

Technical Document

Niagara^{AX-3.x} Developer Driver Framework Guide

January 7, 2008



Niagara^{AX} Developer Driver Framework Guide

Copyright © 2008 Tridium, Inc.

All rights reserved.

3951 Westerre Pkwy., Suite 350

Richmond

Virginia

23233

U.S.A.

Copyright Notice

The software described herein is furnished under a license agreement and may be used only in accordance with the terms of the agreement.

This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from Tridium, Inc.

The confidential information contained in this document is provided solely for use by Tridium employees, licensees, and system owners; and is not to be released to, or reproduced for, anyone else; neither is it to be used for reproduction of this Control System or any of its components.

All rights to revise designs described herein are reserved. While every effort has been made to assure the accuracy of this document, Tridium shall not be held responsible for damages, including consequential damages, arising from the application of the information contained herein. Information and specifications published here are current as of the date of this publication and are subject to change without notice.

The release and technology contained herein may be protected by one or more U.S. patents, foreign patents, or pending applications.

Trademark Notices

BACnet and ASHRAE are registered trademarks of American Society of Heating, Refrigerating and Air-Conditioning Engineers. Microsoft and Windows are registered trademarks, and Windows NT, Windows 2000, Windows XP Professional, and Internet Explorer are trademarks of Microsoft Corporation. Java and other Java-based names are trademarks of Sun Microsystems Inc. and refer to Sun's family of Java-branded technologies. Mozilla and Firefox are trademarks of the Mozilla Foundation. Echelon, LON, LonMark, LonTalk, and LonWorks are registered trademarks of Echelon Corporation. Tridium, JACE, Niagara Framework, Niagara^{AX} and Vykon are registered trademarks, and Workbench, WorkPlace^{AX}, and ^{AX}Supervisor, are trademarks of Tridium Inc. All other product names and services mentioned in this publication that is known to be trademarks, registered trademarks, or service marks are the property of their respective owners. The software described herein is furnished under a license agreement and may be used only in accordance with the terms of the agreement.

Developer Driver Framework Documentation

The developer driver framework helps Niagara AX developers create device drivers that interface the Niagara AX framework to external equipment such as building automation controllers, unitary controllers, security controllers, lighting controllers, etc.

The documentation for the developer driver framework consists of a tutorial whose theme is, *A Driver in a Week*. We hope that by following this tutorial, a Niagara AX developer will be able to arrive at a device driver for his or her own equipment in as little as a week's worth of development effort. This device driver will be able to scan a communications port for equipment, check the status of the equipment, and import data points from the equipment into the Niagara AX framework for reading (polling) and writing.

- [Tutorial](#)
 - [Revision History](#)
 - [Overview](#)
 - [Platforms](#)
 - [Stations](#)
 - [Workbench](#)
 - [Logic](#)
 - [Services](#)
 - [Drivers](#)
 - [Day 1 - Field Device Pinging](#)
 - [Preparation](#)
 - [Chapter 1 - Ping Request](#)
 - [Chapter 2 - Device Id](#)
 - [Chapter 3 - Device](#)
 - [Chapter 4 - Receiver](#)
 - [Chapter 5 - Ping Response](#)
 - [Chapter 6 - Communicator](#)
 - [Chapter 7 - Network](#)
 - [Chapter 8 - Put it Together](#)
 - [Review](#)
 - [Day 2 - Reading Data Points \(Polling\)](#)
 - [Chapter 9 - Read Request](#)
 - [Chapter 10 - Read Parameters](#)
 - [Chapter 11 - Read Response](#)
 - [Chapter 12 - Proxy Extension](#)
 - [Chapter 13 - Parse Read Response](#)
 - [Chapter 14 - Point Id](#)
 - [Chapter 15 - Point Device Extension](#)
 - [Chapter 16 - Testing Your Progress](#)
 - [Conclusion](#)
 - [Day 3 - Writing Data Points](#)
 - [Chapter 17 - Write Request](#)

- [Chapter 18 - Write Parameters](#)
- [Chapter 19 - Write Response](#)
- [Chapter 20 - Update Point Id](#)
- [Chapter 21 - Auto Request and Response](#)
- [Chapter 22 - Testing Your Progress](#)
- [Conclusion](#)
- [Day 4 - Device Discovery/a>](#)
 - [Chapter 23 - Device Discovery Parameters](#)
 - [Chapter 24 - Device Discovery Request](#)
 - [Chapter 25 - Device Discovery Response](#)
 - [Chapter 26 - Device Discovery Preferences](#)
 - [Conclusion](#)
- [Day 5 - Point Discovery](#)
 - [Chapter 27 - Point Discovery Parameters](#)
 - [Chapter 28 - Point Discovery Leaf](#)
 - [Chapter 29 - Point Discovery Request](#)
 - [Chapter 30](#)
 - [Chapter 31](#)
 - [Conclusion](#)
- [Appendix 1 - Driver Actions](#)
- [Appendix 2 - Device Manager Buttons](#)
- [Appendix 3 - Point Manager Buttons](#)
- [Appendix 4 - Exclusive Communications Access](#)
- [Appendix 5 - Defining Tags For Outgoing Requests and Incoming Frames](#)
- [Appendix 6 - Processing Unsolicited Received Data Frames](#)
- [Appendix 7 - Accessing the Point Id From the Read Request](#)

Developer Driver Tutorial

Welcome to the [Developer Driver Tutorial for Niagara AX](#)! This tutorial will walk you through the process of creating a device driver for the Niagara AX framework. Device drivers facilitate access to external equipment that is connected to a Niagara AX server through a communications or network port. This tutorial will start with a general overview of Niagara AX itself. This overview is from the author's perspective as a Niagara AX device driver developer himself. Those already familiar with Niagara AX may use the overview simply for reference. After that, this will guide you through a five-day process of Java development. It is the author's intention that you will have a fully functional, or near fully-functional driver at the end of this tutorial.

- [Revision History](#)
- [Overview](#)
 - [Platforms](#)
 - [Stations](#)
 - [Workbench](#)
 - [Logic](#)
 - [Services](#)
 - [Drivers](#)
- [Day 1 - Field Device Pinging](#)
 - [Preparation](#)
 - [Chapter 1 - Ping Request](#)
 - [Chapter 2 - Device Id](#)
 - [Chapter 3 - Device](#)
 - [Chapter 4 - Receiver](#)
 - [Chapter 5 - Ping Response](#)
 - [Chapter 6 - Communicator](#)
 - [Chapter 7 - Network](#)
 - [Chapter 8 - Put it Together](#)
 - [Review](#)
- [Day 2 - Reading Data Points \(Polling\)](#)
 - [Chapter 9 - Read Request](#)
 - [Chapter 10 - Read Parameters](#)
 - [Chapter 11 - Read Response](#)
 - [Chapter 12 - Proxy Extension](#)
 - [Chapter 13 - Parse Read Response](#)
 - [Chapter 14 - Point Id](#)
 - [Chapter 15 - Point Device Extension](#)
 - [Chapter 16 - Testing Your Progress](#)
 - [Conclusion](#)
- [Day 3 - Writing Data Points](#)
 - [Chapter 17 - Write Request](#)
 - [Chapter 18 - Write Parameters](#)
 - [Chapter 19 - Write Response](#)
 - [Chapter 20 - Update Point Id](#)

- [Chapter 21 - Auto Request and Response](#)
 - [Chapter 22 - Testing Your Progress](#)
 - [Conclusion](#)
- [Day 4 - Device Discovery](#)
 - [Chapter 23 - Device Discovery Parameters](#)
 - [Chapter 24 - Device Discovery Request](#)
 - [Chapter 25 - Device Discovery Response](#)
 - [Chapter 26 - Device Discovery Preferences](#)
 - [Conclusion](#)
- [Day 5 - Point Discovery](#)
 - [Chapter 27 - Point Discovery Parameters](#)
 - [Chapter 28 - Point Discovery Leaf](#)
 - [Chapter 29 - Point Discovery Request](#)
 - [Chapter 30 - Point Discovery Response](#)
 - [Chapter 31 - Point Discovery Preferences](#)
 - [Conclusion](#)
- [Appendix 1 - Driver Actions](#)
- [Appendix 2 - Device Manager Buttons](#)
- [Appendix 3 - Point Manager Buttons](#)
- [Appendix 4 - Exclusive Communications Access](#)
- [Appendix 5 - Defining Tags For Outgoing Requests and Incoming Frames](#)
- [Appendix 6 - Processing Unsolicited Received Data Frames](#)
- [Appendix 7 - Accessing the Point Id From the Read Request](#)

Revision History

June 04, 2007

- Early release.

June 12, 2007

- Added instructions for Udp/Ip drivers.
- Updated instructions for Tcp/Ip drivers.
- Added instructions for usage of devDriver.jar and devIpDriver.jar (which replaces devTcpDriver.jar and adds Udp/Ip support).

August 10, 2007

- Official release.
- Polished tutorial.

January 07, 2008

- Updated Day 3 - Chapter 21 to instruct the developer to extend BDdfldParams and implement interface BIDdfWriteParams instead of extending from BDdfWriteParams (which does not exist).
- There were a few references to *easy driver* throughout the documentation, especially in Chapter 17. Easy driver was an early code-name for Dev Driver when the product was in the earliest stages of creation. All references to *Easy Driver* have been changed to either *Dev Driver* or *Ddf*.
- Updated the sample source in chapters 17 and 21 to cast the result of calling *getWriteParameters* to *BYourDriverWriteParams* instead of *BYourDriverReadParams*.
- Added instructions to Day 4 - Chapter 23 to override BDdfDiscoverParams.getDiscoveryLeafType. This step was accidentally left out of the tutorial but without it, device discovery does not occur.
- Replaced the links to *Conclusion 1* and *Conclusion 2* on the opening lesson for day 3 with a link to the single *Conclusion* page for day 3.
- Fixed some of the formatting for chapter 23.
- Added more description to all line items in the table of contents.
- Added Appendix 7 - Accessing the Point Id From the Read Request
- Replaced bad references to ddfInet.udp, ddfInet.tcp, ddfUdp and ddfTcp with com.tridium.ddfIp.udp or com.tridium.ddfIp.tcp

Niagara AX Framework Overview

- [Platforms](#)
- [Stations](#)
- [Workbench](#)
- [Logic](#)
- [Services](#)
- [Drivers](#)

Niagara AX Framework Overview - Platforms

A **platform** is a computer that supports the Niagara AX framework. A platform is usually a **Jace**, **Web Supervisor**, or a **Workstation**.

Jace

A Jace is a type of server (perhaps with an *embedded operating system*) that is created specifically for the purposes of running Niagara AX stations.

Web Supervisor

Web Supervisors are enterprise servers that support the Niagara AX framework and run a Niagara Station whose purpose (among other things) is typically to gather data from other Niagara stations.

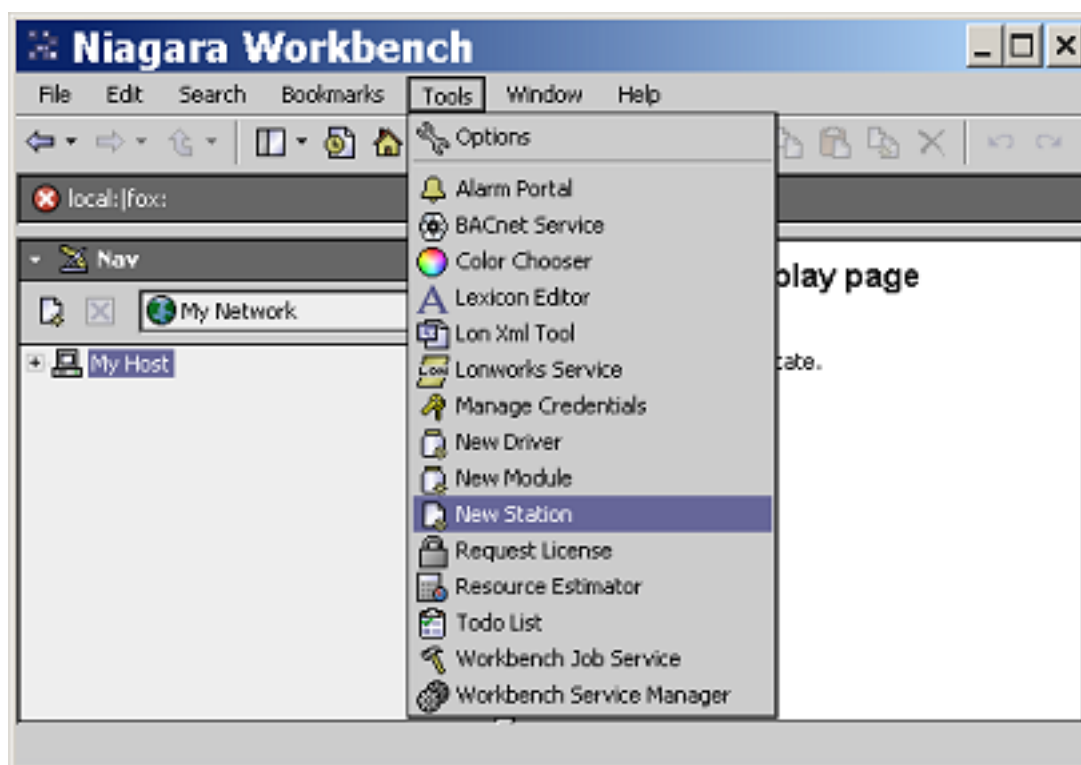
Workstation

Workstations are personal computers that typically run the Niagara AX Workbench and allow Niagara installation and (or) maintenance professionals to configure other Niagara platforms and stations. Workstations are also personal computers used by professional developers to create software applications for the Niagara AX framework (such as Niagara AX drivers).

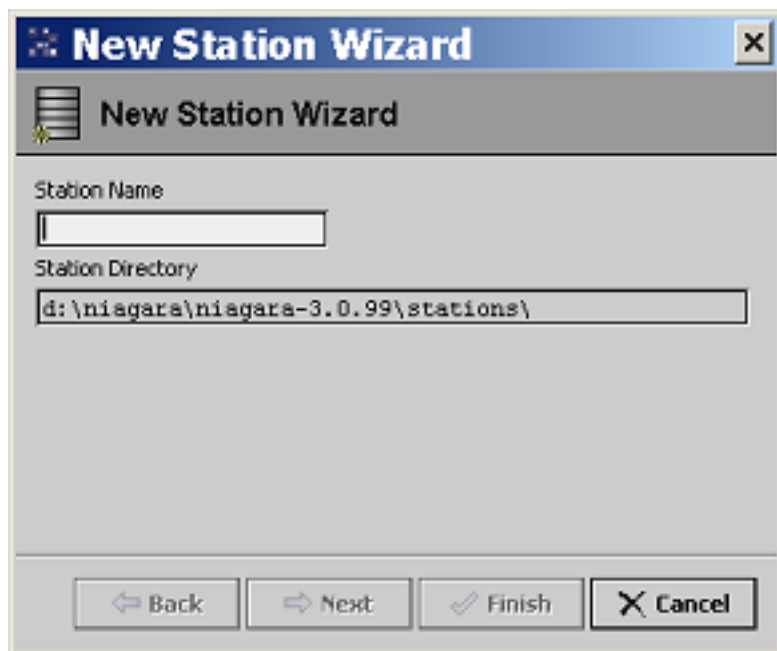
Niagara AX Framework Overview - Stations

A Niagara AX **station** is a program that runs on a Niagara AX *platform*. The **station** program usually starts running immediately after the Niagara AX *platform* boots. The **station** program continues to run forever or until somebody or something causes the *platform* computer to be turned off or rebooted.

Within a Niagara AX **station** you will find **logic**, **services**, and **drivers**. Niagara installation professionals create a Niagara AX **station** by using the Niagara AX Workbench application. To do this they click the **Tools** menu on the main menu and then they click the **New Station** item on the **Tools** menu. This presents a wizard that generates a **station** on the Workstation computer.



This is the **Tools** menu in the Niagara AX Workbench. The **New Station** item is highlighted.



This is a screen-shot of the **New Station Wizard** from the Niagara AX Workbench.

The **station**, as generated by the wizard, is not a running **station**. Instead, it is an **offline station**. The installation professional will then define the behavior of the **station** (by using the Niagara AX Workbench) before copying it to the *platform* computer (using Niagara AX Workbench for this too) that will forever be dedicated to running the **station**. To test his or her **station's** behavior, the installation professional may decide to run the **station** in his or her own *Workstation* PC, before ultimately deploying the **station** in a *platform* computer that is dedicated to running the **station**.

An **online station** is one that is currently running on a Niagara AX *platform* computer. Once again, a Niagara AX *platform* computer can be a *Jace*, *Web Supervisor*, or *Workstation*. Installation professionals can use the Niagara AX Workbench to configure **stations** that are **online** or **offline**.

