition.

Rename Protection

Add `<DisallowRenaming>true</DisallowRenaming>` below the `<Name>` tag for any page to prevent the operator from renaming the item.

5.1.10 Using Action VIs

VeriStand contains eight action VI templates that are triggered by different actions.

The following action VI templates are provided by VeriStand in the **Custom Device API library**.

ActionVIOnLoad

Executes when VeriStand loads a custom device item into memory. This template helps create action VIs that launch background processes.

For example, if your custom device requires large amounts of data, you can customize this template to start a daemon that runs processes or gathers data in the background.

ActionVIOnDeleteRequest

Executes when a user tries to delete an item from the custom device. This template helps create action VIs that prevent a user from deleting a custom device item or warn a user of the implications of deleting a custom device item.

This template has the following unique parameters.

- **Item Ref**—A reference to the custom device item whose XML declaration calls this action VI.
- **Refs that are about to get deleted**—A 1D array of references to the items to be deleted. The 1D array will only contain one reference. Users can only delete one item at a time in System Explorer.
- **Discard reason**—An output that captures the user's reason for deleting the item.
- **Discard delete request?**—A Boolean to discard the delete request after the action VI finishes executing. If `True`, VeriStand will not delete the item. If `False`, VeriStand will delete the item.
- **Additional items to delete**—An array of references to additional items to delete. For example, if other custom device items depend on the item the user wants to delete, you can use this output to automatically delete those items as well.

ActionVIOnDelete

Executes after a user deletes an item from the custom device. You can use this template to alert users which channel mappings break when they delete the custom device item. You can also customize the template to reconfigure hardware.

For example, if the user deletes a page that specifies custom configuration data for your hardware, you can have the VI return the configuration to default settings.

ActionVIOnSystemShutdown

Executes when System Explorer closes. You can customize this template to close hardware connections or to close daemons you launch from an ActionVIOnLoad VI.

The template has the following unique parameters.

- **Device Item Ref**—Reference to the custom device item whose XML declaration calls this action VI.

- **Unload SDF?**—Indicates whether or not the system definition was unloaded. Unload SDF? is always `True`.
- **Saved?**—Indicates whether or not a user saved the system definition file before closing System Explorer.
- **Path**—Disk path to the system definition file.
- **System Explorer Shutdown?**—Indicates whether or not System Explorer closed. This parameter is always True.

ActionVIOnSave

Executes when a user saves the system definition. For example, you can customize this template to log each time the custom device is saved.

ActionVIOnDownload

Executes when a user deploys the system definition containing the custom device to a real-time target.

Note: This action VI does not execute if a user deploys the system definition to a Windows target.

This template helps create action VIs that finalize the target configuration after you deploy the system definition.

You can also customize this template to deploy any additional files or dependencies your custom device requires. For example, if your custom device reads and writes to shared variables, you can deploy those variables.

The template has the following unique parameters.

- **Device Item Ref in**—A reference to the custom device item whose XML declaration calls this action VI.
- **ftp session**—The Open FTP session used to download the system definition to the target. You can use this open session to move additional files to the target.
- **System Definition Dir**—A path to the system definition file on disk.
- **IP Address**—The IP address of the target.
- **ftp session out**—An open FTP session used to download the system definition file to the target.

ActionVIOnPaste

Executes when a user pastes a custom device item. This template helps create action VIs that check channel properties. For example, if the user pastes a page that configures a target, you can create an action VI to ensure that the new page does not attempt to reconfigure the target.

You can also customize this template to prompt a user to enter new values for the pasted item. For example, if a user pastes a page that will conflict with existing pages, you can prompt the user to enter new values for the page.

The template has the following unique parameters.

- **Ptr in**—A reference to the custom device item whose XML declaration calls this action VI.
- **Parent**—A reference to the parent of the custom device item whose XML declaration calls this action VI.
- **All Ptrs**—An array of references to the items the user pasted. You can only select one item to copy. This array only contains one reference that matches the Ptr in reference.

ActionVIOnCompile

Executes when VeriStand compiles the system definition file.

Note: If you deploy, undeploy, and redeploy a system definition without making changes, this template does not execute.

You can customize this template to finish configuring your hardware. The system definition file compiles when a user deploys the system definition. This means you can configure your hardware based on the final settings from the system definition.

You can also customize the template to quickly gather host-side settings. For example, often the custom device RT Engine VI uses properties set in the system definition. You can customize this template to read the values on the host side. This is faster than reading them from the real-time target.

You can then gather the properties into a single cluster, convert that cluster to a data variant, and write the variant as a single item property.

5.1.11 Adding Toolbar buttons

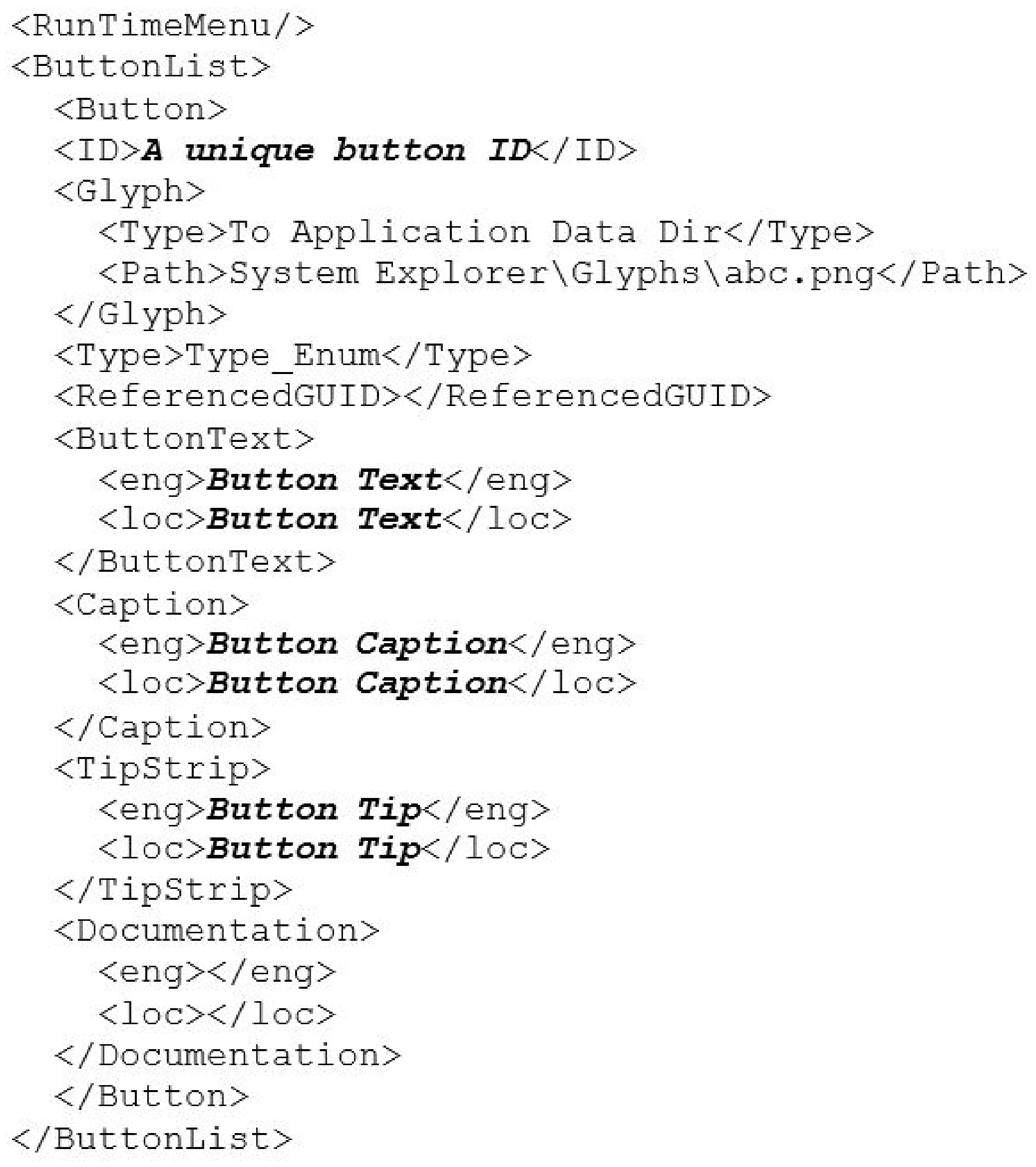
A *Toolbar button* appears in the toolbar of **System Explorer**. These buttons only appear when displaying the configuration page associated with the button.