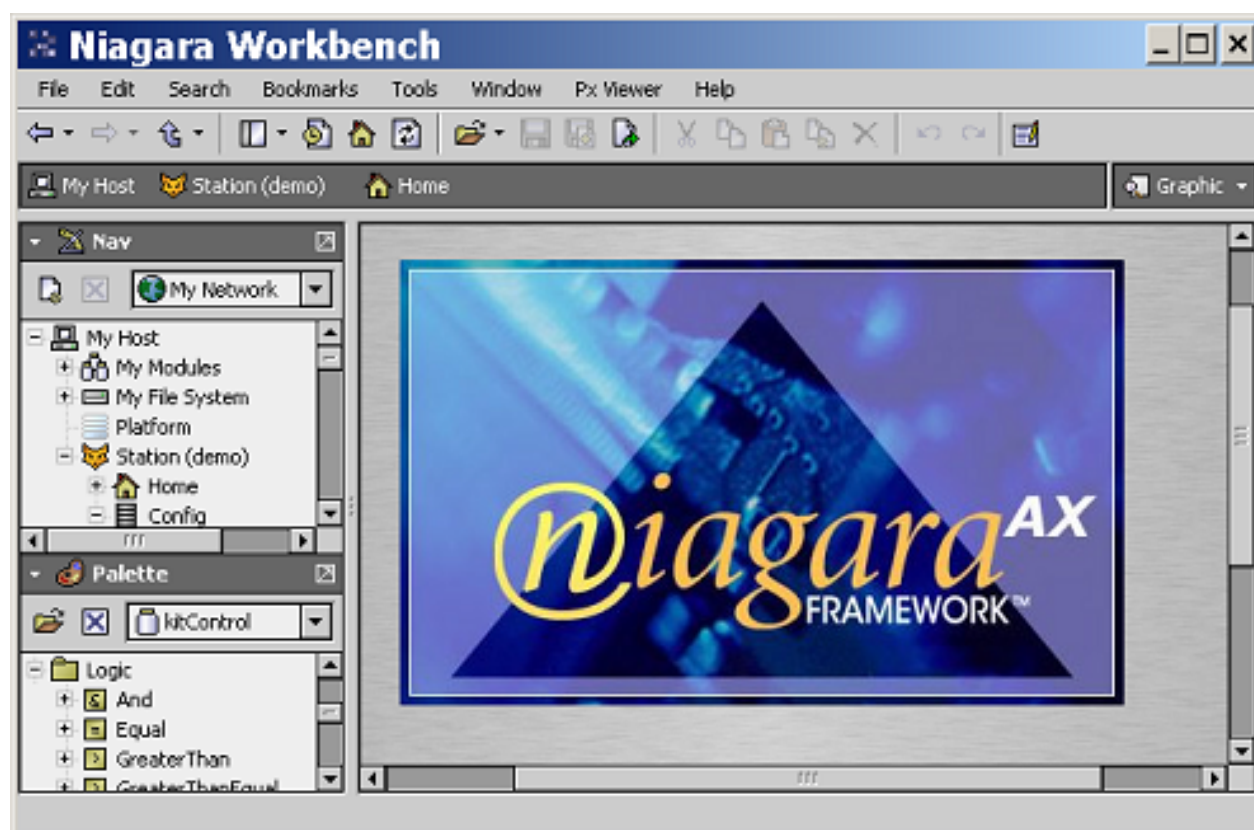
Niagara AX Framework Overview - Workbench

The Niagara AX **Workbench** is a powerful tool with features of a file explorer and a computer-aided-design (CAD) application. It allows Niagara AX installation or maintenance professionals to graphically review and edit the contents and behavior of a Niagara AX *station* as well as the configuration of a Niagara AX *platform* -- the computer on which the station is running.

The main window of the Niagara AX **Workbench** application is split into sections. Approximately the left-most third of the window displays the **navigation tree**. The **navigation tree** displays the *station* in terms of a tree structure. This looks similar to how file explorers show the contents of the file system (hard drives, compact disk drives, flash drives etc.) The right-most two-thirds of the main **Workbench** window shows further information and details about whatever is selected in the **navigation tree**.

Workbench - Palette

The Niagara AX installation professional can add logic (read further please for details about logic) to the station by dragging components (read further please for details about components) out of one or more **Palettes** and dropping these components into the station. The **palettes** are typically displayed at the bottom of the left-most third of the main **Workbench** application window, below the **navigation tree**. If the **palette** is not there then you can reveal it by clicking on the Window item of the main menu, hovering over to the Side Bars menu item, and then clicking the Palette item from the Side Bars menu.

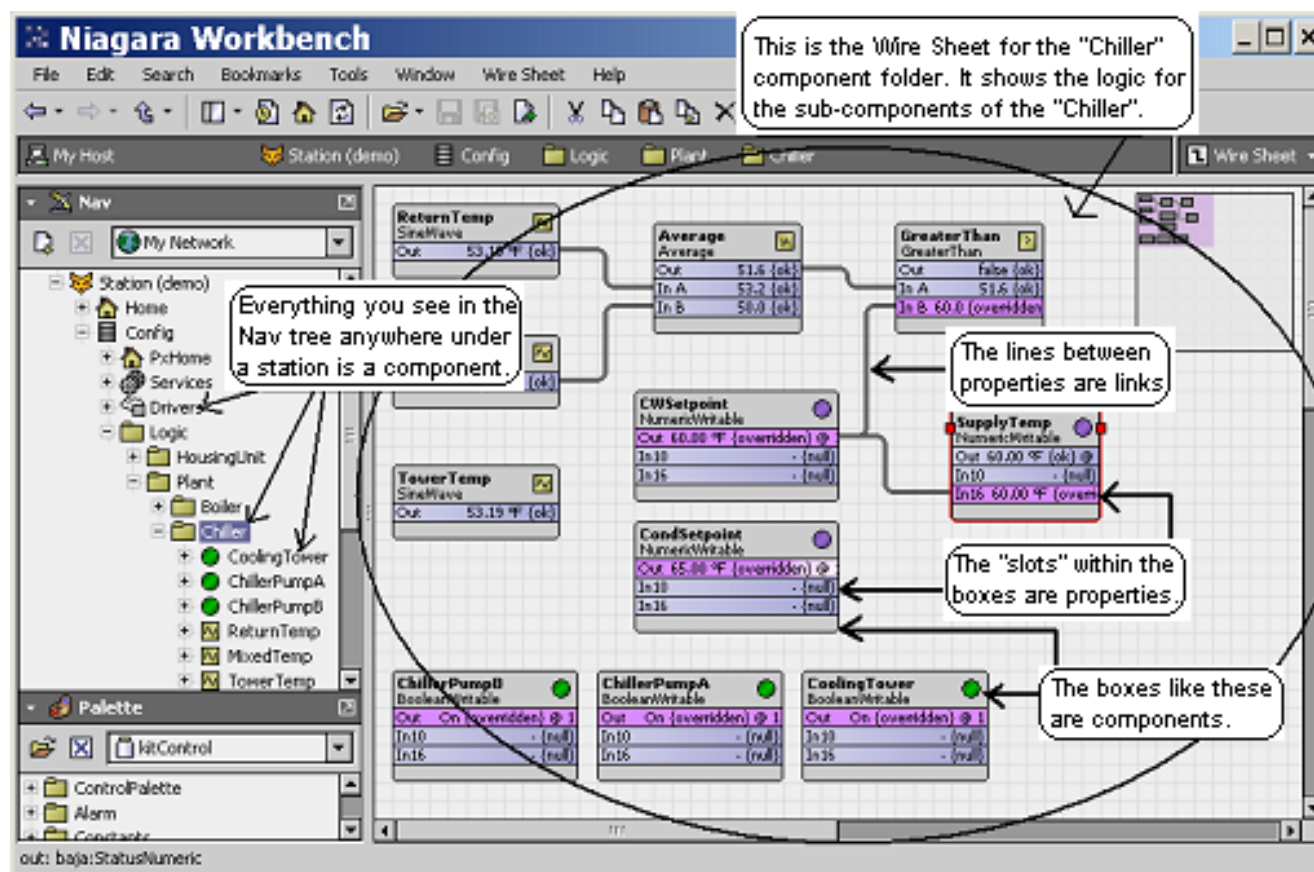


This is a screen-shot of the Niagara AX **Workbench**. The *navigation tree* (**Nav**) is located towards the left-center. The **palette** is located at the lower-left. The main display area is located towards the center and right.

Niagara AX Framework Overview - Logic

The **logic** is the station's program or purpose. A Niagara AX installation professional or systems integrator creates the **logic** in each station on each platform that is deployed. The installation professional uses the Niagara AX Workbench software to create the **logic** for the station.

Please refer to the following illustration while reading this section.



Logic - Components

The installation or maintenance professional drags **components** from the palettes within the Niagara AX Workbench into the station as needed. Each **component** performs a simple or complicated computation upon its properties. Properties are data values on a **component**.

Logic - Properties & Links

Properties can be assigned values directly from within the workbench. Alternatively, the installation professional can draw a line from a **property** on one **component** to a **property** on another **component**. This is called a **link**. The Niagara AX station program ensures that whenever a value on the source **property** changes, the result *flows* across the link and updates the **property** on the other side of the **link**. This causes the latter **component** to reprocess its own computations and possibly update the values for its other **properties** as a result of the change to its .

In short, **properties** can be assigned values directly from within the Niagara AX Workbench, or by being **linked** to -- from another **property** that changes, or by a user from a web browser (please read further about "Px" Pages for more information).

Driver Components: Niagara AX drivers feature special software components whose properties display data values from some external equipment that is connected to the platform computer on which the Niagara AX station is running. When the property values are changed from within Niagara, these special components update the corresponding data values in the external equipment.

The special components that feature properties and synchronize data values between a Niagara AX station and some external equipment are called *driver control points*.

Logic - Actions

Components sometimes have **actions** on them. An **action** is a way of telling the component to perform a special computation or to do something special. **Actions** are identified by names that usually describe the special computation that the component will take if the **action** is invoked. **Actions** can be invoked directly from within the Niagara AX Workbench or from a web browser (please read further about "Px" Pages for more information). To view or invoke an **action** from within Workbench, right-click a **component** either in the *Nav* tree or in the *Wire Sheet*, a pop-up menu will appear, hover over the **Actions** side-menu. All available **actions** on the component will appear in the **Actions** side-menu. If the **Actions** side-menu is grayed-out, then that means that there are no **actions** available for the selected **component**.

Niagara AX drivers feature special software components. The **actions** on these special component's can be programmed so that when invoked, they cause some behavior to occur within the equipment that is connected to the platform computer on which the station is running. Typical **actions** might be called **Reboot**, **Set Time**, or **Set Date**. When invoked on a special component in your driver, it might *reboot* any equipment that the component represents, *set the time* in the equipment, or *set the date*.

The special driver components that would likely feature these actions are called *driver devices* and *driver control points*.

Logic - Topics

Components sometimes have **topics**. A **topic** is an event that the component fires when the component detects that a certain condition has occurred, whatever the certain condition might be depends on the component itself.

Niagara AX drivers feature special software components. The **topics** on these special component's can be programmed to *fire* whenever the driver detects that some behavior has just occurred inside the external equipment. This is one way of allowing a Niagara AX station to know when something occurs inside the external equipment.

In general, topics are not used very often in drivers. Here are a few typical examples though of when a driver might make use of **topics**:

- Depending on your driver's abilities, you could add a **topic** called *Rebooted* to a *driver device component*. Your driver could fire the *rebooted* **topic** whenever it detects that the corresponding unit of equipment has been power-cycled or otherwise rebooted.
- Depending on your driver's abilities, you could add a **topic** called *Time Changed* to a *driver device component*. Your driver could fire the *Time Changed* **topic** whenever it detects that the corresponding unit of equipment has experienced a change to its internal clock.
- Depending on your driver's abilities, you could add a **topic** called *Date Changed* to a *driver device component*. Your driver could fire the *Date Changed* **topic** whenever it detects that the corresponding unit of equipment has experienced a change to its internal calendar. This assumes that your external device supports such a notion.

The special driver components that would likely feature these topics are called *driver devices* and *driver control points*.

Logic - More About Links

A Niagara installation professional uses the Workbench to edit the logic in a station. The installation professional drags components into the station, as needed, to define the station's behavior. The installation professional draws lines from properties on one component to properties on another component. These lines are called **links**. Whenever the value of a property changes on one component, the value is automatically propagated across the **link** to the value of the corresponding property that is on the other side of the **link**.

Driver *control point* components are virtually tied to a data value that is in a unit of equipment that the driver communicates with. The value for the property named *out* on the *driver control point* reflects the value that is really in the unit of equipment that the driver communicates to. One or more **links** can be drawn from the *out* property to feed any other logic in the station. Likewise, *driver control points* that are writable, feature several properties whose names start with *in*. The value flowing through the logic that is **linked** to the *in* property with the lowest index will be transferred into the proper unit of equipment for the driver, thereby changing a setpoint or other setting in the corresponding equipment.

The Niagara installation professional can also draw a **link** from any property on a component to any action on another component. Whenever the property changes on the first component, the action is invoked on the corresponding component, thereby causing the corresponding component to perform the special computation or behavior described by the action's name.

The Niagara installation professional can also draw a **link** from any topic on a component to any action on another component. Whenever the topic is fired on the first component, the corresponding action on the other component is invoked, thereby also causing the corresponding component to perform the special computation described by the action's name.

Logic - Px Pages

Niagara AX installation professionals create **Px-Pages** to provide or gain feedback from other web users. The web users are those who will ultimately be interacting with the equipment that the driver provides access to. **Px-Pages** are assigned a URL (Internet address) that can be viewed from a web browser or the Workbench software. Niagara AX installation professionals can add one or more **Px-Pages** to any component. To do this, they right-click a component from the *Nav Tree* in Workbench and select the *New View* item from the pop-up menu.

After adding the new **Px-Page** to the component, the main viewing area of the Workbench will become like a graphic editor. The Niagara installation professional will drag one or more of the other components from the station (under the *Nav tree* in Workbench) and drop them into the **Px-Page** editor. When this happens, the editor prompts the installation professional and asks him or her to choose one or more properties or actions from the corresponding components that were dropped onto the graphic. Then text boxes, images, buttons, or other graphical items appear on the graphic. These graphical items are called *Px-Widgets*.

The Niagara AX installation professional then drags these widgets around the screen and possibly edits them to change their color, font, size, etc. As previously mentioned, the **Px-Page** is assigned a URL (special Internet address) that can be visited from a web browser. These text boxes, images, buttons, etc. can be viewed or manipulated directly from a web browser to gain real-time access to the equipment over the Internet.

Niagara AX Framework Overview - Services

Services are essentially mini-programs that run within a station. For example, there is a web service and a database service. The web service runs within the station and listens for HTTP communications on either the HTTP port or whatever port the web service is configured to use. The web service also resolves URL's to Px-Pages (as described in the previous section) or other web serviceable components within the station and returns the resulting web page back to the web browser.

For another example, the "serial" service listens for serial communications on any of the serial ports that are part of the special platform computer that is running the station. For a final example, the Tcp/Ip service listens for Tcp/Ip communications over the platform computer's Ethernet or LAN adapter.

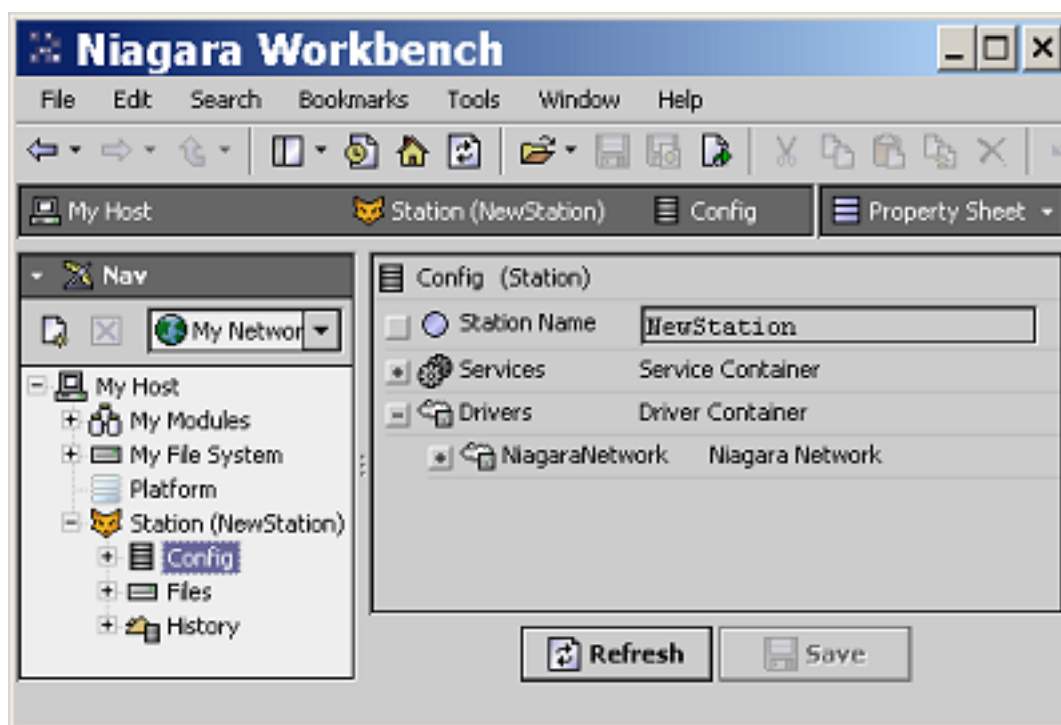
Drivers typically tie into these services, especially the serial or Tcp/Ip services, since most equipment will connect to the station's platform computer either through a serial port, Ethernet adapter, or other LAN adapter. The Developer Driver Framework handles most of the details necessary for this.


Niagara AX Framework Overview - Drivers

If you use the Niagara AX Workbench to connect to a Niagara station, expand the station in the *Nav tree*, and then double-click the *Config* component of the station, you will see that underneath there are two sub-components. One is named *Services* and the other is named *Drivers*. The *Services* component was described in the previous section. The section further describes the *Drivers* component.

Drivers Component of a Station

The following image shows the Niagara AX workbench connected to a very simple station. Please notice that the station is located under *My Host* in the *Nav tree*. *My Host* is this tutorial author's Workstation (a personal computer). In fact, a station named *NewStation* is running as a separate program on the author's Workstation. The author is also running Workbench on the same Workstation. The author has connected from Workbench to the station named *NewStation*. Both the Workbench and the Station are running on the same Workstation.



Please notice that the image shows the station *New Station* in the *Nav tree*. It has an icon of a fox next to it (🦊). Under the station there is a component named *Config*. All stations feature this *Config* component. The author has double-clicked the *Config* component to reveal (in the main Workbench view area) the properties and sub-components of the *Config* component. The author has expanded the *Drivers* component by clicking the  that is next to it in the main view area.