Within the `<Page>` tags for an item, you can use the `<ButtonList>` tag to configure the toolbar buttons that appear with the item's configuration page. Each `<Button>` must include a unique `<ID>` string that identifies the button. The toolbar button displays by default.

In each page VI, you can use the Disable Dynamic Button VI and the Enable Dynamic Button VI to dynamically disable and enable a button for that page based on its unique ID. These VIs are useful when you want the toolbar button to appear only when certain conditions are true.

These VIs are located in the `labview\vi.lib\NI VeriStand\Custom Device API` directory.

The following XML schema is an example framework you can use to implement a toolbar button.



```
<RunTimeMenu/>
<ButtonList>
  <Button>
  <ID>A unique button ID</ID>
  <Glyph>
    <Type>To Application Data Dir</Type>
    <Path>System Explorer\Glyphs\abc.png</Path>
  </Glyph>
  <Type>Type_Enum</Type>
  <ReferencedGUID></ReferencedGUID>
  <ButtonText>
    <eng>Button Text</eng>
    <loc>Button Text</loc>
  </ButtonText>
  <Caption>
    <eng>Button Caption</eng>
    <loc>Button Caption</loc>
  </Caption>
  <TipStrip>
    <eng>Button Tip</eng>
    <loc>Button Tip</loc>
  </TipStrip>
  <Documentation>
    <eng></eng>
    <loc></loc>
  </Documentation>
  </Button>
</ButtonList>
```

5.1.12 Adding shortcut menus

A *shortcut menu* for an item appears when you right-click the item in System Explorer.

Within the <`Page`> tags for an item, you can use the <`RunTimeMenu`> tag to configure the shortcut menu for the item. Each <`MenuItem`> you add under <`RunTimeMenu`> includes an <`Item2Launch`> section. This section specifies a VI to run when an operator selects the menu item.

The Custom Device API library includes a template for this VI. Navigate to the `labview\vi.lib\NI VeriStand\Custom Device API` directory and open *RunTimeMenu Custom Item 2 Launch.vit*.

The following XML schema is an example framework you can use to implement a shortcut menu.

```
</Item2Launch>
<RunTimeMenu>
  <MenuItem>
    <GUID>GUID</GUID>
    <Type>Type_Enum</Type>
    <Execution>Execution_Enum</Execution>
    <Position>Position_Enum</Position>
    <Behavior>Behavior_Enum</Behavior>
    <Name>
      <eng>Extra Page Name</eng>
      <loc>Extra Page Name</loc>
    </Name>
    <Item2Launch>
      <Type>To Common Doc Dir</Type>
      <Path>...\Configuration.llb\Extra Page Name.vi</Path>
      </Item2Launch>
    </MenuItem>
</RunTimeMenu>
```

5.2 Scripting APIs

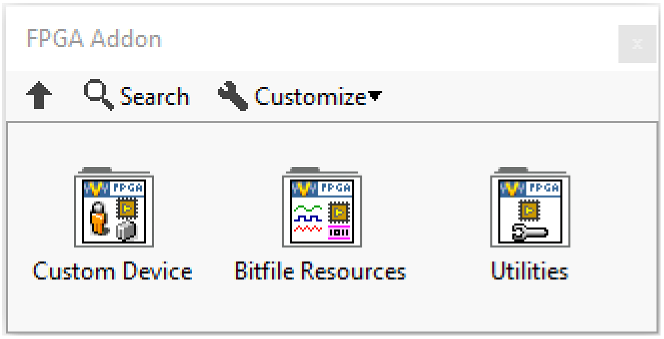
Scripting APIs use LabVIEW VIs to configure custom devices in a VeriStand system definition file.

Depending on how much functionality was made available through scripting, users can change basic settings or fully configure a custom device. You can develop a scripting API to create flexible and reusable system definition files.

5.2.1 NI Supported Scripting APIs

NI has developed scripting APIs for some custom devices.

One example is the FPGA Addon Scripting API. The following image displays how the API appears in LabVIEW.



The following custom devices also have a scripting API.

- Instrument Addon Scripting API
- Routing and Faulting Scripting API
- Engine Simulation Toolkit Scripting API

- Communications Bus Template Scripting API
- Ballard ARINC 429 Scripting API
- Ballard MIL-STD-1553 Scripting API

5.3 Migrating from LLB to PPL

A *packed project library* (PPL) is a compiled LabVIEW library (*.lvlib*) that contains the VIs of that library. PPLs allow you to call public VIs similarly to their use in the original project library.

PPLs were first introduced in LabVIEW 2010. From LabVIEW 2017 onward, you can compile a PPL for newer versions of LabVIEW. This option is enabled by default.

Note: To change this setting, in the Packed Library build specification, click **Advanced** and disable **Allow future versions of LabVIEW to load this packed library**.

Using a PPL to package a VeriStand custom device does not block the use of a *LabVIEW library* (LLB). The two options can coexist as a part of the same project. To use both packaging methods, retain the existing LLB build specifications and add the additional PPL build specifications to the project.

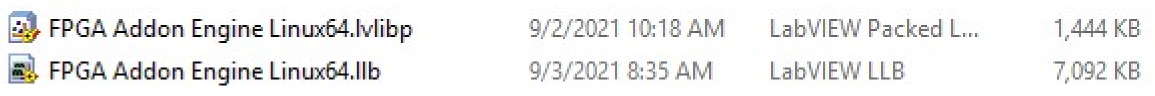
To support PPL and LLB packaging, you may need to make the following modifications to your custom device:

- Update the custom device source code.
- Update the XML configuration file to reflect the new loading paths for the custom device elements.
- Include mutation code for backwards compatibility of custom data types.

5.3.1 Benefits of Using PPL Based Custom Devices

A PPL provides namespacing for all contained items and preserves the file hierarchy of the source project library. These features result in several benefits for a custom device:

- Grants each packaged custom device a copy of shared VI dependencies. These copies are compiled and included in the PPL through namespacing. The PPL also avoids load-time conflicts.
- Avoids internal naming conflicts by preserving the file hierarch. LLBs could generate compile-time warnings and extra files. You can include multiple LabVIEW project libraries in the same PPL, even if the libraries contain items with the same name. This also allows for the easier use of LabVIEW classes when building custom devices.
- Improves deployment time through a smaller disk footprint. The following image demonstrates the relative size difference between PPL and LLB packaging of the same FPGA Addon Engine.



FPGA Addon Engine Linux64.lvlibp	9/2/2021 10:18 AM	LabVIEW Packed L...	1,444 KB
FPGA Addon Engine Linux64.llb	9/3/2021 8:35 AM	LabVIEW LLB	7,092 KB

5.3.2 Drawbacks of Using PPL Based Custom Devices

Converting a custom device to PPL comes with a number of manageable drawbacks.

- Constant relative path updates
- Incompatible LabVIEW features
- RT Driver VI path errors
- Deployment errors

Constant Relative Path Updates

Relative paths for PPL items start from the most common directory on disk. Changing the organization or disk location of an item in the library build specification can affect the relative path. Changes within the custom device folder structure can also alter the organization of the items.

Each change will require an update to the configuration XML file to include the newest paths. To resolve this issue, refer to the Update the XML.

Incompatible LabVIEW Features

Some LabVIEW features are not compatible with PPL. For example, public malleable VIs (VIMs) cannot be included in a PPL for export. Only VIMs set with a private scope can be included. To resolve this issue, remove these features from your custom device at the code or project level.

RT Driver VI Path Errors

Opening a PPL based custom device in VeriStand will generate invalid path errors for the RT Driver VIs. VeriStand System Explorer does not recognize PPL paths. These errors do not affect how the Custom Device runs. To resolve this issue, refer to the Update VeriStand System Explorer.

Deployment Errors

Some custom device APIs, such as the NI VeriStand Custom Device Channel APIs, use Global Data References. This will cause deployment errors for a PPL based custom device. To resolve this issue, refer to the Update Global Data References.

5.3.3 Implementing a PPL Based Custom Device

To create a new PPL based Custom Device, use the VeriStand Custom Device Wizard to generate a template project.

5.3.4 Migrating an LLB Based Custom Device to PPL

To migrate to a PPL based custom device, you must perform the following actions.

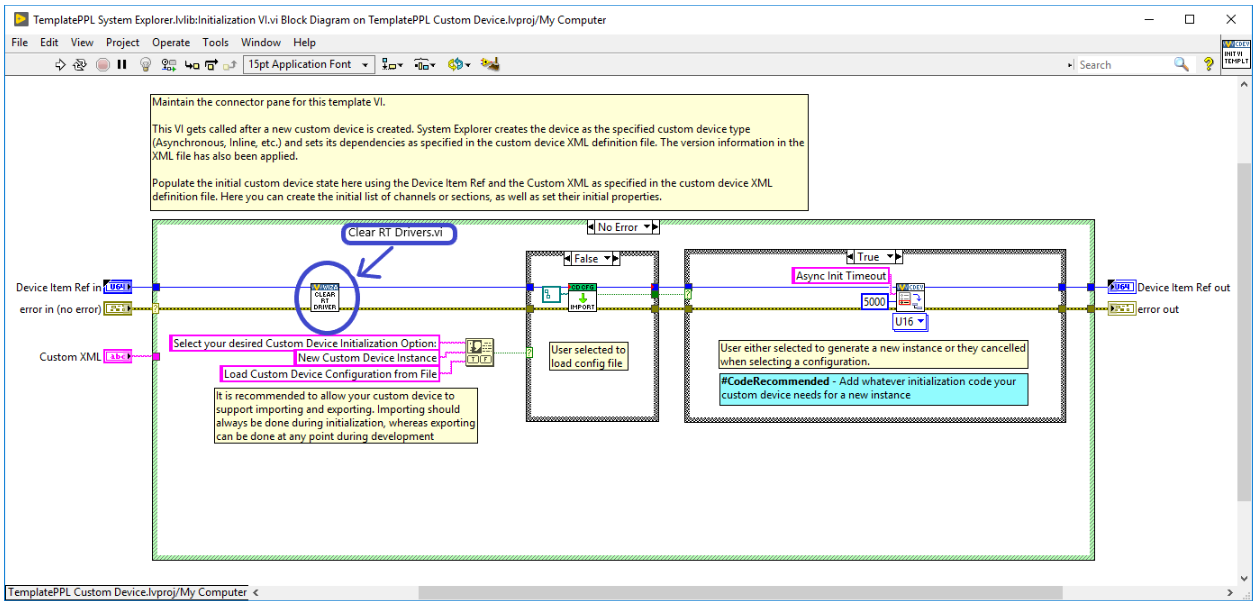
1. Update VeriStand System Explorer
2. Update Global Data References
3. Update Libraries
4. Update the XML
5. Build the PPL File

Update VeriStand System Explorer

To remove RT Driver path errors, update the *Initialization* and *Action on Compile* VIs.

Inside the Initialization VI, add a *Clear RT Drivers* subVI to delete all RT Driver paths from the system definition after the custom device is created. The paths can be removed because they are only used during and after deployment.

The following image displays where to insert the subVI.


TemplatePPL System Explorer.lvlib:Initialization VI.vi Block Diagram on TemplatePPL Custom Device.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
Clear RT Drivers.vi
No Error
False
True
Async Init Timeout
5000
U16
Device Item Ref in
error in (no error)
Custom XML
Device Item Ref out
error out
Select your desired Custom Device Initialization Option:
New Custom Device Instance
Load Custom Device Configuration from File
TemplatePPL Custom Device.lvproj/My Computer

Maintain the connector pane for this template VI.

This VI gets called after a new custom device is created. System Explorer creates the device as the specified custom device type (Asynchronous, Inline, etc.) and sets its dependencies as specified in the custom device XML definition file. The version information in the XML file has also been applied.

Populate the initial custom device state here using the Device Item Ref and the Custom XML as specified in the custom device XML definition file. Here you can create the initial list of channels or sections, as well as set their initial properties.