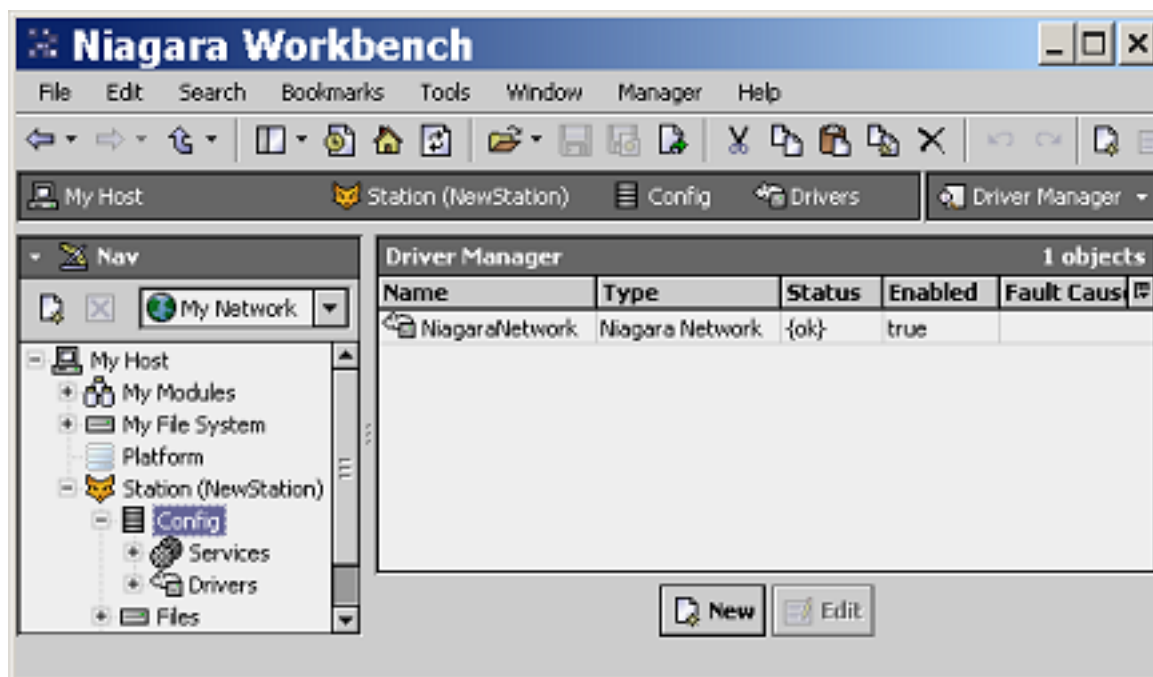
Below the *Drivers* component you will see all of the *driver networks* that are in your station. The *NewStation* as shown contains only one driver network, the *Niagara Network*, which will be further described later.

Driver Networks and the Driver Manager

If you click the *Drivers* component in the main Workbench view or if you double-click the *Drivers* component in the *Nav tree*, the graphical interface that appears is called the *Driver Manager*. It provides more details about the drivers that are currently in the station. It also allows you to add new *driver networks* to the station. A *driver*

network represents a field-bus and a protocol that the station will communicate through in order to access other equipment that is located across a field-bus, network, or communications port to which the station's platform computer is connected.

The following image illustrates the *Driver Manager*. If you click the *New* button on the *Driver Manager* then a window will pop up and ask you to choose which type of new driver network to create. There will be a drop-down list-box that contains all possible new driver networks. The list of all possible networks is determined based on the Niagara jar files that are present on your Workstation.



Driver Devices and the Device Manager

The external equipment is represented inside the Niagara station as *driver devices*, under a *driver network*. You can double-click a *driver network* to get more information about the driver devices that are in the station, or to introduce (add) new *driver devices* to the station. The interface that appears when you double-click a driver network is called the *Device Manager*. It looks similar to the *Driver Manager* but is customized to suite the needs of the particular driver.

Driver Control Points and the Point Manager

A *driver device* represents a single unit of hardware, such as a building automation or industrial controller that is external to the platform computer yet connected to one of the same networks, field-buses, or communications ports as the platform computer. *Driver devices* usually feature a sub-component named *Points*. In Niagara driver-terminology, we call this the *point device extension*. As a reminder, all of this can be seen in the navigation tree by expanding the *Drivers* component, then expanding one of the *driver networks*, and expanding one of the *driver devices* underneath.

The *point device extension* component will ultimately contain one or more of the driver's control points. You can double-click the *point device extension* to visit the driver's *Point Manager*. The driver's *Point Manager* looks similar to the driver's *Device Manager*. Just as the *Driver Manager* allows you to review or add *driver networks*, and just as the *Device Manager* allows you review or add *driver devices*, the *Point Manager* allows you to review or add *driver control points*.

Each of the driver's control points represents one data value that is inside the corresponding, external field-device. Control points are special logic components (as previously defined in this document). This is how Niagara monitors and controls external, smart devices -- such as those found in commercial building control systems (for heating, ventilation, air conditioning, security, lighting, automation, etc), industrial control systems (for automation, etc.), and/or residential building control systems (for heating, ventilation, air conditioning, security, lighting, automation, etc.).

Niagara AX has support for smart devices that communicate over common protocols such as Lon, Bacnet, Obix, OPC, Snmp, and Modbus. Neither Niagara nor Tridium own these protocols. Furthermore, the names of these protocols are likely registered trademarks of their respective owners. Niagara AX also features a rich collection of drivers that allow stations to communicate to equipment that use proprietary protocols, such as Siemens' Staefa Smart II ® commercial series of heating, ventilation, and air conditioning controllers. Staefa Smart II ® is a registered trademark of Siemens.

Niagara AX = Open Architecture

The Niagara AX framework is an open architecture. It is completely extensible and allows Niagara AX enthusiasts to develop their own drivers for communication to equipment that might not otherwise be communicated with using any of the drivers that are among the rich, growing set of drivers available for the Niagara AX framework.

The benefit of this is that once you have a driver that introduces your equipment into the Niagara AX framework, you can inter-connect your equipment, through software (Niagara AX Station logic), with any other equipment that the Niagara AX framework can access. Moreover, you can create one or more web pages to provide access to some or all of your equipment, using Niagara AX, regardless of your equipment's manufacturer. You can also take advantage of alarming, scheduling, and any of the many other features of the Niagara AX framework.

Drivers - Niagara Network

All stations start with one *driver network*, the *Niagara Network*. The Niagara AX station uses the *Niagara Network* component to communicate to Niagara stations that are running on other platform computers (or to allow a station to be accessed by other Niagara stations that are running on other platform computers).

Day 1 - Create Your Driver's Network, Device, and Start Pinging

The developer driver framework will allow you to build a driver for the Niagara AX framework.

Pre Requisite

- A general understanding of the Niagara-AX framework.

Having taken the Tridium Technical Certification Program is strongly recommended. At the very least, please study the [Niagara AX Framework Overview](#) section at the very beginning of this tutorial.

- A bachelors degree (or equivalent experience) in computer science, computer engineering, information technology, or a degree that is generally equivalent to one of these.
- You should have already mastered at least one programming language, such as Java, C, C++, or Visual Basic.

Although the Niagara AX framework is programmed in Java, you do not need to be an experienced Java programmer to successfully develop a driver using the developer driver framework.

- Protocol documentation for the equipment that your driver will communicate with.

*If you are working directly for the manufacturer of the equipment, then they should be able to provide you with one or more documents that describes the way in which the equipment communicates, plus the structure of the data that the equipment expects to see on the **field-bus**. If you have purchased this equipment then you will need to negotiate with the equipment's manufacturer in order to gain access to the equipment's protocol.*

Definitions

field-bus

A network that connects equipment together. A platform computer must also connect to this network, in order to access the equipment from within the Niagara framework. The platform computer will either need to masquerade as a similar unit of equipment or as a master controller, if the equipment's communication protocol is designed to accommodate a master.

Chapters - Day 1

- [Preparation](#)
- [Chapter 1](#)
- [Chapter 2](#)
- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Chapter 6](#)
- [Chapter 7](#)

- [Chapter 8](#)
- [Review](#)

Tutorial Day 1 Preparation

Create an Initial Directory Structure on Your Hard Drive

Follow these directions to create the directory structure on your hard drive that is necessary to develop a driver.

Anywhere that this instructs you to create something called "YourDriver", replace "YourDriver" with the name of your driver. For example, if you have decided to name your driver the "ultimate driver", then replace "yourDriver" with "ultimateDriver" or "YourDriver" with "UltimateDriver". Follow the same capitalization format too. Replace "yourCompany" with the name of your company. For example, if you are developing this driver for a company called Acme Co, then replace "yourCompany" with "acmeCo" and "YourCompany" with "AcmeCo". Follow the same capitalization format here too. Replace "jarfilename" with the name that you wish to assign to the resulting jar file when your driver is compiled. For example, if you want your driver's jar file to be named "ultimate.jar" then replace "jarFileName" with "ultimate".

1. Make a directory somewhere named **jarFileName**
2. Add two empty text files named **build.xml** and **module-include.xml** to the "jarFileName" folder.
3. Add a folder named **src** to the **jarFileName** folder.
4. Add a folder named **com** to the **jarFileName/src** folder.

In java terminology, com means commercial.

5. Add a folder named **yourCompany** to the **jarFileName/src/com** folder.
6. Add a folder named **yourDriver** to the **jarFileName/src/com/yourCompany** folder.
7. Add a folder named **identify** To the **jarFileName/src/com/yourCompany/yourDriver** folder.
8. Add a folder named **point** To the **jarFileName/src/com/yourCompany/yourDriver** folder.
9. Add a folder named **discover** To the **jarFileName/src/com/yourCompany/yourDriver** folder.
10. Add a folder named **comm** To the **jarFileName/src/com/yourCompany/yourDriver** folder.
- In Niagara-AX developer terminology, comm (with two m's) means communication.*
11. Add a folder named **req** to the **jarFileName/src/com/yourCompany/yourDriver/comm** folder.

In Niagara-AX developer terminology, req means request or requests.

12. Add a folder named **rsp** to the **jarFileName/src/com/yourCompany/yourDriver/comm** Folder.

In Niagara-AX developer terminology, rsp means response or responses.

Create an Initial build.xml File

Next, open the empty build.xml file and add the following (replacing the text *yourDriver*, *jarFileName* and *yourCompany* as previously described). Please replace yd with the lower-case version of your driver's initials. For example, if you decide to name your driver the *ultimate driver* then please replace yd with *ud*.

Please consider changing the *description* attribute to a brief sentence that describes your driver.

```

<!-- Module Build File -->

<module
  name = "jarFileName"
  bajaVersion="3.2"
  preferredSymbol = "yd"
  description = "A driver built on the development driver framework."
  vendor = "yourCompany"
>
  <!-- Dependencies -->
  <dependency name="alarm" vendor="Tridium" vendorVersion="3.0" />
  <dependency name="baja" vendor="Tridium" vendorVersion="3.0" />
  <dependency name="driver" vendor="Tridium" vendorVersion="3.0" />
  <dependency name="devDriver" vendor="Tridium" vendorVersion="3.2" />
  <dependency name="platform" vendor="Tridium" vendorVersion="3.0" />
  <dependency name="control" vendor="Tridium" vendorVersion="3.0" />

  <!-- Packages -->
  <package name="com.yourCompany.yourDriver" doc="true"/>
  <package name="com.yourCompany.yourDriver.comm" doc="true"/>
  <package name="com.yourCompany.yourDriver.comm.req" doc="true"/>
  <package name="com.yourCompany.yourDriver.comm.rsp" doc="true"/>
  <package name="com.yourCompany.yourDriver.discover" doc="true"/>
  <package name="com.yourCompany.yourDriver.identify" doc="true"/>
  <package name="com.yourCompany.yourDriver.point" doc="true"/>
</module>

```

Please remember to save the file!

Create an Initial module-include.xml File

Next, open the empty module-include.xml file and add the following:

```

<!-- Module Include File -->

<!-- Types -->
<types>
  <!-- Type Example:
  <type name="YourClass" class="com.yourDriver.BYourClass"/>
  -->
</types>

```

Chapter 1 - Create Your Driver's Ping Request

The first thing you should do to create a driver using the developer driver framework is create your driver's **ping request**. The **ping request** represents the data that the platform computer must transmit over the field-bus to a particular unit of the external equipment (the field-device) in order to determine whether or not the particular unit of external equipment (field-device) is currently online and ready to communicate.

Please follow these steps to create your **ping request** (replacing the text *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

1. Review your equipment's protocol documentation. If you are working directly for the manufacturer of the equipment, then they should be able to provide you with one or more documents that describes the way in which the equipment communicates, plus the structure of the data that the equipment expects to see on the field-bus. If you have purchased this equipment then you will need to negotiate with the equipment's manufacturer in order to gain access to the equipment's protocol.
2. Pick a 'ping' message from your protocol. After reviewing the equipment's protocol documentation, choose a message from the protocol that looks like the simplest message to which a unit of the equipment (field-device) will respond.
3. Make a Java class that extends **BDdfPingRequest**. Name it **BYourDriverPingRequest**. Create this in the package named **com.yourCompany.yourDriver.comm.req**. Also add an empty slotomatic comment immediately after the opening brace for the class declaration.

To do this, create a text file named `BYourDriverPingRequest.java` in the `jarFileName/src/com/yourCompany/yourDriver/comm/req` folder. Inside the text file, start with the following text:

Please replace *yourCompany* and *yourDriver* in both the file name and the following text, as previously described.

```
package com.yourCompany.yourDriver.comm.req;

import javax.baja.sys.*;

import com.tridium.ddf.comm.req.*;
import com.tridium.ddf.comm.rsp.*;
import com.tridium.ddf.comm.*;

import com.yourCompany.yourDriver.identify.*;

public class BYourDriverPingRequest
    extends BDdfPingRequest
{
    /*-
    class BYourDriverPingRequest
    {
    }
    -*/
}
```

4. Override the `toByteArray` method. Build the byte array following your protocol's ping message.
5. Inside the body of the `toByteArray` method, you will need to construct a Java byte array and return it. The next step will further describe how to do this.

To add the `toByteArray` method, add the following lines of text on the line that is immediately above the last, closing brace. A closing brace looks like this: }


```
public byte[] toByteArray()
{
}
```

6. Assume that any data that you need in order to construct your byte array to return from the `toByteArray` method/function is the value in a frozen property on another, *device Id* class that extends *BDdfldParams* (we'll further discuss the *device Id* class in the next chapter). Please do not be overwhelmed at the mention of these. Suffice it to say, you will soon create another class that we will call a *device id*. On the *device id*, you will define one or more frozen properties that uniquely identify a particular unit of equipment (*field-device*) on the field-bus.

Frozen properties are a special kind of Niagara AX property on a Niagara AX component that you can easily access from Java source code. You will make the *device id* class in subsequent sections of this document. For now, please assume that you have already created it.

However, when you create this *device id* class (please do not worry about creating it now!!!), you will define some frozen properties on the class. More specifically, you will define the frozen properties whose values you will need in order to construct the byte array that this `toByteArray` method/function will return. For example, if you need something that your protocol document might call the unit number in order to build a request message, then you will later add a property called *unitNumber* to your *device id*.

To do that, you will add a special comment just after the class statement in the Java file. After doing that, you will run a Niagara AX development utility called **slotomatic** that will parse the comment and add some Java code to your file -- the Java code necessary to add the property to the Niagara AX structure that the Java file defines. More specifically, the **slotomatic** utility will generate a method (function) called `getUnitNumber` that will return a Java int, or another type as you would have defined in the special comment. By the way, let's call that special comment a **slotomatic statement**.

In light of all this discussion, please finish updating the `toByteArray` method to return a byte array that matches the description that your protocol document defines for the message that you choose to be the ping request. Please follow this example as a guide:

```
public byte[] toByteArray()
{
    // In the developer driver framework, all requests are automatically
    // Assigned a deviceId when they are created. The developer driver
    // Framework calls this method (function) after it creates the
    // Request, therefore this particular request has already been
    // Assigned a deviceId. The deviceId will be an instance of
    // BYourDriverDeviceId - that is how the developer driver works!

    BYourDriverDeviceId deviceId =
        (BYourDriverDeviceId)deviceId();

    final byte SOH = 0x01;
    final byte EOT = 0x04;
    // In this hypothetical example, the protocol document would
    // Indicate that all requests start with a hex 01 byte and
    // All requests end with a hex 04 byte.
    // So, after the hex 01, the protocol expects a number between
    // 0 and 255 to identify the device, followed by some ASCII
    // Characters ("ping" in this case), followed by the hex 04
    // Terminator byte.
    return new byte[]{
        SOH,
        (byte)deviceId.getUnitNumber(),
    }
```

```
(byte)'\p',  
(byte)'\i',  
(byte)'\n',  
(byte)'\g',  
    EOT};  
}
```

7. Override the processReceive method but simply return null for now. We will revisit this later. The developer driver framework calls the toByteArray method (function), transmits the resulting byte array onto the field-bus, looks for incoming data frames, and passes them to this method (until this method returns a response (not null), throws an exception, or times out. We'll discuss this in further detail later. For now, please do as follows:

```
public BIDDfResponse processReceive(IDdfDataFrame recieveFrame)  
    throws DdfResponseException  
{  
    return null;  
}
```

Chapter 2 - Create a Device Id for Your Driver & Associate it to Your Ping Request

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

1. Make a class named `BYourDriverDeviceId` that extends `BDdfDeviceId`. Create this in a package named ***com.yourCompany.yourDriver.identify***. Add a *slotomatic* properties declaration and declare any properties that you needed for coding the `toByteArray` method in the previous chapter. In the property declaration for each property, add the `MGR_INCLUDE` slot facet as shown in the following examples.

NOTE: The *Dev* in `BDdfDeviceId`, and in many other locations throughout the developer driver, stands for *Developer*.