It is recommended to allow your custom device to support importing and exporting. Importing should always be done during initialization, whereas exporting can be done at any point during development

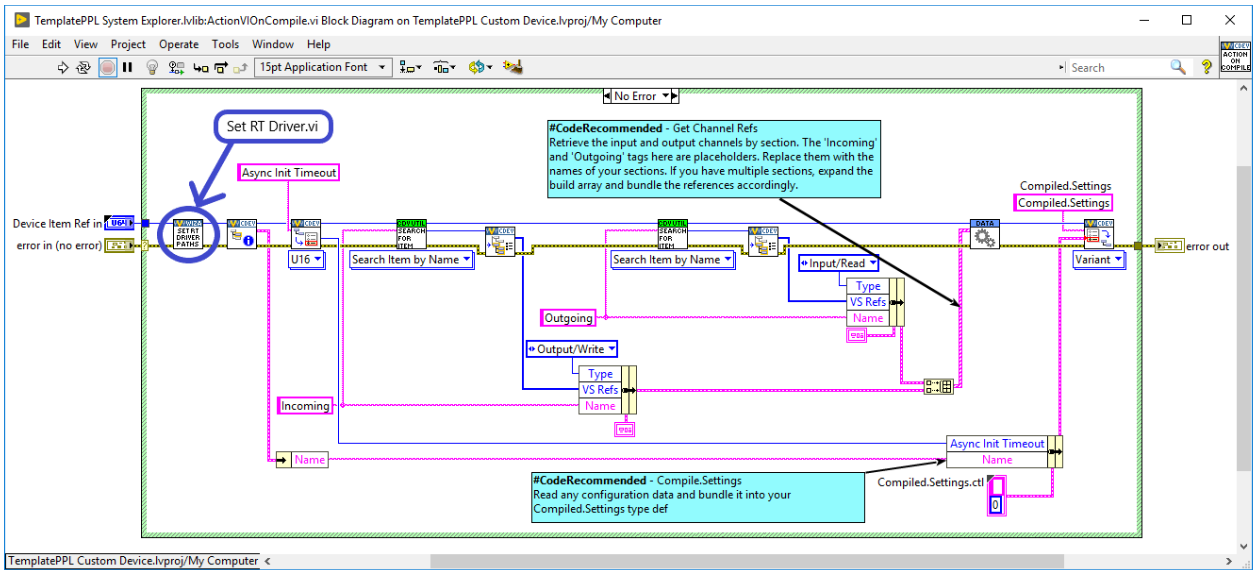
User selected to load config file

User either selected to generate a new instance or they cancelled when selecting a configuration.

#CodeRecommended - Add whatever initialization code your custom device needs for a new instance

Inside the Action on Compile VI, add a *Set RT Driver* subVI to re-insert the corresponding RT Driver paths before system deployment. VeriStand keeps a copy of the deployed system definition in the local cache unless the system definition is unmodified between deployments. The latest changes are preserved, allowing VeriStand to use the "complete" system definition from the local cache during deployment.

The following image displays where to insert the subVI.



#CodeRecommended - Get Channel Refs
Retrieve the input and output channels by section. The 'Incoming' and 'Outgoing' tags here are placeholders. Replace them with the names of your sections. If you have multiple sections, expand the build array and bundle the references accordingly.

#CodeRecommended - Compile.Settings
Read any configuration data and bundle it into your Compiled.Settings type def

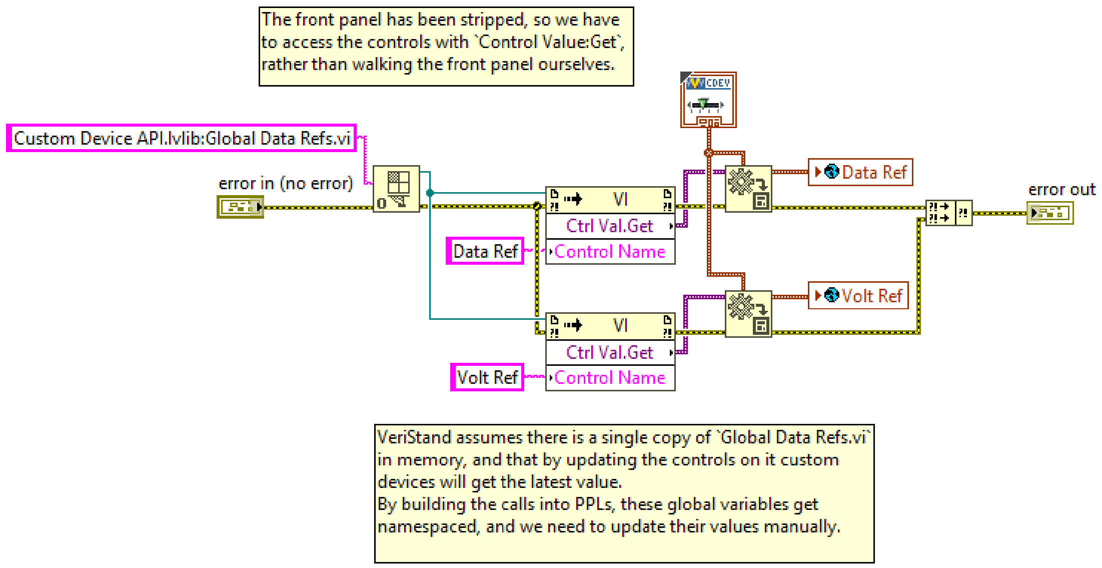
Update Global Data References

When using certain VeriStand Custom Device APIs, such as in an inline custom device, you must update how global data references are initialized.

For performance reasons, channel values are stored in VeriStand as a single block of data. For example, they can be sorted as an array of double values. To access a value element corresponding to a given channel, VeriStand uses Global Variables to pass engine pointer information to the calling APIs.

This approach works for LLB based custom devices, but not for PPL. When compiling a PPL, a separate, namespaced, copy of the global variable is created and included within the package. The global variable cannot be used to transfer data between the VeriStand engine and the running custom device. Performing a data transfer results in a runtime error. To mitigate this problem, implement an alternative way to access the values within these global variables.

The following initialization code needs to be incapsulated within a subVI, such as **Initialize Global Variables**.