To do this, create a text file named ***BYourDriverDeviceId.java*** in the ***jarFileName/src/com/yourCompany/yourDriver/identify*** folder. Inside the text file, start with the text that follows. The multi-line comment that immediately follows the Java class declaration is the Niagara ***slotomatic*** declaration. In the properties section, declare any properties that you needed for the `toByteArray` method in the previous chapter.

```
package com.yourCompany.yourDriver.identify;

import javax.baja.sys.*;

import com.tridium.ddf.identify.*;

import com.yourCompany.yourDriver.comm.req.*;

public class BYourDriverDeviceId
    extends BDdfDeviceId
{
    /*-
    class BYourDriverDeviceId
    {
        properties
        {
            unitNumber : int
                -- This is the unitNumber in our hypothetical protocol.
                default{[0]}
                slotfacets{[MGR_INCLUDE]}
        }
    }
    -*/
}
```

2. Override the `'getPingRequestType()'` method and return `BYourDriverPingRequest.TYPE`. To do this, insert the following text to the line immediately before the closing brace.

```
public Type getPingRequestType( )
{
    return BYourDriverPingRequest.TYPE;
}
```

3. Run the Niagara AX **slot** utility on your driver.

*To do this, open a Niagara Console and do the following, replacing **directoryAboveYourCompanyYourDriverFolder** with the path to the folder that contains the outer-most **yourCompanyYourDriver** folder, as created in the [Preparation](#) section.*

```
d:\Niagara-3.2.x> cd d:/directoryAboveYourCompanyYourDriverFolder
d:\directoryAboveYourCompnayYourDriverFolder> slot -mi yourCompanyYourDriver
```

The slotomatic utility parses the slotomatic comment that you added to the top of the **BYourDriverPingRequest** and **BYourDriverDeviceId** classes and generates the boiler-plate code necessary to add frozen properties to those classes. The **-mi** option works in Niagara AX versions 3.2 and later. It causes the slotomatic utility to also add an entry to your driver's **module-include.xml** file that is necessary for each Niagara AX type in your driver.

4. Do a full build on your driver and resolve any compiler errors.

*To do this, open a Niagara Console from a Niagara 3.2.x release (or later) and do the following, replacing **directoryAboveYourCompanyYourDriverFolder** with the path to the folder that contains the outer-most **yourCompanyYourDriver** folder, as created in the [Preparation](#) section.*

```
d:\Niagara-3.2.x> cd d:/directoryAboveYourCompanyYourDriverFolder
d:\directoryAboveYourCompnayYourDriverFolder> build yourCompanyYourDriver full
```

5. Resolve any errors that might appear in the console window. Proceed after you achieve a completely successful build that is free of any Java and Niagara AX build errors.
6. Return to the **toByteArray** method that you created in the previous chapter. Verify that you pull any data you need to make your byte array from the variable named **yourDriverDeviceId** and that you call the *getters* that were generated automatically by the Niagara AX **slot** utility in this chapter.

Chapter 3 - Create Your Driver's Device Component & Associate it to Your Driver's Device Id

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

1. Make a class named **BYourDriverDevice** in the package named **com.yourCompany.yourDriver**.
To do this, create a text file named BYourDriverDevice.java in the jarFileName/src/com/yourCompany/yourDriver folder. Inside the text file, start with the text that follows.

```
package com.yourCompany.yourDriver;

import javax.baja.sys.*;

import com.tridium.ddf.identify.*;

import com.yourCompany.yourDriver.identify.*;

public class BYourDriverDevice
{
}
```

2. If your driver will communicate over the serial port of the station platform computer:

- o Extend BDdfSerialDevice.
- o Import com.tridium.ddfSerial.* in BYourDriverDevice.java.

```
package com.yourCompany.yourDriver;

import javax.baja.sys.*;

import com.tridium.ddf.identify.*;
import com.tridium.ddfSerial.*;

import com.yourCompany.yourDriver.identify.*;

public class BYourDriverDevice
    extends BDdfSerialDevice
{
}
```

- o Add the following lines to the <-- Dependencies --> section of the build.xml file that you created in the [Preparation](#) section.

```
<dependency name="serial" vendor="Tridium" vendorVersion="3.0" />
<dependency name="devSerialDriver" vendor="Tridium" vendorVersion="3.2" />
```

3. If your driver will communicate over Tcp/Ip through the station platform computer's LAN or network adapter then:

- o Extend BDdfTcpDevice.
- o Import com.tridium.ddflp.tcp.* in BYourDriverDevice.java.

```
package com.yourCompany.yourDriver;

import javax.baja.sys.*;

import com.tridium.ddf.identify.*;
import com.tridium.ddfIp.tcp.*;

import com.yourCompany.yourDriver.identify.*;

public class BYourDriverDevice
    extends BDdfTcpDevice
{
}
```

- Add the following line to the <-- Dependencies --> section of the build.xml file that you created in the [Preparation](#) chapter.

```
<dependency name="devIpDriver" vendor="Tridium" vendorVersion="3.2" />
```

4. If your driver will communicate over Udp/Ip through the station platform computer's LAN or network adapter then:

- Extend [BDdfUdpDevice](#).
- Import [com.tridium.ddfIp.udp.*](#) in BYourDriverDevice.java.

```
package com.yourCompany.yourDriver;

import javax.baja.sys.*;

import com.tridium.ddf.identify.*;
import com.tridium.ddfIp.udp.*;

import com.yourCompany.yourDriver.identify.*;

public class BYourDriverDevice
    extends BDdfUdpDevice
{
}
```

- Add the following line to the <-- Dependencies --> section of the build.xml file that you created in the [Preparation](#) chapter.

```
<dependency name="devIpDriver" vendor="Tridium" vendorVersion="3.2" />
```

5. In any case (serial, Tcp/Ip, or Udp/Ip device), add a slotomatic properties declaration and re-define the 'deviceId' property as follows (make sure you give it the MGR_INCLUDE slotfacet). Place the following text on the line immediately following the opening brace for the Java class declaration. This is the Niagara **slotomatic** declaration. The *properties* section redefines the *deviceId* property. The *deviceId* type should be *BDdfIdParams*. The *deviceId*'s default value should be an instance of *BYourDriverDeviceId*.

```

/*-
class BYourDriverDevice
{
    properties
    {
        deviceId : BDdfIdParams
        -- This plugs in an instance of yourDriver's
        -- device id as this device's deviceId
        default {[new BYourDriverDeviceId()]}
        slotfacets{[MGR_INCLUDE]}
    }
}
-*/

```

6. Place the following text on the line immediately following the slotomatic declaration that you just added. We will revisit this method (function) later in the tutorial.

```

public Type getNetworkType()
{ // We will soon create a driver network and return
  // The driver network type instead of null
  return null;
}

```

7. Do not forget to save the BYourDriverDevice.java file.
8. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

Summary

In Chapter 1 you created a ping request class and started defining the outgoing frame. In Chapter 2 you created a deviceId class that feeds data to your ping request. In Chapter 3 you created a device and put a deviceId property on your device. You have just used a deviceId to glue your Niagara AX device to your ping request class.

At this point, you should have gained some familiarity with your protocol's messaging format. By defining the toByteArray method in Chapters 1 and 2 above, you have already let Niagara know how to frame an outgoing request. Next, you need to let Niagara know how to 'frame' up incoming messages that Niagara receives back from the field bus.

Chapter 4 - Create Your Driver's Receiver

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section.

In this section, the tutorial will show you how to configure your driver to receive incoming messages from your field-bus. This is accomplished by analyzing the incoming data and recognizing the beginning and ending of messages. The beginning of incoming messages is usually determined by recognizing a particular pattern of one or more bytes within the data that is being received. The exact pattern and circumstances differs for each protocol. Some protocols place a byte or series of bytes immediately following the beginning pattern that indicates the size of the incoming message. To determine the ending of an incoming message, some protocols will define this in terms of a length. As just mentioned, the length is sometimes transmitted just after the beginning sequence of bytes. Other protocols that do not include a length, count, or message size in the request feature another pre-defined sequence of bytes as the ending sequence. Some very simple protocols might state that all incoming messages are the same, fixed size.

Note: In Udp/Ip, incoming messages are automatically delivered to the software from the operating system. This is possible because Udp/Ip features the distinct notion of data packets, also known as datagrams. If your driver communicates over Udp/Ip then you probably will not need to define the incoming message structure. It is possible that a Udp/Ip driver will not need to define a custom receiver at all. **Therefore, if your driver communicates over Udp/Ip then please skip this chapter and proceed to the next chapter.**

1. Make a class named **BYourDriverReceiver** that extends **BDdfTcpReceiver** (if the protocol of your driver is Tcp/Ip based) or **BDdfSerialReceiver** (if the protocol of your driver is serial or serial-wireless based). Create this in the package named **com.yourCompany.yourDriver.comm**.

Please consider whether you need to make your class extend **BDdfSerialReceiver** or **BDdfTcpReceiver**, declare the corresponding package (**com.tridium.ddfSerial.comm.*** or **com.tridium.ddfIp.tcp.comm.***) among your import statements, and change the **extends** clause accordingly in your file (to extend either **BDdfSerialReceiver** or **BDdfTcpReceiver**).

To do this, create a text file named **BYourDriverReceiver.java** in the **jarFileName/src/com/yourCompany/yourDriver/comm** folder. Inside the text file, start with the text that follows.

```
package com.yourCompany.yourDriver.comm;

import javax.baja.sys.*;

import com.tridium.ddf.comm.*;
import com.tridium.ddfSerial.comm.*; // This would import com.tridium.ddfIp.tcp.comm.*
                                     // if the protocol was Tcp/Ip based.

public class BYourDriverReceiver
    extends BDdfSerialReceiver      // This would extend "BDdfTcpReceiver" if the
protocol                           // was Tcp/Ip based
{
    /*-
    class BYourDriverReceiver
    {
    }
    -*/
}
```


NOTE: This example extends *BDdfSerialReceiver* so it imports *com.tridium.ddfSerial.comm.**. In your case, if you choose to extend *BDdfTcpReceiver* then you should import *com.tridium.ddfIp.tcp.comm.** instead.

NOTE: The multi-line comment that immediately follows the Java class declaration is an empty Niagara **slotomatic** declaration.

ADVANCED TOPIC: The rest of this example works best for protocols whose data frames have a definite, identifiable beginning and ending sequence of characters (bytes). If your protocol document does not have such a notion of data frames, then you will need to override the protected IDdfDataFrame doReceiveFrame() throws Exception method, read raw data directly from your driver's field bus, analyze the data, and return an IDdfDataFrame. The doReceiveFrame method is called in a tight loop. In fact, by default, the doReceiveFrame calls the following method that should make it convenient in the event that the driver protocol supports definite, identifiable data frames.

NOTE: This is an advanced topic that will be discussed in an appendix in a future version of this documentation.

2. Override the **isStartOfFrame** method. This is called every time a byte is received from the field bus. Return YES if the data in the frame so far looks like the start of a receive message according to your driver's protocol. Return MAYBE if you need more bytes to determine this. Return NO if the data in the frame does not make sense and cannot possibly be the start sequence for a receive message in your protocol. *Here is an example that would frame up receive messages in our hypothetical protocol where receive messages start with a hex 02 byte.*

```
package com.yourCompany.yourDriver.comm;

import javax.baja.sys.*;

import com.tridium.ddf.comm.*;
import com.tridium.ddfSerial.comm.*; // This would import com.tridium.ddfIp.tcp.comm.*
                                     // if the protocol was Tcp/Ip based.

public class BYourDriverReceiver
    extends BDdfSerialReceiver      // This would extend "BDdfTcpReceiver" if the
protocol                            // was Tcp/Ip based
{
    /*-
    class BYourDriverReceiver
    {
    }
    -*/

    public int isStartOfFrame(IDdfDataFrame frameSoFar)
    {
        if (frameSoFar.getFrameBytes()[0]==0x02)
            return YES;
        else
            return NO;
    }
}
```

We understand that chances are, your protocol is much more complicated than this. However, the general principle still applies no matter how complicated the protocol is (provided that it still features the notion of data frames). Analyze the given frame and return YES if the bytes that are in the frame so far are the beginning sequence of frames in your protocol, NO if not, or MAYBE if you need more bytes before being able to decide

either way.

3. Override the **isCompleteFrame** method.

After your **isStartOffFrame** method returns YES, the **isCompleteFrame** method is then called for every byte that is received. Return true if the data in the frame looks like a completed, receive frame in your protocol. Return false if you need more data. If all of a sudden, you decide that the data in the frame is invalid, return true for now (the **checkFrame** method (that you may also override) will take care of that scenario.

```
package com.yourCompany.yourDriver.comm;

import javax.baja.sys.*;

import com.tridium.ddf.comm.*;
import com.tridium.ddfSerial.comm.*; // This would import com.tridium.ddfIp.tcp.comm.*
                                     // if the protocol was Tcp/Ip based.

public class BYourDriverReceiver
    extends BDdfSerialReceiver      // This would extend "BDdfTcpReceiver" if the
protocol                            // was Tcp/Ip based
{
    /*-
    class BYourDriverReceiver
    {
    }
    -*/

    public int isStartOfFrame(IDdfDataFrame frameSoFar)
    {
        if (frameSoFar.getFrameBytes()[0]==0x02)
            return YES;
        else
            return NO;
    }

    public boolean isCompleteFrame(IDdfDataFrame frameSoFar)
    { // This returns true if the last valid byte is 0x04.
      // Keep in mind, Java arrays start at index 0.
      // There is one caveat in the hypothetical protocol, the
      // Length of the response is always greater than 2 bytes.
      // This allows for the second byte to be 0x04 as necessary,
      // To indicate that the response came from unit 0x04
      return
          frameSoFar.getFrameSize()>2 &&                                // Length must be more
than 2
          frameSoFar.getFrameBytes()[frameSoFar.getFrameSize()-1]==0x04; // Frame must end
with 0x04
    }
}
```

4. Override the **checkFrame** method. This is called after your **isCompleteFrame** method returns true. Run the bytes in the given frame through any validation or checksum logic that your protocol requires. Return true if you can verify the frame is valid. Return false if not. Note that if your java code returned true in step 3 because it all of sudden decided invalid data in the frame, then your **checkFrame** method should also recognize this and return false.

```

package com.yourCompany.yourDriver.comm;

import javax.baja.sys.*;

import com.tridium.ddf.comm.*;
import com.tridium.ddfSerial.comm.*; // This would import com.tridium.ddfIp.tcp.comm.*
                                     // if the protocol was Tcp/Ip based.

public class BYourDriverReceiver
    extends BDdfSerialReceiver        // This would extend "BDdfTcpReceiver" if the
protocol                             // was Tcp/Ip based
{
    /*-
    class BYourDriverReceiver
    {
    }
    -*/

    public int isStartOfFrame(IDdfDataFrame frameSoFar)
    {
        if (frameSoFar.getFrameBytes()[0]==0x02)
            return YES;
        else
            return NO;
    }

    public boolean isCompleteFrame(IDdfDataFrame frameSoFar)
    { // This returns true if the last valid byte is 0x04.
      // Keep in mind, Java arrays start at index 0.
      // There is one caveat in the hypothetical protocol, the
      // Length of the response is always greater than 2 bytes.
      // This allows for the second byte to be 0x04 as necessary,
      // To indicate that the response came from unit 0x04
      return
        frameSoFar.getFrameSize()>2 &&                                // Length must be more
than 2
        frameSoFar.getFrameBytes()[frameSoFar.getFrameSize()-1]==0x04; // Frame must end
with 0x04
    }

    public boolean checkFrame(IDdfDataFrame frameSoFar)
    {
        return (frameSoFar.getFrameBytes()[0]==0x02) &&
            (frameSoFar.getFrameBytes()[frameSoFar.getFrameSize()-1]==0x04);
    }
}

```

Note that if your **checkFrame** method returns True, the developer driver framework will pass the received frame to the corresponding request, perhaps the one that you created in chapter 1. Also, your **checkFrame** method will likely be much more complicated than in this example. Most serial protocols will place a checksum, circular redundancy check (CRC), or longitudinal redundancy check (LRC) as one of the last bytes in the message. Your **checkFrame** method, in that case, would implement the same algorithm, recompute the checksum, CRC, or LRC, and verify that the result equals the checksum, CRC, or LRC that was received in the data frame. This is done to guarantee data integrity -- that no bytes have been corrupted during transmission (due to static, etc.)

5. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

Chapter 5 - Create Your Driver's Ping Response

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

1. Make a class named **BYourDriverPingResponse** that extends **BDdfResponse**. Create this in a package named **com.yourCompany.yourDriver.comm.rsp**. Add an empty slotomatic statement too. *To do this, create a text file named **BYourDriverPingResponse.java** in the **jarFileName/src/com/yourCompany/yourDriver/comm/rsp** folder. Inside the text file, start with the text that follows.*

```
package com.yourCompany.yourDriver.comm.rsp;

import javax.baja.sys.*;

import com.tridium.ddf.comm.rsp.*;

public class BYourDriverPingResponse
    extends BDdfResponse
{
    /*-
    class BYourDriverPingResponse
    {
    }
    -*/
}
```

NOTE: The multi-line comment that immediately follows the Java class declaration is an empty Niagara **slotomatic** declaration.

2. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).
For now, you have been intentionally instructed to create empty response class. Later you will learn about more that you can do with the response class. Fortunately, an empty response class often works sufficiently for *pinging* in a simple request-response transactions. For now, however, let us not worry about parsing the data that might be in the ping response. It is sufficient to simply know that we received a response because at the very least, we can conclude that the external equipment is online, otherwise, it would not have responded at all.
3. Next, please open **BYourDriverPingRequest.java** in your text editor.
4. Add the following import statement to the imports of **BYourDriverPingRequest.java**:

```
import com.yourCompany.yourDriver.comm.rsp.*;
```

5. Update the processReceive method in your ping request class (**BYourDriverPingRequest**). Return an instance of **BYourDriverPingResponse**.
- 6.

```

package com.yourCompany.yourDriver.comm.req;

import javax.baja.sys.*;

import com.tridium.ddf.comm.req.*;
import com.tridium.ddf.comm.rsp.*;
import com.tridium.ddf.comm.*;

import com.yourCompany.yourDriver.identify.*;
import com.yourCompany.yourDriver.comm.rsp.*;

public class BYourDriverPingRequest
    extends BDdfPingRequest
{
    /*-
    class BYourDriverPingRequest
    {
    }
    -*/

}

...

/**
 * For this example, we will assume that the mere fact that a data
 * frame was received after transmitting this response means that
 * the equipment must have responded to the request. Since in
 * Niagara AX, the primary purpose of a ping request-response
 * transaction is to determine whether or not the corresponding
 * field-device is online, then this will suffice.
 */
public BIDdfResponse processReceive(IDdfDataFrame recieveFrame)
    throws DdfResponseException
{
    return new BYourDriverPingResponse();
}
}

```

ADVANCED: After transmitting the bytes for your request (the bytes returned by your request's **toByteArray** method), the developer driver framework will pass all frames that it receives to your request's **processReceive** method. This is done until your request returns a **BIDdfResponse** from this method or throws a **DevResponseException** (or a subclass of **DevResponseException**).

The **processReceive** method needs to take one of the following steps:

1. (ADVANCED) Ignore the frame and return null.
This could happen in more advanced protocols that are not "master-slave" by nature. In this scenario, by returning *null*, the developer driver framework will continue to pass subsequent data frames that it receives to this method.
2. (ADVANCED) Collect the frame and return a **BIDdfMultiFrameResponse**. In which case, you need to implement your own collection mechanism. For example, this could be as simple as putting them all in a **Vector** in the **BIDdfMultiFrameResponse** (until the request is considered to have *timed-out*). This could happen in more advanced protocols where the equipment might respond by sending multiple frames of data. By returning an instance of **BIDdfMultiFrameResponse**, the developer driver framework will know that your request is making progress. The developer driver framework will also

continue to pass subsequent data frames that it receives to this method (until the request is considered to have *timed-out*).

3. (TYPICAL) Return a `BIDdfResponse` for the data frame (*ADVANCED -- and any previously collected frames -- ADVANCED*) that you determine together make up a completed response.

By returning a response from your ping request's **processReceive** method, the developer driver framework will know that the corresponding field-device is online. The particular request-response transaction will be considered to be complete. No subsequent data frames will be passed to this particular request's **processReceive** method.

4. (SOMEWHAT TYPICAL) Throw a `DevResponseException` or subclass of `DevResponseException` to indicate that the frame forms a complete message but indicates an invalid condition within the frame (perhaps a frame that indicates a negative acknowledgement or NACK).
By throwing a `DevResponseException` or a subclass of `DevResponseException`, the developer driver framework will know that the corresponding field-device is not completely online. The particular request-response transaction will be considered to be complete. No subsequent data frames will be passed to this particular request's **processReceive** method.

WARNING: In the middle two scenarios, if you need to keep the bytes that are received then it is important that you retain only a copy the frame's bytes. The frame's actual byte array could be a direct reference to an internal buffer in the receiver.

7. Run `slotomatic` and perform a full build on your driver (as described in [Chapter 2](#)).

At this point, not only have you gained familiarity with your protocol and defined an outgoing request, you have now told Niagara how to parse incoming data frames. You have also defined the incoming response class and associated it with the outgoing request. Next, you will further associate the receiver that you created in Chapter 4 to the device you created in Chapter 3 or to your driver's network class, which you will create in chapter 7.

Chapter 6 - Create Your Driver's Communicator Component

(And Plug Your Receiver Into It)

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

1. Create a class named **BYourDriverCommunicator** that extends one of the following:
 - `com.tridium.ddflp.tcp.comm.BDdfTcpCommunicator`,
 - `com.tridium.ddfSerial.comm.singleTransaction.BDdfSerialSitCommunicator` (for master-slave serial protocols)
 - `com.tridium.ddfSerial.comm.multipleTransaction.BDdfSerialMutCommunicator` (for multiple transaction serial or likely for wireless serial radio protocols).
 - `com.tridium.ddflp.udp.comm.singleTransaction.BDdfUdpSitCommunicator` (for master-slave Udp/Ip protocols)
 - `com.tridium.ddflp.udp.comm.multipleTransaction.BDdfUdpMutCommunicator` (for multiple transaction Udp/Ip or likely for wireless protocols where the Jace interfaces over Udp/Ip to a wireless radio).

Choosing the proper class to extend could prove tricky. For Tcp/Ip protocols you should always extend **BDdfTcpCommunicator**.

However for serial protocols (or wireless protocols where a wireless router or radio connects to a serial port on the platform computer) then you will need to choose either `BDdfSerialSitCommunicator` or `BDdfSerialMutCommunicator`. If your driver's protocol is strictly master-slave in nature (that is, if your protocol indicates that the platform computer transmits serial data onto the serial port at will but the field-devices will only transmit in response to such a message) then you should extend **BDdfSerialSitCommunicator**. If your driver's serial protocol is more complicated than this then you should probably extend **BDdfSerialMutCommunicator**. You can certainly modify this later if you change your mind.

Likewise, for Udp/Ip protocols you need to choose between a `BDdfUdpMutCommunicator` or a `BDdfUdpSitCommunicator`. The choice depends on the nature of the protocol. The same rules for choosing the *Mut* version as opposed to the *Sit* version apply as for serial drivers, as explained in the previous paragraph.

2. Create the class in package **com.yourCompany.yourDriver.comm**.
 - If you decide to extend `BDdfTcpCommunicator` then please add an import statement to `import com.tridium.ddflp.tcp.comm.*;`
 - If you decide to extend `BDdfSerialSitCommunicator` then please add an import statement to `import com.tridium.ddfSerial.comm.singleTransaction.*;`
 - If you decide to extend `BDdfSerialMutCommunicator` then please add an import statement to `import com.tridium.ddfSerial.comm.multipleTransaction.*;`
 - If you decide to extend `BDdfUdpSitCommunicator` then please add an import statement to `import com.tridium.ddflp.udp.comm.singleTransaction.*;`
 - If you decide to extend `BDdfUdpMutCommunicator` then please add an import statement to `import com.tridium.ddflp.udp.comm.multipleTransaction.*;`

To create the class for ***BYourDriverCommunicator***, create a text file named ***BYourDriverCommunicator.java*** in the ***jarFileName/src/com/yourCompany/yourDriver/comm*** folder. Inside the text file, start with the text that follows. Please replace the right side of the ***extends*** clause, if necessary, as described in the previous two paragraphs.

This example extends *BDdfSerialSitCommunicator* so it imports *com.tridium.ddfSerial.comm.singleTransaction.**;

3. Add a slotomatic statement with a properties declaration.
4. Redefine the property named *receiver*. The type must be *BDdfReceiver*. The default value should be an instance of *BYourDriverReceiver*.

```
package com.yourCompany.yourDriver.comm;

import javax.baja.sys.*;

import com.tridium.ddf.comm.defaultComm.*;

import com.tridium.ddfSerial.comm.singleTransaction.*;

public class BYourDriverCommunicator
    extends BDdfSerialSitCommunicator
{
    /*-
    class BYourDriverCommunicator
    {
        properties
        {
            receiver : BDdfReceiver
                -- Plugs yourDriver's custom receiver
                -- into yourDriver's communicator
            default{[new BYourDriverReceiver()]}
        }
    }
    -*/
}
```

5. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

Chapter 7 - Create a Device Folder and a Network Component

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

Create a Device Folder Component For Your Driver

1. Create a class named **BYourDriverDeviceFolder** in package **com.yourCompany.yourDriver**
*To do this, create a text file named **BYourDriverDeviceFolder.java** in the **jarFileName/src/com/yourCompany/yourDriver** folder. Paste the following text into the **BYourDriverDeviceFolder.java** file:*

```
package com.yourCompany.yourDriver;

import com.tridium.ddf.*;

import javax.baja.sys.*;

public class BYourDriverDeviceFolder
    extends BDdfDeviceFolder
{
    /*-
    class BYourDriverDeviceFolder
    {
        properties
        {
        }
    }
    -*/
}
```

2. Run slotomatic and perform a full build on your driver.

NOTE: The device folder is practically a formality in the Niagara AX framework.

Create a Network Component For Your Driver

1. Create a class named **BYourDriverNetwork** in package **com.yourCompany.yourDriver**
*To do this, create a text file named **BYourDriverNetwork.java** in the **jarFileName/src/com/yourCompany/yourDriver** folder. Paste the text from one of the following examples into the **BYourDriverDeviceFolder.java** file.*
2. If yourDriver is a serial (or serial-wireless) driver, make your class extend **BDdfSerialNetwork** and start with the text that follows:

```

package com.yourCompany.yourDriver;

import com.tridium.ddfSerial.*;

import com.yourCompany.yourDriver.comm.*;

import javax.baja.sys.*;

public class BYourDriverNetwork
    extends BDdfSerialNetwork
{
    /*-
    class BYourDriverNetwork
    {
        properties
        {
        }
    }
    -*/
}

```

3. If yourDriver is a Tcp/Ip driver whose driver protocol communicates directly to Tcp/Ip field-devices extend **BDdfTcpNetwork** and start with the text that follows.

```

package com.yourCompany.yourDriver;

import com.tridium.ddfIp.tcp.*;

import com.yourCompany.yourDriver.comm.*;

import javax.baja.sys.*;

public class BYourDriverNetwork
    extends BDdfTcpNetwork
{
    /*-
    class BYourDriverNetwork
    {
        properties
        {
        }
    }
    -*/
}

```

4. If yourDriver is a Tcp/Ip driver whose driver protocol communicates directly to a field-gateway that has field-devices attached to it then extend **BDdfTcpGatewayNetwork** and name the class **BYourDriverGatewayNetwork** instead of just **BYourDriverNetwork**. Name the text file **BYourDriverGatewayNetwork.java** too! Also, (only if using a field-gateway) update your device from chapter 3 and make it extend **BDdfTcpDeviceBehindGateway** instead of **BDdfTcpDevice**. Start with the following text in **BYourDriverGatewayNetwork.java**:

```

package com.yourCompany.yourDriver;

import com.tridium.ddfIp.tcp.*;

import com.yourCompany.yourDriver.comm.*;

import javax.baja.sys.*;

public class BYourDriverGatewayNetwork
    extends BDdfTcpGatewayNetwork
{
    /*-
    class BYourDriverGatewayNetwork
    {
        properties
        {
        }
    }
    -*/
}

```

And then in this case, also change your driver's device (*BYourDriverDevice.java* like this:

```

package com.yourCompany.yourDriver;

import com.tridium.ddf.identify.*;
import com.tridium.ddfSerial.*;
import javax.baja.sys.*;

public class BYourDriverDevice
    // NOTE: Replace BDdfTcpDevice with BDdfTcpDeviceBehindGateway in the extends clause
    extends BDdfTcpDevice BDdfTcpDeviceBehindGateway
{
    /*-
    class BYourDriverDevice
    {
        properties
        {
            deviceId : BDdfIdParams
                -- This plugs in an instance of yourDriver's
                -- device id as this device's deviceId
                default {[new BYourDriverDeviceId()]}
        }
    }
    -*/
}

```

5. If yourDriver is a Udp/Ip driver whose driver protocol communicates directly to Udp/Ip field-devices extend **BDdfUdpNetwork** and start with the text that follows.

```

package com.yourCompany.yourDriver;

import com.tridium.ddfIp.udp.*;

import com.yourCompany.yourDriver.comm.*;

import javax.baja.sys.*;

public class BYourDriverNetwork
    extends BDdfUdpNetwork
{
    /*-
    class BYourDriverNetwork
    {
        properties
        {
        }
    }
    -*/
}

```

6. If yourDriver is a Udp/Ip driver whose driver protocol communicates directly to a field-gateway that has field-devices attached to it then extend **BDdfUdpGatewayNetwork** and name the class **BYourDriverGatewayNetwork** instead of just **BYourDriverNetwork**. Name the text file **BYourDriverGatewayNetwork.java** too! Also, (only if using a field-gateway) update your device from chapter 3 and make it extend **BDdfUdpDeviceBehindGateway** instead of **BDdfUdpDevice**. Start with the following text in **BYourDriverGatewayNetwork.java**:

```

package com.yourCompany.yourDriver;

import com.tridium.ddfIp.udp.*;

import com.yourCompany.yourDriver.comm.*;

import javax.baja.sys.*;

public class BYourDriverGatewayNetwork
    extends BDdfUdpGatewayNetwork
{
    /*-
    class BYourDriverGatewayNetwork
    {
        properties
        {
        }
    }
    -*/
}

```

And then in this case, also change your driver's device (**BYourDriverDevice.java** like this:

```

package com.yourCompany.yourDriver;

import com.tridium.ddf.identify.*;
import com.tridium.ddfSerial.*;
import javax.baja.sys.*;

public class BYourDriverDevice
    // NOTE: Replace BddfUdpDevice with BddfUdpDeviceBehindGateway in the extends clause
    extends BddfUdpDevice BddfUdpDeviceBehindGateway
{
    /*-
    class BYourDriverDevice
    {
        properties
        {
            deviceId : BddfIdParams
            -- This plugs in an instance of yourDriver's
            -- device id as this device's deviceId
            default {[new BYourDriverDeviceId()]}
        }
    }
    -*/
}

```

7. Regardless of which sample code example you choose to start with in **BYourDriverNetwork.java**, add the following methods to your network on the line immediately above the closing brace for the class:

```

public Type getDeviceType()
{
    return BYourDriverDevice.TYPE;
}
public Type getDeviceFolderType()
{
    return BYourDriverDeviceFolder.TYPE;
}

```

8. Open *BYourDriverDevice.java* and modify the *getNetworkType* method to have it return *BYourDriverNetwork.TYPE* or *BYourDriverGatewayNetwork.TYPE* (if you choose to use a Tcp/Ip gateway network for your driver).

```

public Type getNetworkType()
{
    return BYourDriverNetwork.TYPE;
}

```

9. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

Chapter 8 - Tie Your Driver's Communicator Component to Your Driver's Tcp/Ip Device, Serial Network, or Tcp/IpGateway Component.

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

1. Redefine the **communicator** property as follows on your driver's *Tcp Device*, *Tcp Gateway Network*, *Udp Device*, *Udp Gateway Network* or *Serial Network* (which ever one applies to your driver, if you followed chapters 3 and 7 then only one of these three scenarios should apply):
 - o For a **TCP or UDP device**, your device component's slotomatic header should look something like this:

```
/*-
class BYourDriverDevice
{
  properties
  {
    communicator : BValue
    -- This plugs in an instance of yourDriver's
    -- communicator onto the device to which
    -- the Niagara station's platform will directly
    -- communicate.
    default{[ new BYourDriverCommunicator() ]}
  }
}
-*/
```

- o For a **TCP or UDP gateway network**, your gateway network component's slotomatic header should look something like this:

```
/*-
class BYourDriverGatewayNetwork
{
  properties
  {
    communicator : BValue
    -- This plugs in an instance of yourDriver's
    -- communicator onto the gateway network component.
    -- The Niagara station's platform will communicate
    -- directly to the corresponding gateway unit on the
    -- field-bus.
    default{[ new BYourDriverCommunicator() ]}
  }
}
-*/
```

- o For a **serial network**, your serial network component's slotomatic header should look something like this:

```
/*-
class BYourDriverNetwork
{
    properties
    {
        communicator : BValue
        -- This plugs in an instance of yourDriver's
        -- communicator onto the serial network component.
        -- The Niagara station's platform will communicate
        -- over a serial port that is configured on this
        -- serial network. You can look at the property
        -- sheet of this communicator to review the exact
        -- settings.
        default{[ new BYourDriverCommunicator() ]}
    }
}
-*/
```

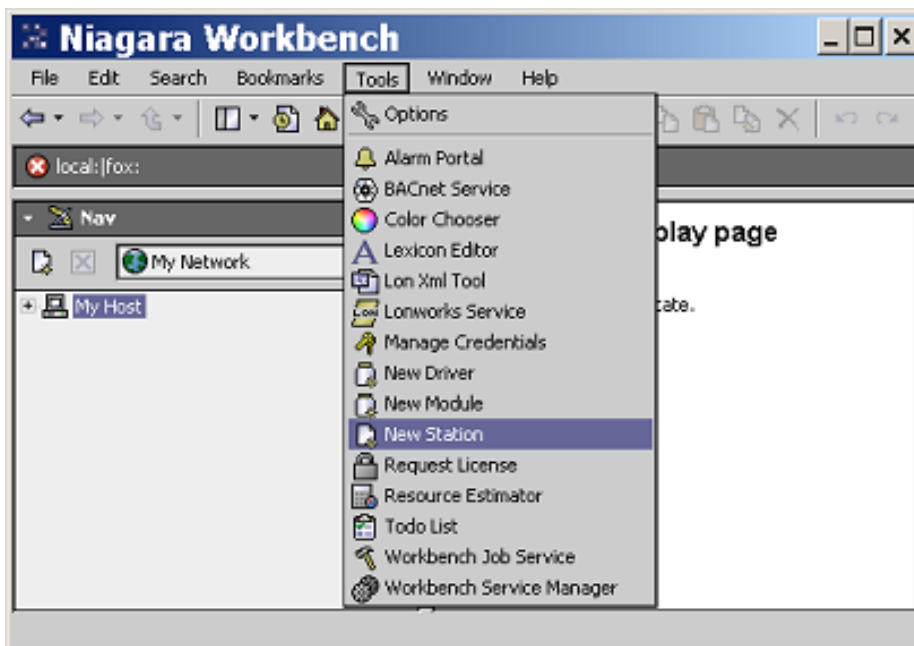
2. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

Day 1 - Review Your Progress

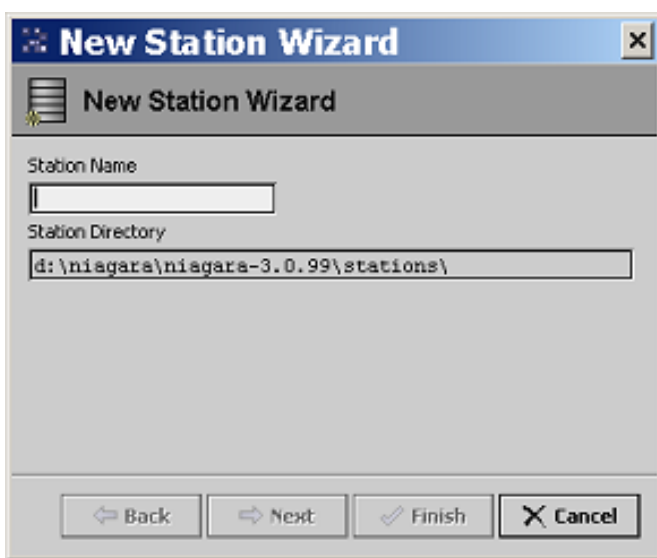
You may now begin testing your driver's functionality. Please do the following to create a station, add your driver to it, and run your station:

Create a New Station in Workbench

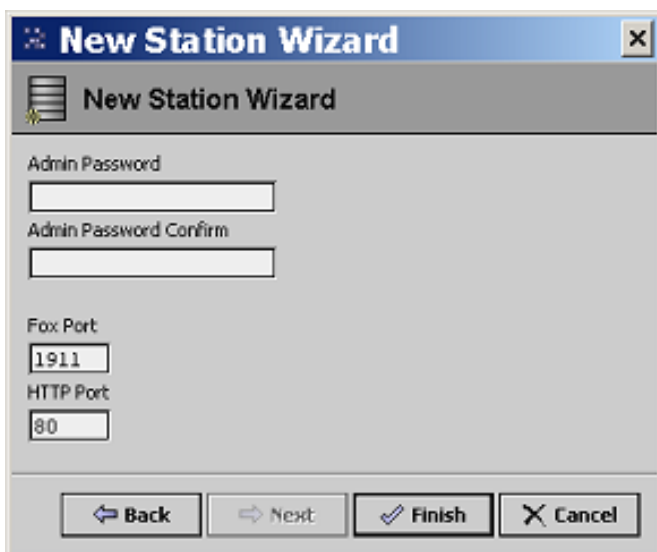
1. To be sure that your driver is completely built, Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).
2. If Workbench is open then please close it and re-open it (otherwise, it might not load the latest version of your driver).
3. On the Workbench main menu, click **Tools** and then click **New Station**. A window titled New Station Wizard should appear.