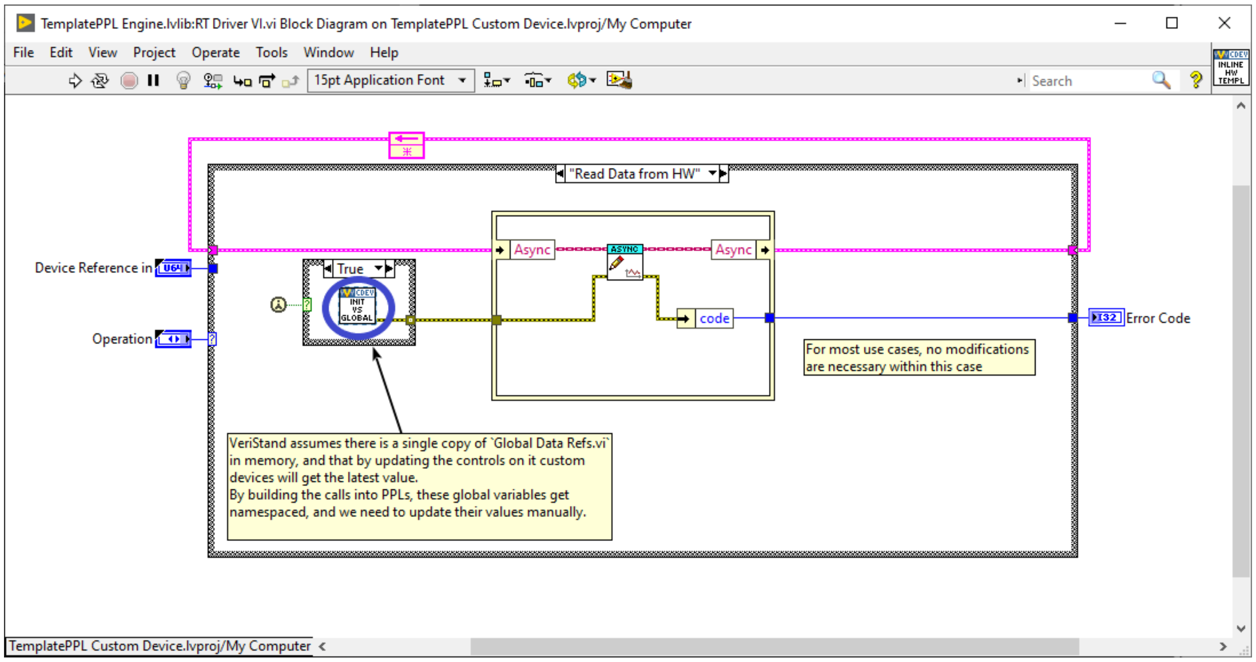


The front panel has been stripped, so we have to access the controls with `Control Value:Get`, rather than walking the front panel ourselves.

VeriStand assumes there is a single copy of `Global Data Refs.vi` in memory, and that by updating the controls on it custom devices will get the latest value.
By building the calls into PPLs, these global variables get namespaced, and we need to update their values manually.

This code can be called only once from within the target custom device. For an inline custom device, it must also be called in the RT Driver **Read Data from HW** case for engine initialization.

The following image displays where to insert this code.


TemplatePPL Engine.lvlib:RT Driver VI.vi Block Diagram on TemplatePPL Custom Device.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
"Read Data from HW"
Device Reference in
Operation
True
INIT VS GLOBAL
Async
Async
code
Error Code
TemplatePPL Custom Device.lvproj/My Computer

For most use cases, no modifications are necessary within this case

VeriStand assumes there is a single copy of `Global Data Refs.vi` in memory, and that by updating the controls on it custom devices will get the latest value.
By building the calls into PPLs, these global variables get namespaced, and we need to update their values manually.

Update Libraries

You must create a PPL for each LLB build specification in the project.

The PPL needs to have a similar configuration to the LLB. You should keep the same built files structure, including the same file structure as the LLB build specifications. This will allow you to reuse the build post-step to copy generated files to the VeriStand directory.

To make these library updates in LabVIEW, right-click **Build Specifications** and select **New** » **Packed Library**.

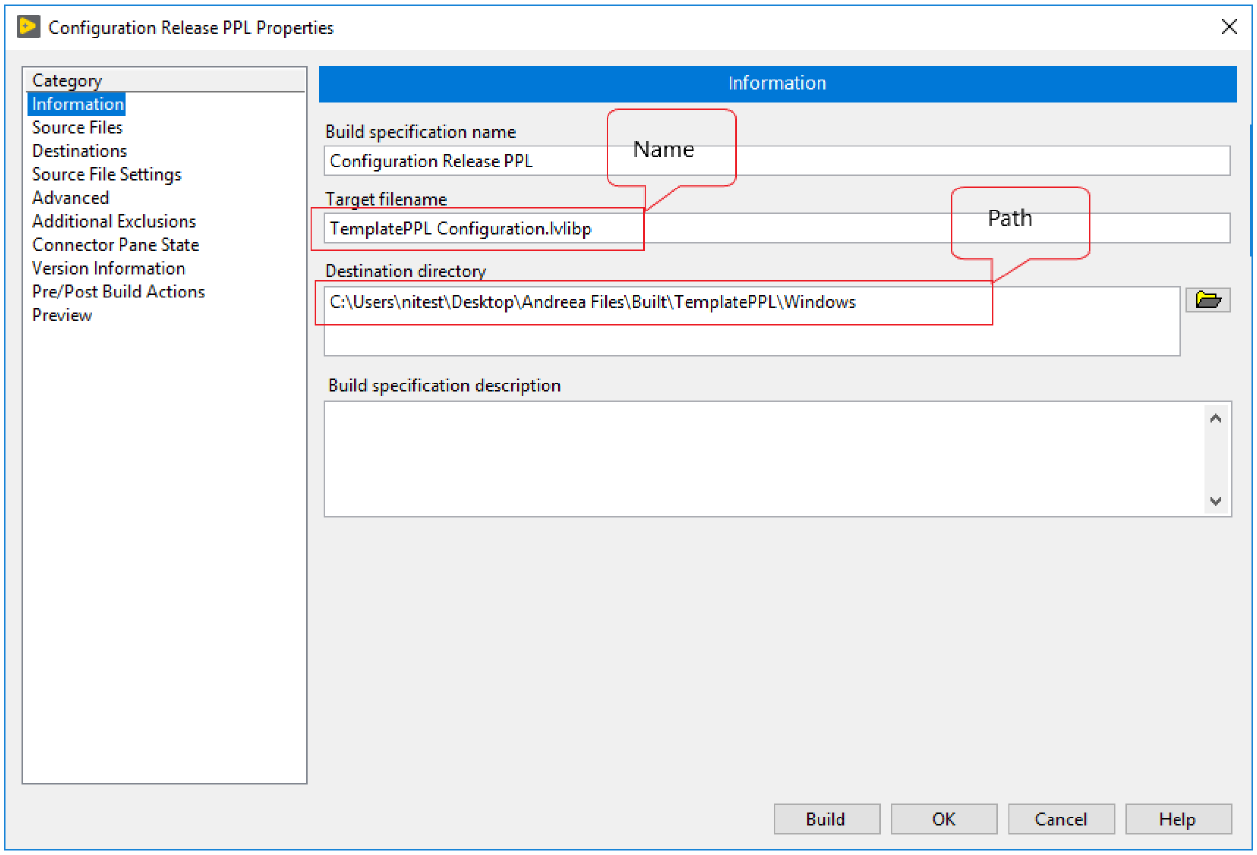
Information Settings

In the Configuration Release PPL Properties dialog box, click **Information** to enter a new **Build Specification Name**.

There are rules to consider when organizing the built files. These rules ensure that the files can be copied or moved to the VeriStand Custom Device's directory once built.

Note: The Post Build Action VI operates based on the same naming and path convention.

a) Custom Device System Explorer



Build specification name: Configuration Release PPL

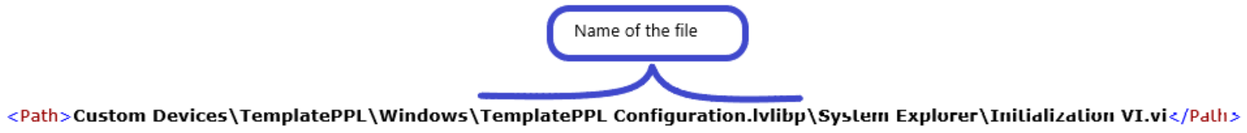
Target filename: TemplatePPL Configuration.lvlibp

Destination directory: C:\Users\nitest\Desktop\Andreea Files\Built\TemplatePPL\Windows

Build specification description:

- The destination path should have the following structure: `"..\built\Custom Device Name\Operating System`
- The PLL file name should use the following format: [Custom Device Name] + [Configuration]

The following image displays an example name and path of a Windows Configuration library file.