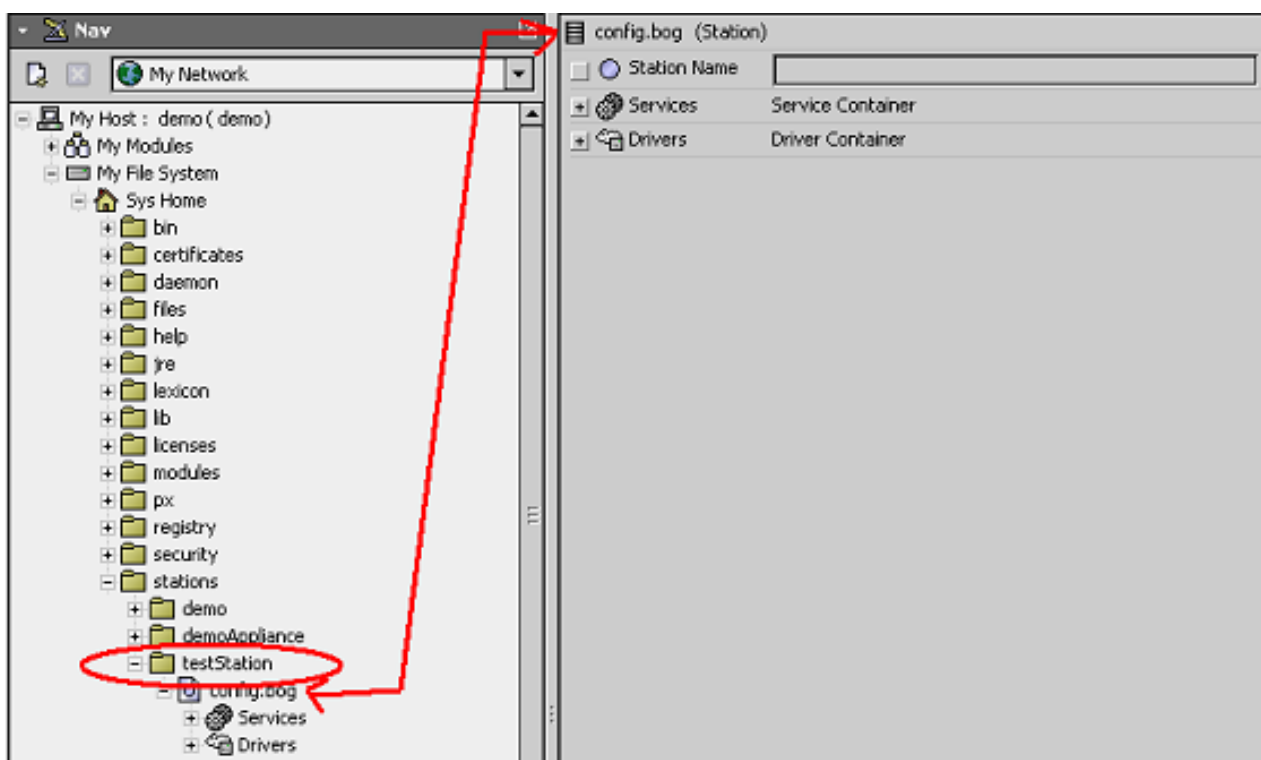
4. Enter a name (such as *TestStation*) into the **Station Name** text box.



5. Click the **Next** button. On the next screen of the wizard, you may enter an administrator password if you wish. You may also leave this blank if you'd rather.

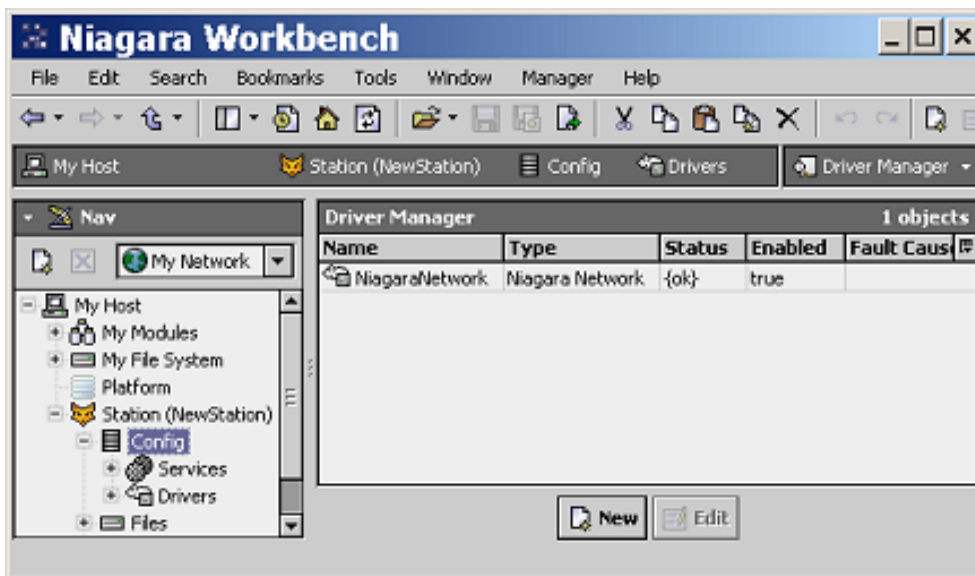


6. Click the **Finish** button. The **New Station Wizard** window will then disappear.
7. Notice that the Workbench **Nav** tree has automatically navigated to a file named *config.bog* that is directly under a folder with the same name as the one you supplied for the station name. This is your station database. Your new station is in **offline** mode. It is not actively running. It can still be configured though.

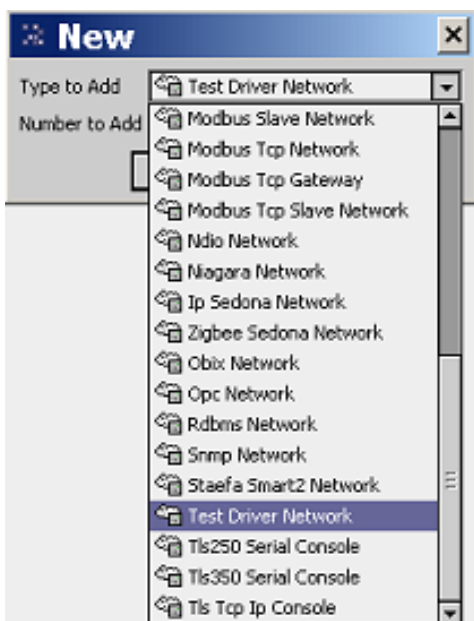


Add Your Driver's Network to the New Station

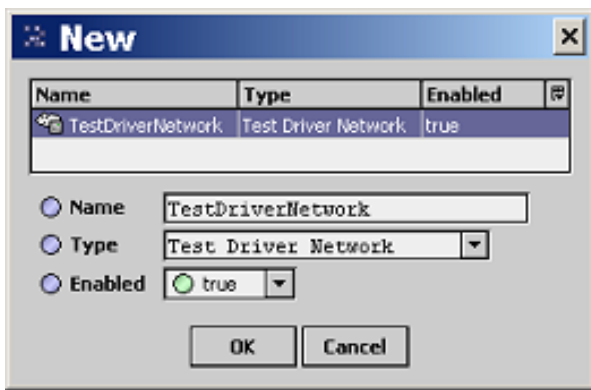
1. Expand the **config.bog** file in the **Nav** tree.
2. Double-click the **Drivers** component. The *Driver Manager* should appear in the main Workbench view area.



3. Click the **New** button that is located towards the bottom of the *Driver Manager*. A window titled New should appear. It has a drop down list box named **Type to Add** and a text box named **Number to Add**.
4. Click the drop down list box and verify that a listing for your driver's network appears in the list. Hint: If the Java file for your driver's network is named `BYourDriverNetwork.java` then you should see a listing of Your Driver Network in the list.
5. Select your driver's network from the drop down list box.



6. Click **Ok**. The **New** window disappears but another window titled **New** should appear.



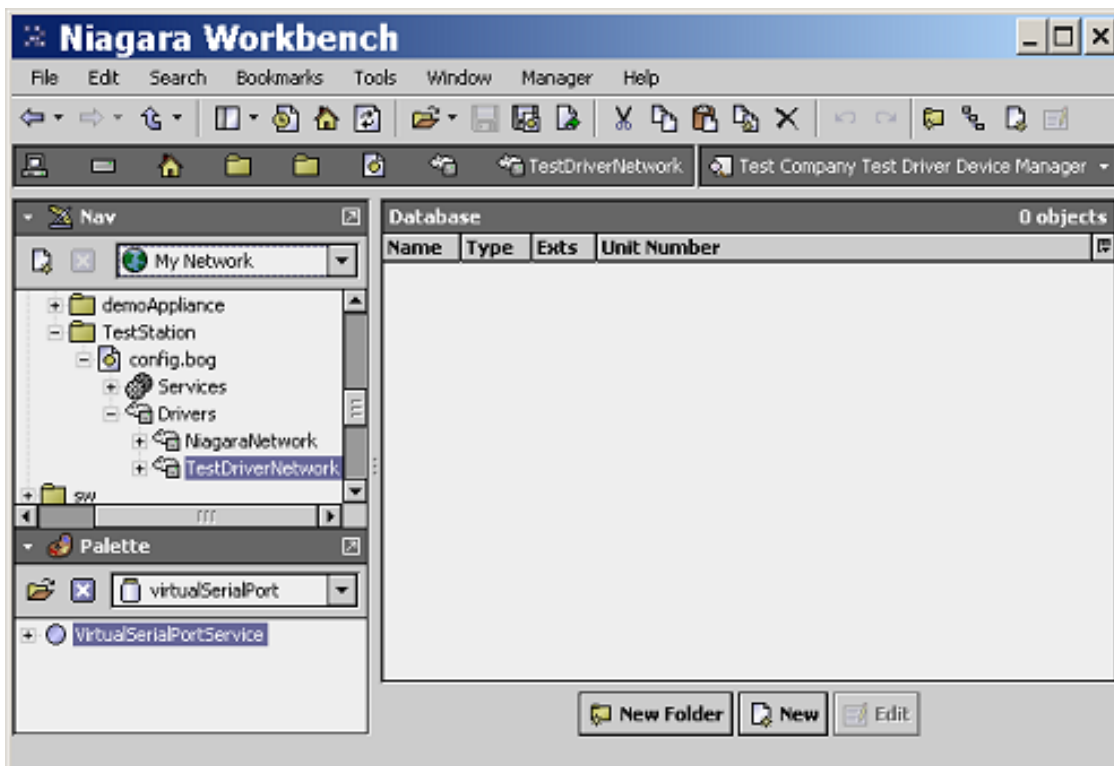
7. Click **Ok**. An instance of your driver's network has been added to the station. If you expand the *Driver* component below the *config.bog* file in the *Nav tree* you should see your network underneath. You should also see your network in the *Driver Manager's* list.

Add Your Driver's Device to Your Driver's Network That You Added to the New Station

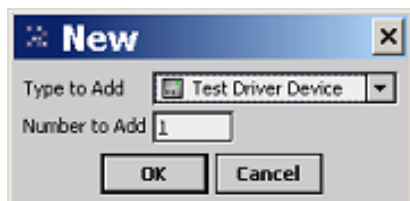
1. Double click the new instance of your driver's network that you added to the station. You may double-click it in the **Nav** tree or in the **Driver Manager**. The Device Manager for your driver should replace the *Driver Manager* in the main Workbench view area.

You should see columns in the Device Manager that correspond to the names of the properties that you added to *BYourDriverDeviceId*. This happens as a result of adding the slot facet *MGR_INCLUDE* to the *deviceId* property in *BYourDriverDevice.java* and then to the properties that you declared in *BYourDriverDeviceId.java*.

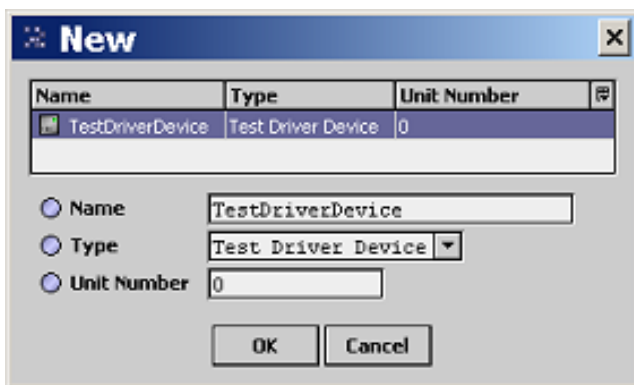
The following image shows the *Device Manager* for the *Test Driver Network* that the previous illustration added to the *TestStation*.



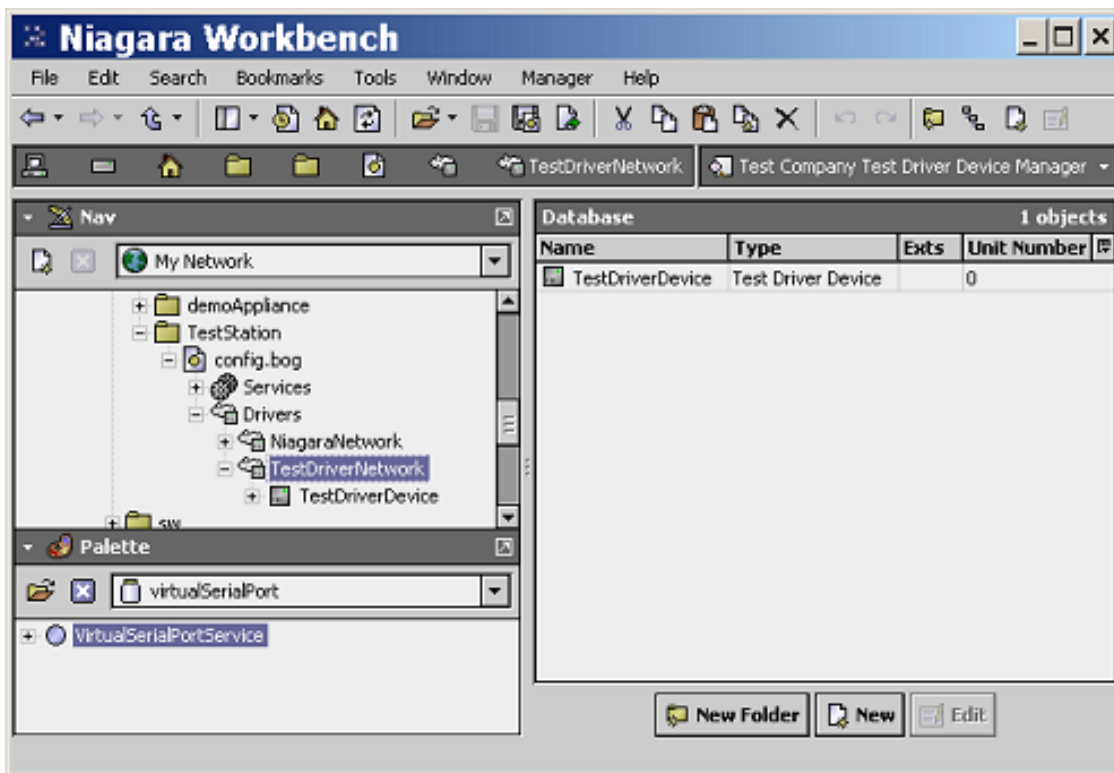
2. Click the **New** button that is located towards the bottom of the Device Manager.
 - o A window titled New should appear.
 - o This window should look nearly identical to the New window that appeared when you added your network to the new station.
 - o This New window has a drop down list box named **Type to Add** and a text box named **Number to Add**.



3. Click the drop down list box and verify that a single item is in the listing. The single item should be your driver's device. Hint: If the Java file for your driver's device is named *BYourDriverDevice.java* then you should see a listing of Your Driver Device in the list.
4. Select your driver's device from the drop down list box.
5. Click **Ok**. The **New** window disappears but another window titled **New** should appear. This window should have a text box or drop down list box for each property that you declared in *BYourDriverDeviceId* with the MGR_INCLUDE slot facet.

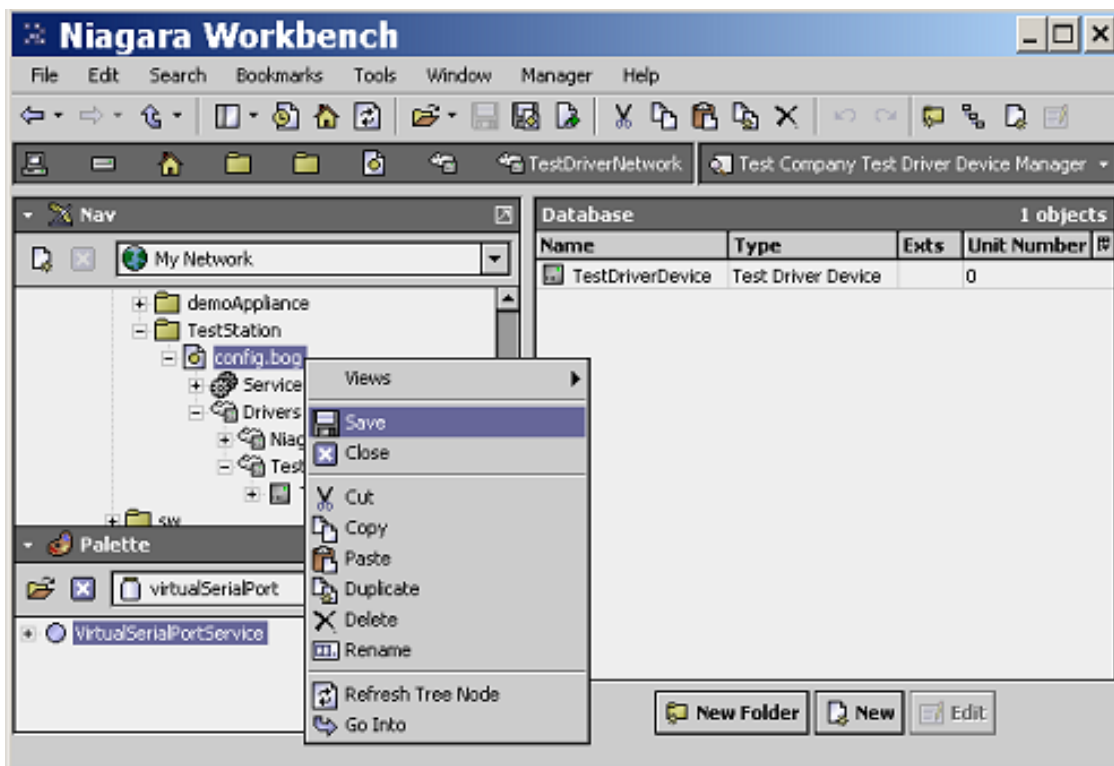


6. Enter what you believe are reasonable values into the fields for your device id. Your goal should be to identify a field-device that you would like your driver to access.
7. Click **Ok**. An instance of your driver's device has been added to the station. If you expand your driver's network component (below the *Drivers* component of the *config.bog* file) in the *Nav tree* you should see your device underneath. You should also see your device in the *Device Manager's* list.



Save your station

1. In the *Nav* tree, right click the config.bog file that is the station folder for your offline station.
2. Select Save from the pop-up menu. The following image illustrates this.



Run Your Station

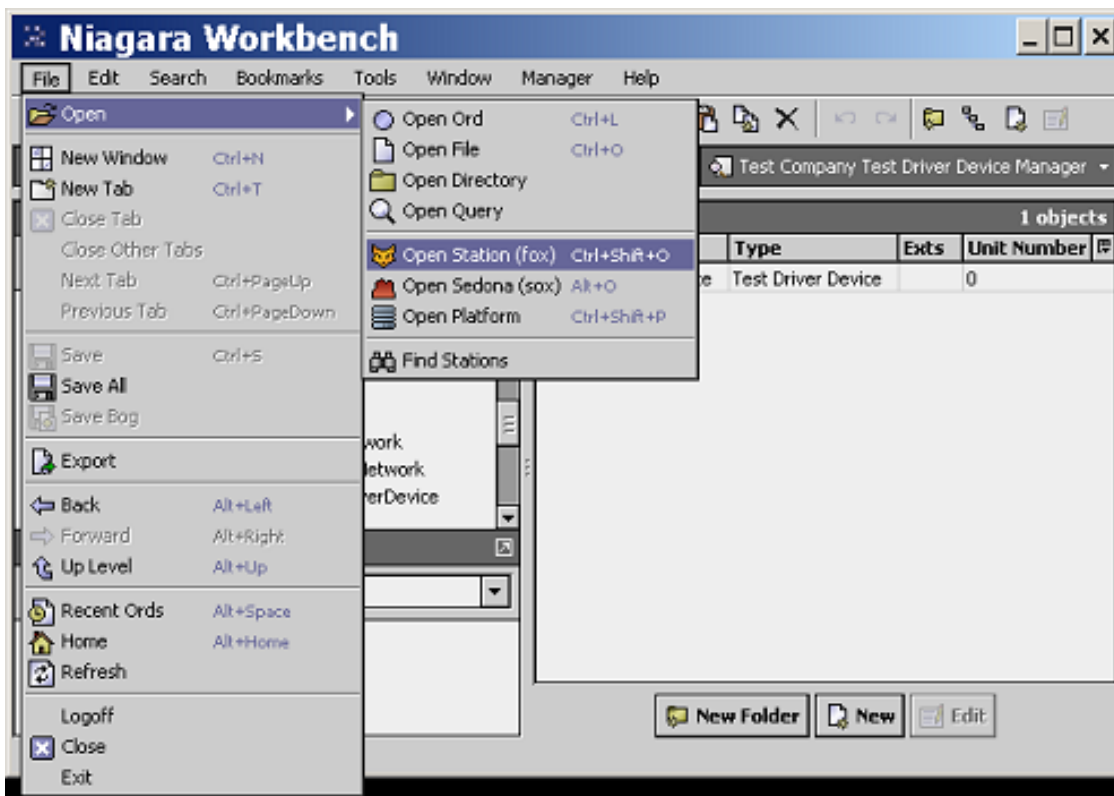
1. Open a **Niagara Console**. You typically do this by clicking the Windows Start menu, hovering over programs, then hovering over Niagara 3.2.8 (or whatever version of Niagara AX you are using), and selecting *Console*.
2. In the **Niagara Console** type `station <NewStation>` but replace `<NewStation>` with the name that you assigned to your station. Press enter after you type this text.

```
d:\Niagara-3.2.x> station TestStation
MESSAGE [12:27:06 22-Mar-07 EDT][sys.registry] Up-to-date [140ms]
MESSAGE [12:27:07 22-Mar-07 EDT][sys] Baja runtime booted ("d:\niagara\niagara-3.2.8")
MESSAGE [12:27:07 22-Mar-07 EDT][sys.registry] Loaded [390ms]
MESSAGE [12:27:08 22-Mar-07 EDT][sys] Loading "d:\niagara\niagara-3.2.8\stations\TestStation\config.
bog"...
MESSAGE [12:27:12 22-Mar-07 EDT][sys] Loaded (4166ms)
MESSAGE [12:27:16 22-Mar-07 EDT][alarm.database] Created
MESSAGE [12:27:16 22-Mar-07 EDT][sys] Services Initialized (210ms)
MESSAGE [12:27:16 22-Mar-07 EDT][sys.mixin] Updated [0ms]
WARNING [12:27:16 22-Mar-07 EDT][platform] Local daemon session not available, station not started by
niagarad
MESSAGE [12:27:16 22-Mar-07 EDT][web.server] HTTP Server started on port 80
MESSAGE [12:27:16 22-Mar-07 EDT][fox] Service started on port 1911
MESSAGE [12:27:16 22-Mar-07 EDT][sys] nre.version: 3.2.8
MESSAGE [12:27:16 22-Mar-07 EDT][sys] *** Station Started (832ms) [11938ms total] ***
niagara>
```

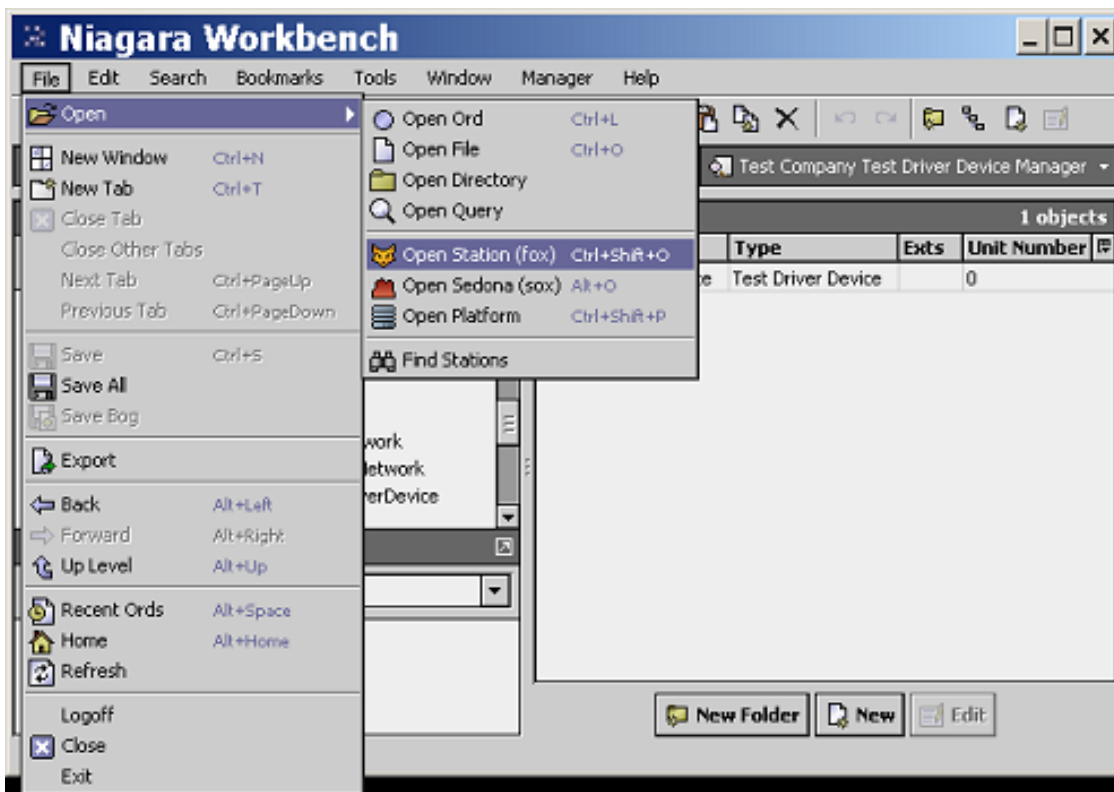
3. Your station program should begin executing. Your station is now online.

Connect to Your Online Station

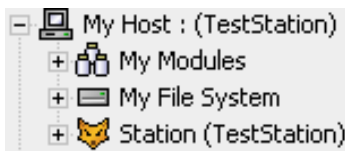
1. Return to Workbench.
2. Click the **File** menu. Hover over the **Open** menu item. Select **Open Station (fox)** in the side-menu that appears. A window titled Open Station should appear.



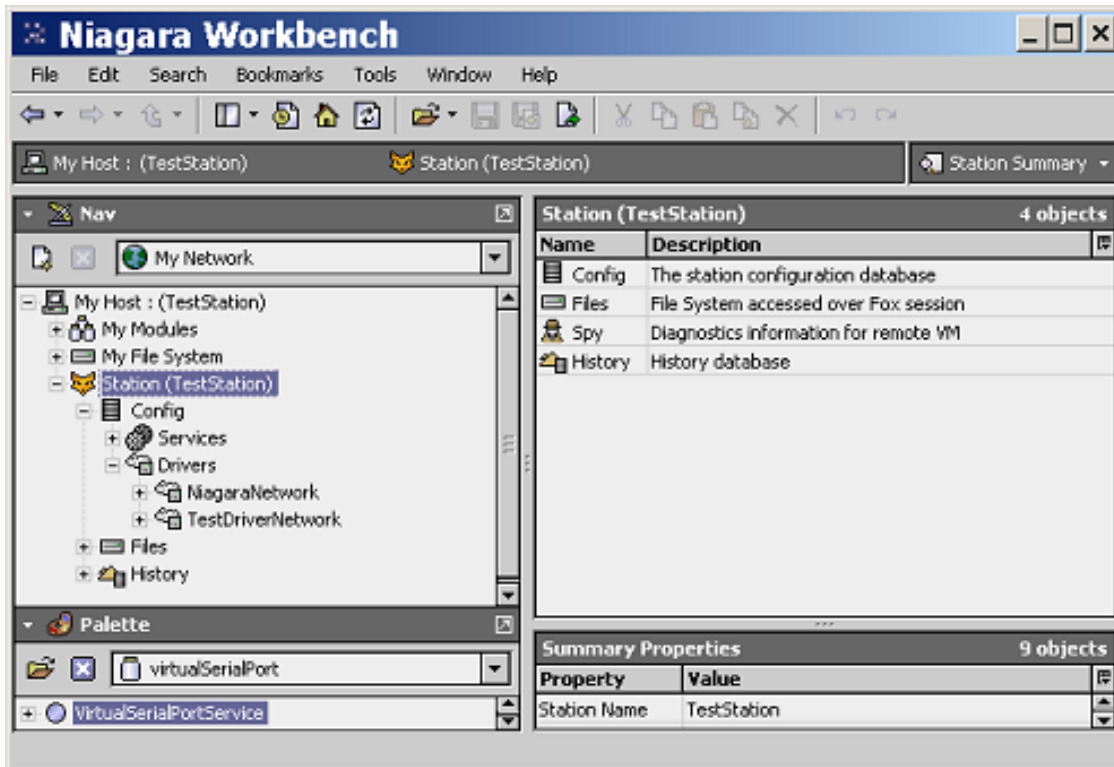
3. Leave the **Host** empty. That will cause it to default to the local host, which is your Workstation (personal computer).
4. For **User** enter **admin** (all lower case)
5. For **Password** either leave empty or enter the administrator password that you entered back when you created the station.
6. Click **Ok**.



- The **Open Station** window should disappear. After collapsing the **My File System** item that is in the Workbench **Nav** tree, the **Nav** tree should look something like the following image.

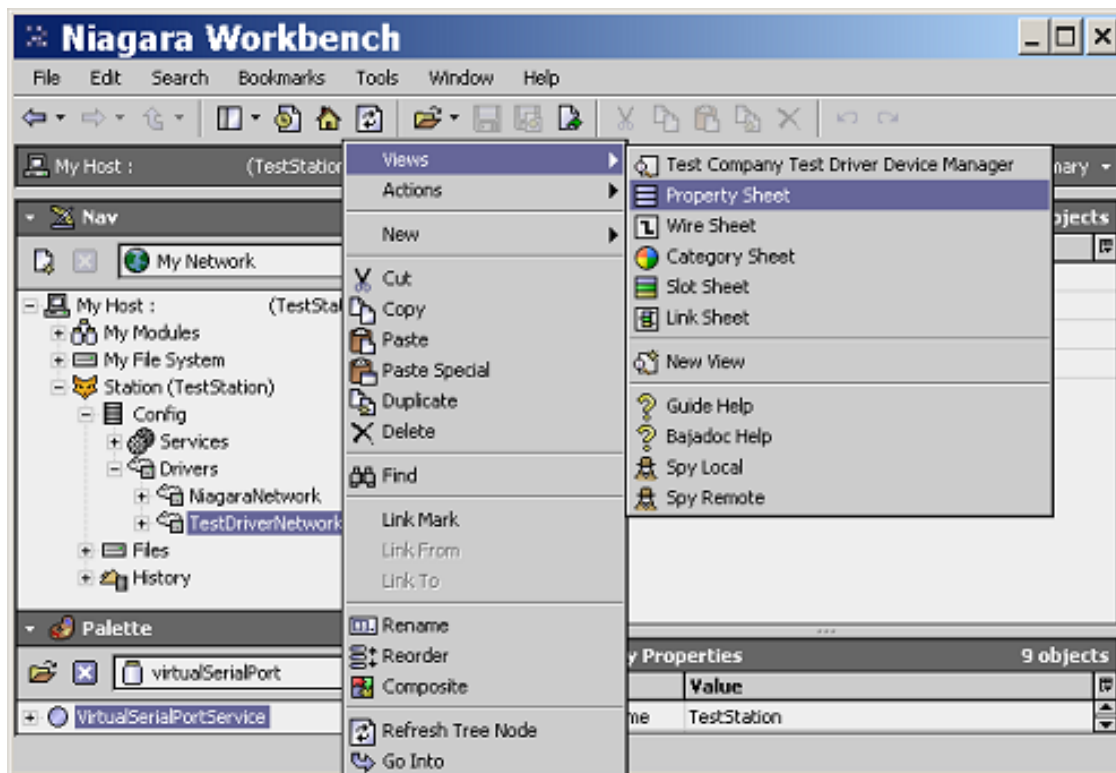


- Expand the online station component that appears under your host in the **Nav** tree.
- Expand the **Config** component of your station.
- Expand your driver's network that is under the **Config** component in your station.