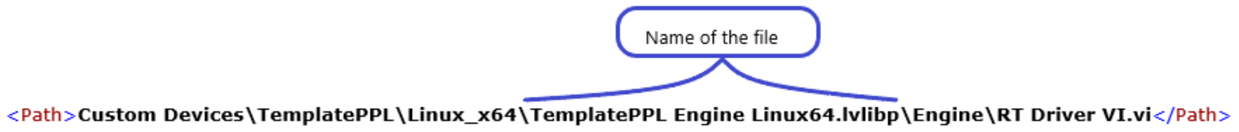


```
<Path>Custom Devices\TemplatePPL\Windows\TemplatePPL Configuration.lvlibp\System Explorer\Initialization VI.vi</Path>
```

b) Custom Device Engine

- The destination path should use the following structure: `"..\built\Custom Device Name\Operating System`
- The PLL file name should use the following format: [Custom Device Name] + [Engine] + [Operating System Name]

The following image displays an example name and path of a Linux engine file.



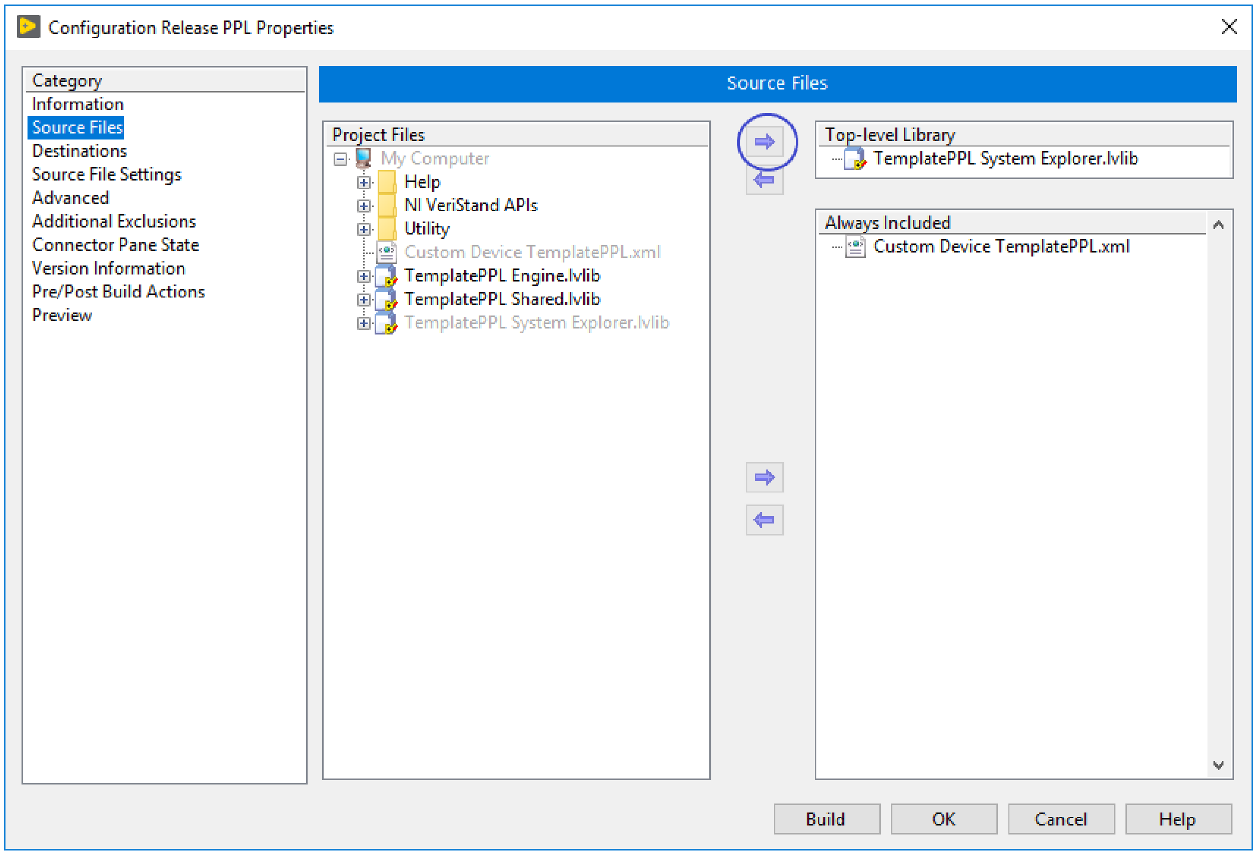
```
<Path>Custom Devices\TemplatePPL\Linux_x64\TemplatePPL Engine Linux64.lvlibp\Engine\RT Driver VI.vi</Path>
```

Note: Starting with VeriStand 2021, VeriStand only supports a Linux x64 real-time operating system.

Source Files Settings

a) Custom Device System Explorer

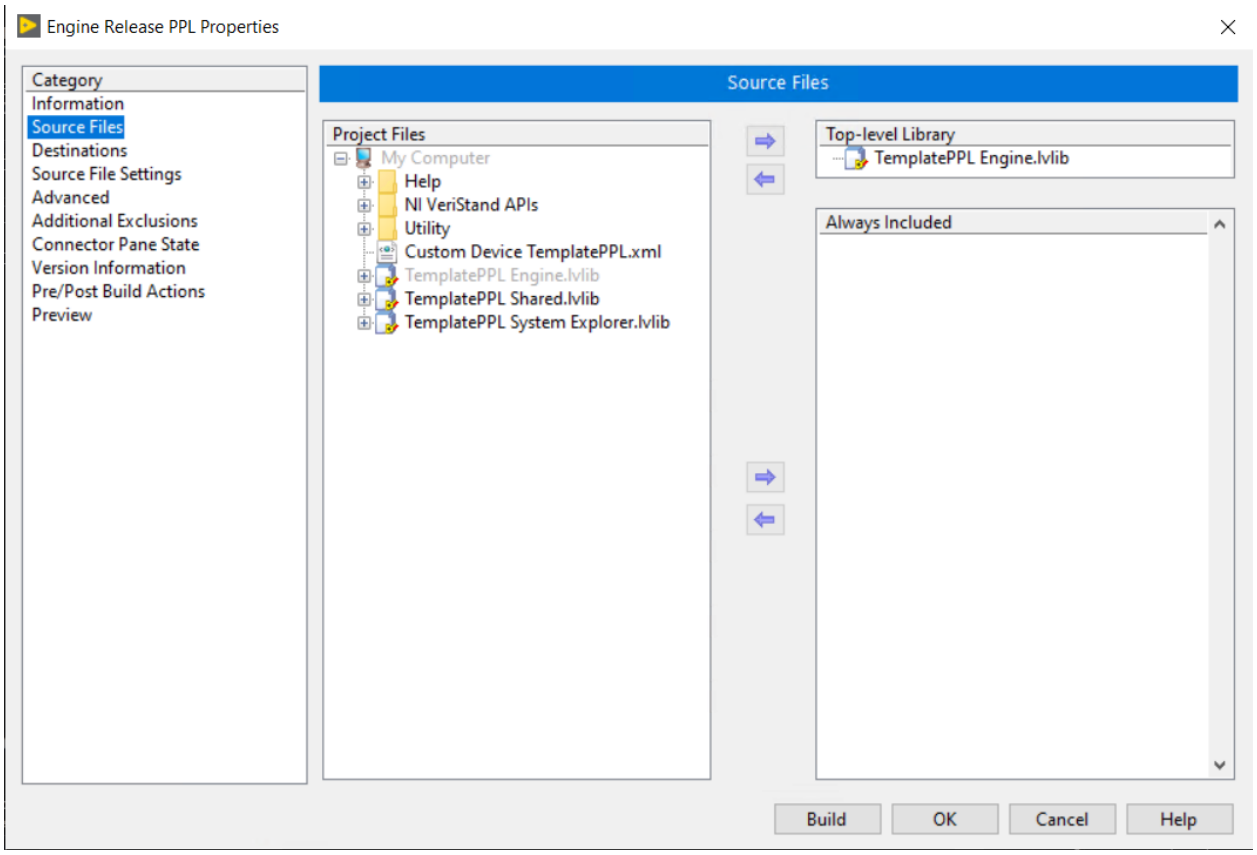
In the Configuration Release PPL Properties dialog box, click **Source Files** and select the library containing the System Explorer files of your custom device (i.e. Configuration library). Set this library as the **Top-level Library**.



Note: To distribute additional files with the PPL, such as a configuration XML file, add them to the **Always-Included** files list.

b) Custom Device Engine

In the Configuration Release PPL Properties dialog box, click **Source Files** and select the library containing the engine files of your custom device (i.e. Configuration library). Set this library as the **Top-level Library**.



Note: You should also configure PPLs for the Real Time targets build specifications.

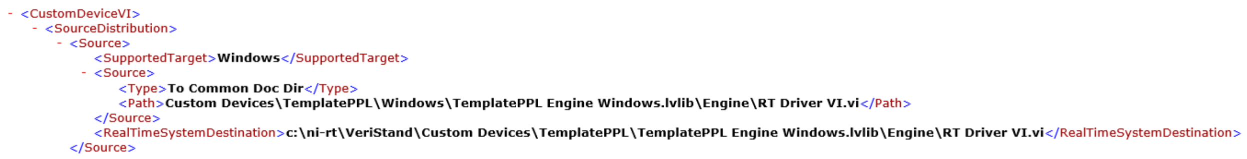
Update the XML

You must update XML files either manually or by creating a **Post-Build Action** VI.

Manual Updates

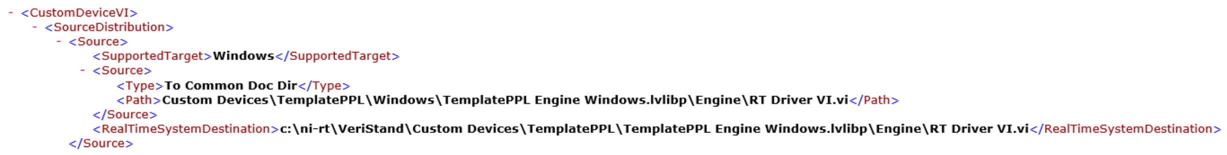
To update manually, you will need to update the path for each LLB with the path of each corresponding PPL.

The following image displays the XML code sequence for the custom device RT Driver VI on a Windows target.



```
- <CustomDeviceVI>
  - <SourceDistribution>
    - <Source>
        <SupportedTarget>Windows</SupportedTarget>
      - <Source>
          <Type>To Common Doc Dir</Type>
          <Path>Custom Devices\TemplatePPL\Windows\TemplatePPL Engine Windows.lvlib\Engine\RT Driver VI.vi</Path>
        </Source>
        <RealTimeSystemDestination>c:\ni-rt\VeriStand\Custom Devices\TemplatePPL\TemplatePPL Engine Windows.lvlib\Engine\RT Driver VI.vi</RealTimeSystemDestination>
      </Source>
```

You need to update what comes after the <Path> tag.



```
- <CustomDeviceVI>
  - <SourceDistribution>
    - <Source>
        <SupportedTarget>Windows</SupportedTarget>
      - <Source>
          <Type>To Common Doc Dir</Type>
          <Path>Custom Devices\TemplatePPL\Windows\TemplatePPL Engine Windows.lvlibp\Engine\RT Driver VI.vi</Path>
        </Source>
        <RealTimeSystemDestination>c:\ni-rt\VeriStand\Custom Devices\TemplatePPL\TemplatePPL Engine Windows.lvlibp\Engine\RT Driver VI.vi</RealTimeSystemDestination>
      </Source>
```

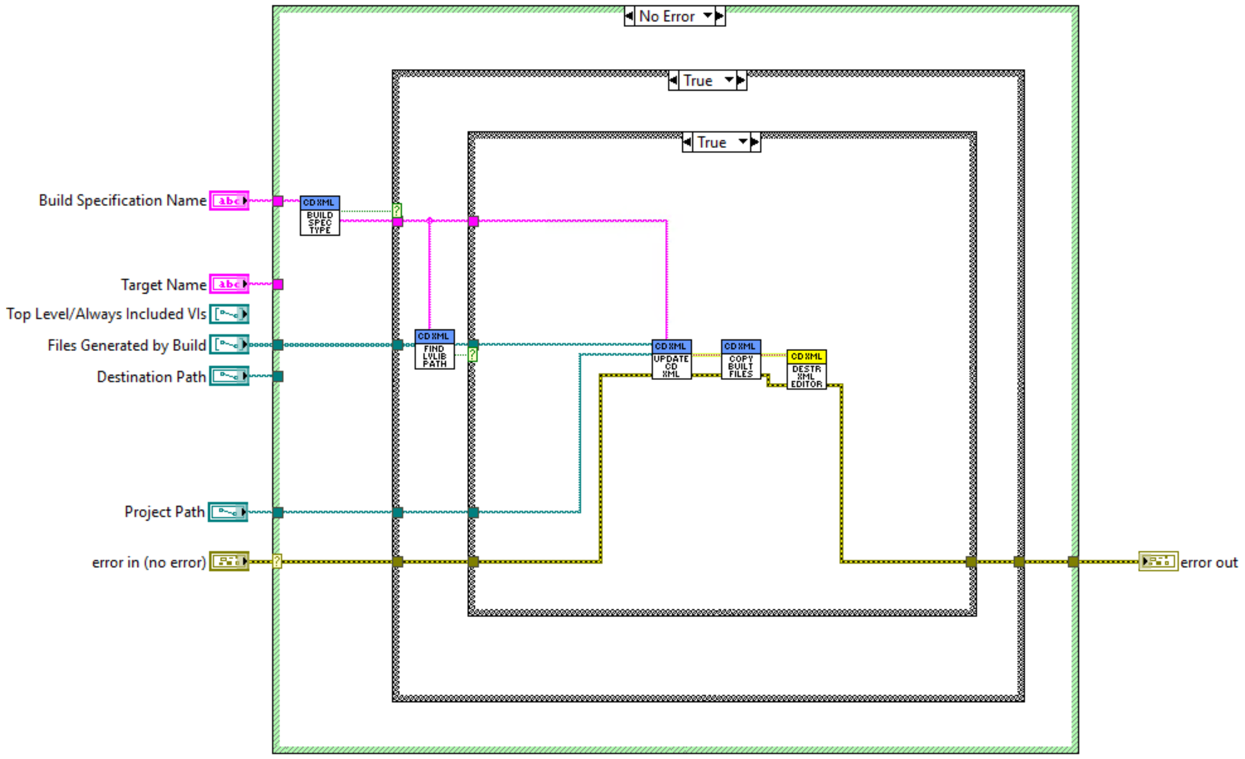
The <Path> tag contains a path for a VI located inside a PPL. You need to update all the <Path> tags in the XML that reference VIs inside the newly created PPLs.

Note: You also need to update the <`RealTimeSystemDestination`> tags.

Post-Build Action VI

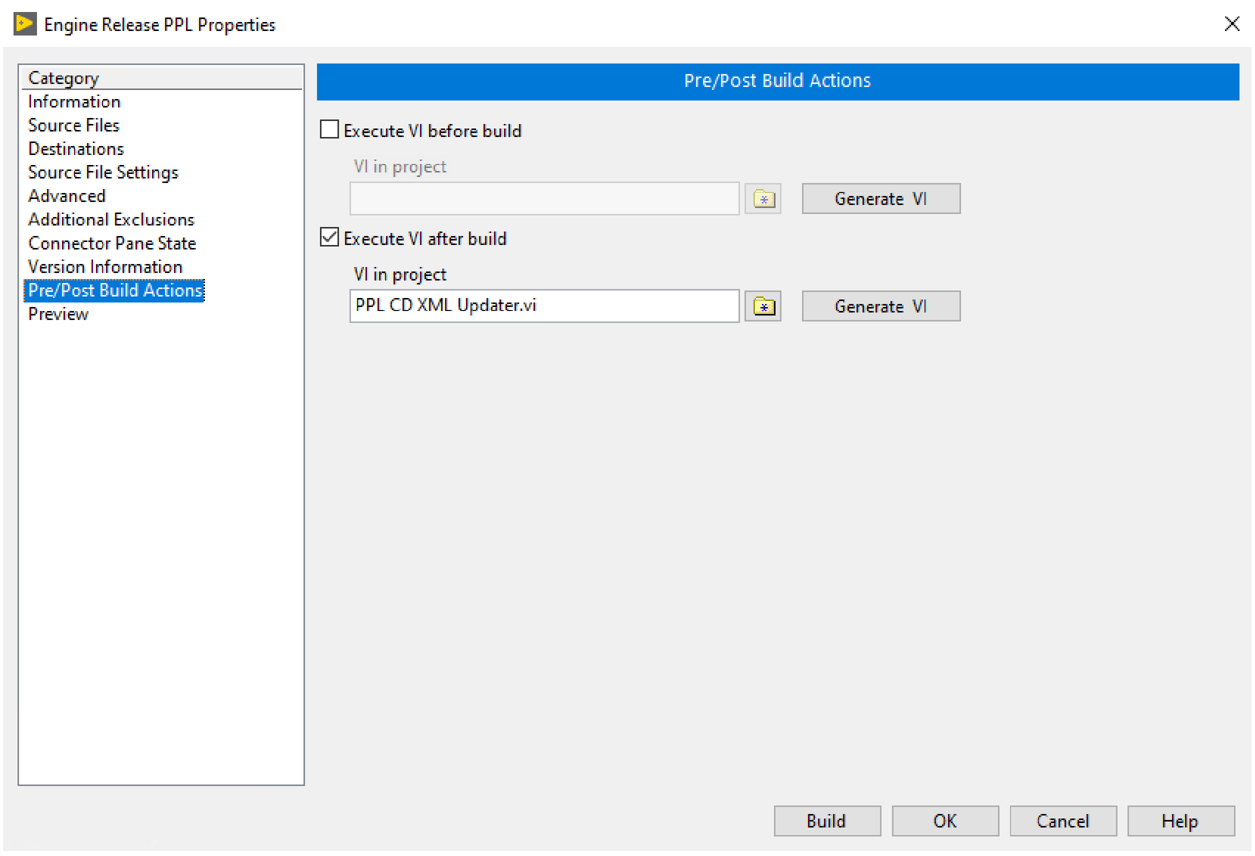
To automate path updates, implement a VI to update the XML file.

The following image displays one possible VI.



You can set this VI in LabVIEW as a Post-Build Action for your engine or configuration build specification.

To set the action in LabVIEW, open the Engine Release PPL Properties dialog box and click **Pre/Post Build Actions** to enable **Excute VI after build**.



☐ Execute VI before build

VI in project

☑ Execute VI after build

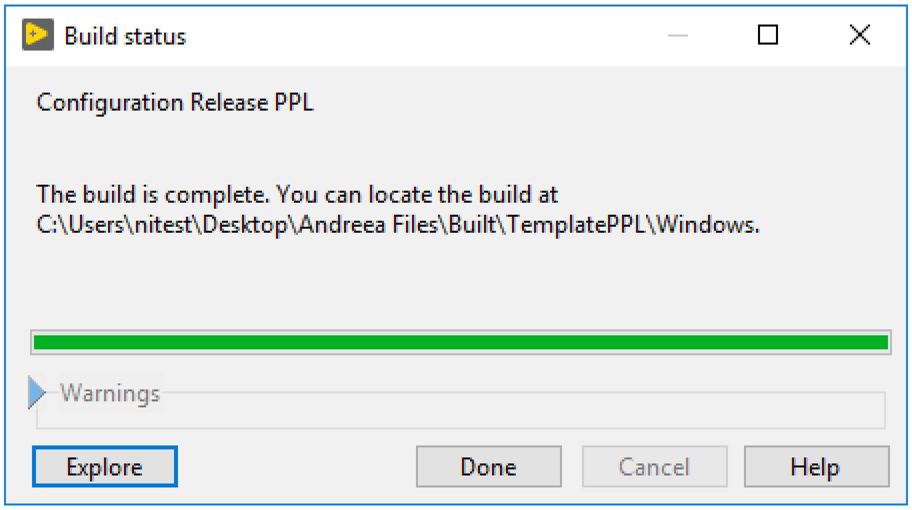
VI in project

PPL CD XML Updater.vi

Build the PPL File

Once finished with your updates, you can build the PPL.

The following window should display after the build finishes.



Configuration Release PPL

The build is complete. You can locate the build at
C:\Users\nitest\Desktop\Andreea Files\Built\TemplatePPL\Windows.