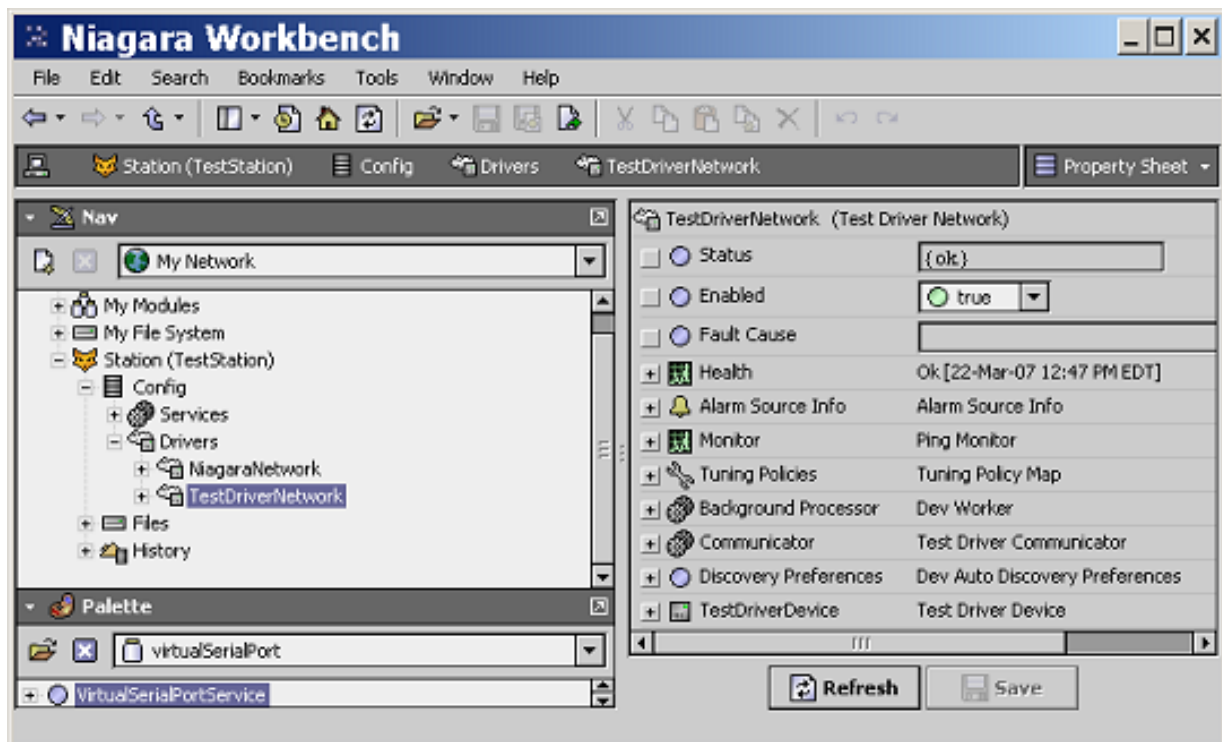


Configure the Communications Settings For Your Driver

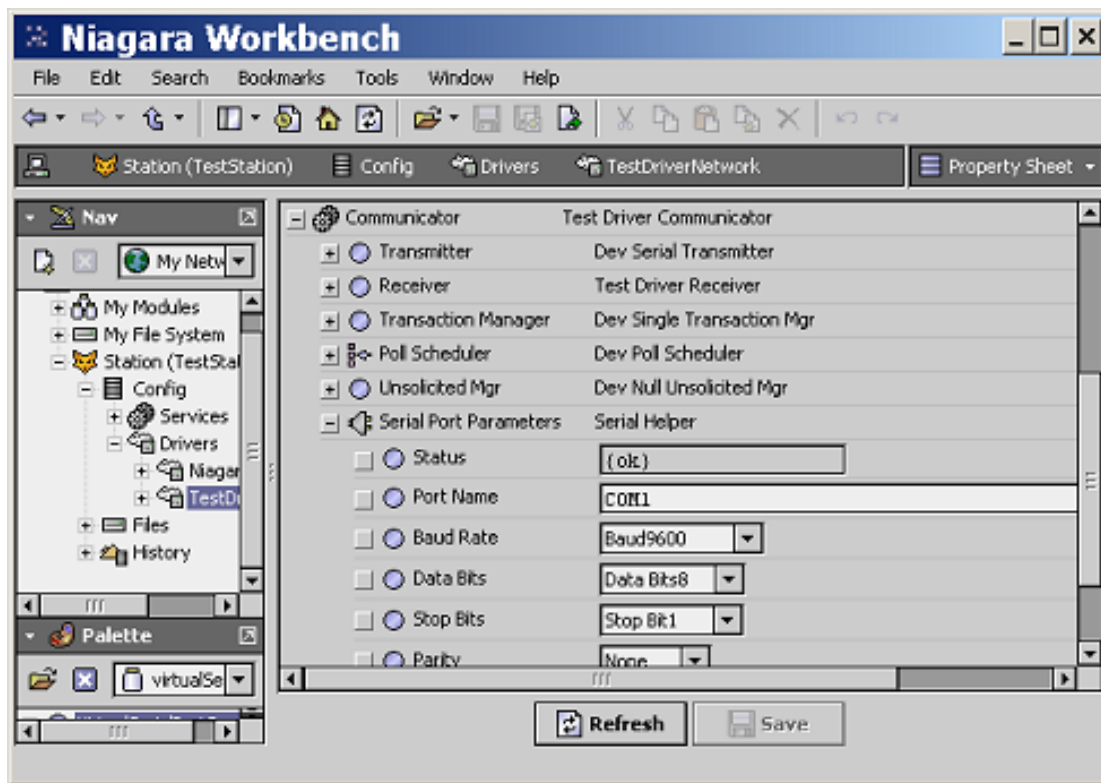
- If you created a serial device or a Tcp/Ip gateway device during the day's lesson then right-click your driver's network in the **Nav** tree.
- If you created a regular Tcp/Ip device (not a Tcp/Ip gateway device) during the day's lesson then expand your driver's network in the **Nav** tree and then right-click your driver's Tcp/Ip device.
- After right-clicking, hover over the **Views** menu item. A side-menu should appear.
- Click **Property Sheet** in the **Views** side-menu.



- The main Workbench view are should now display the properties for the component that your right-clicked.



- Notice the Communicator component appears in the *Property Sheet*. Expand it in the **Property Sheet** by clicking the plus sign that is next to it.
- Expand the **Serial Port Parameters** or the **Tcp Ip Parameters** property (which ever one is present) and enter the correct serial settings (such as comm. port, baud rate, etc.) or the correct Tcp/Ip settings (i.p. address and port of a field-device or field-gateway).

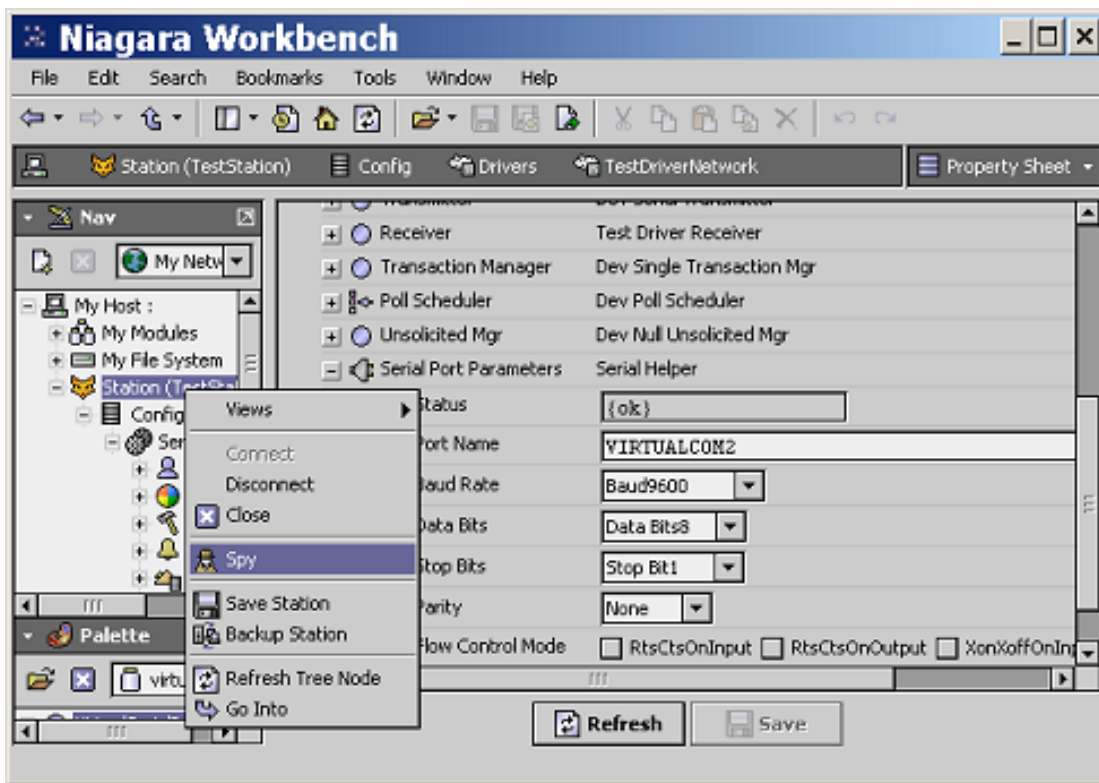


WARNING: To stop your station, type quit in its console window. Failure to do so will likely discard any modifications that you made to the station while it was online.

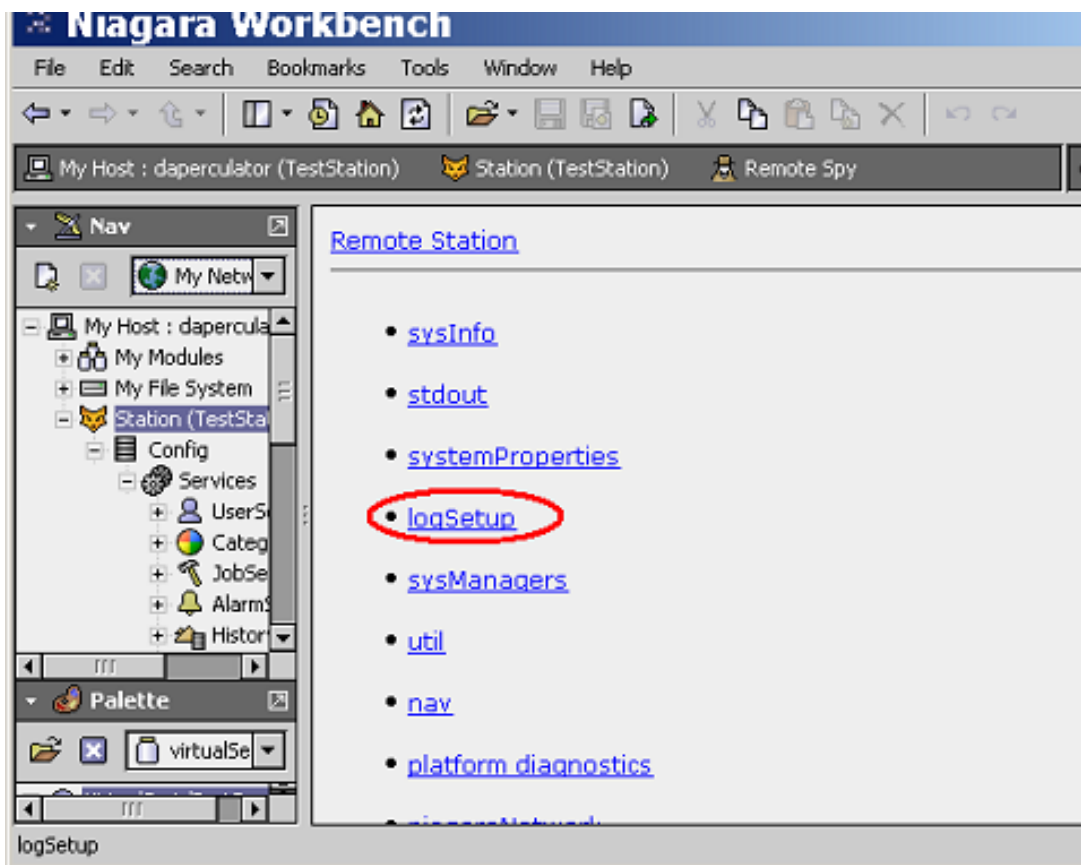
The station will automatically, periodically **ping** your device. You can also right-click your device and invoke the **ping** action.

To review the data that your driver is transmitting and receiving

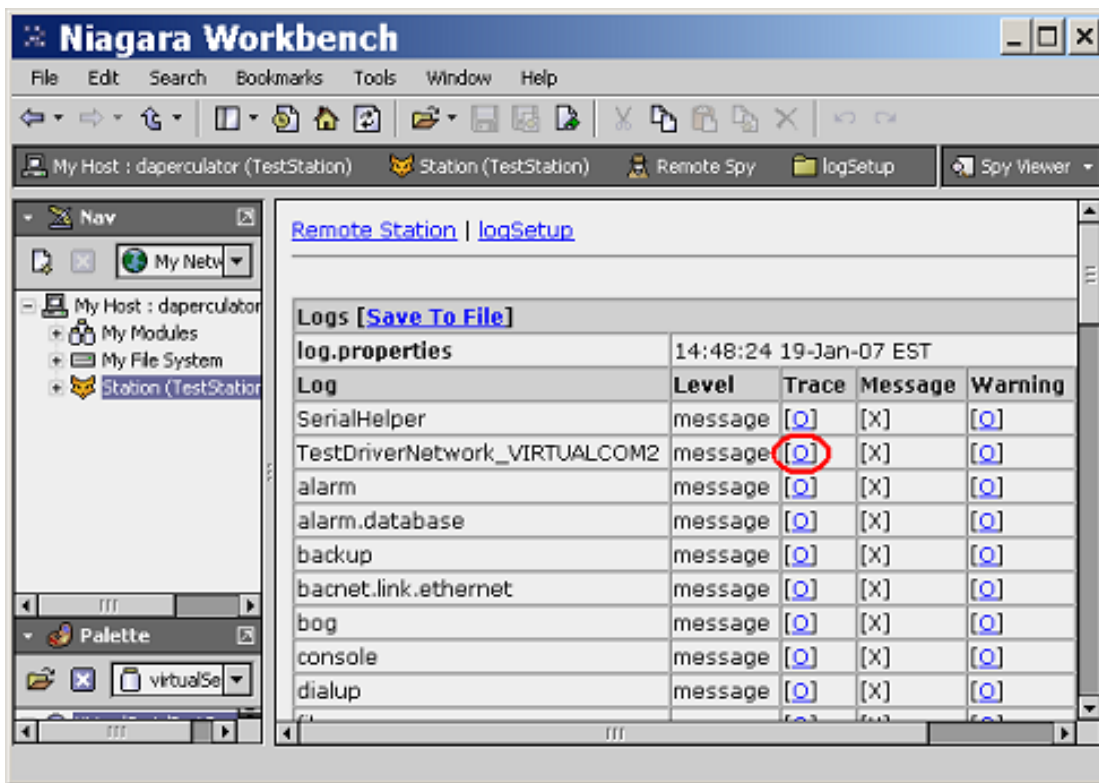
1. Right click the station in the **Nav** tree.
2. Click **Spy**.



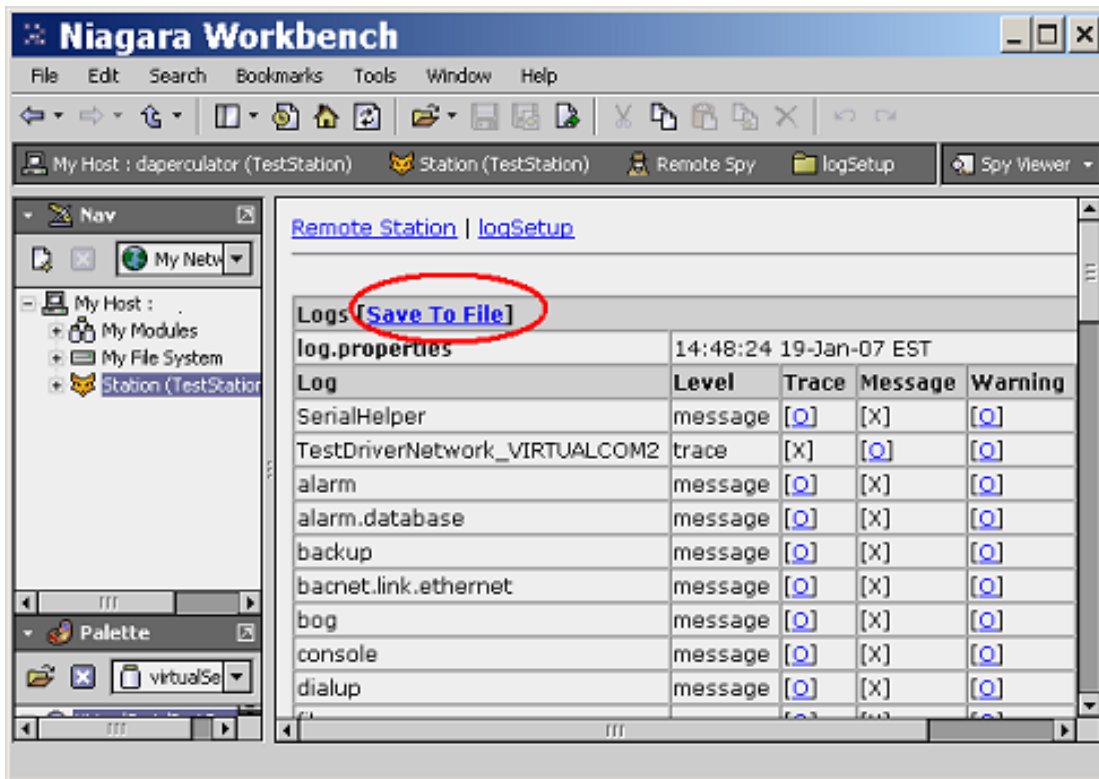
3. Click **logSetup** in the main Workbench view that appears.



4. Notice that there is an entry for your network or device. Activate trace for it by clicking the [O] that is in the **Trace** column.



5. Save the log settings.



6. Then you will see in the station's standard output a summary of all bytes being sent and received.

It might take some testing to verify that your driver is indeed transmitting the correct bytes to your field device and that your field device is indeed responding as expected.

Congratulations! You now have a driver with a network and a device that communicates!

Day 2 - Reading Data Points

During the lesson for day 1, you created a ping request, device id, device, receiver, communicator, network, and ping response. This allowed you to use the developer driver framework to model your driver's device in Niagara AX and allow Niagara AX to automatically **ping** the actual device on the field-bus and then report whether the status of the device is **ok** or **down**. After completing the lesson for day 2, you will have driver control points that Niagara AX will automatically poll (to pull point values into Niagara AX). The process of doing this will be similar to what you followed on day 1.

Definitions:

field-bus

A network that connects equipment together.

field-device

A single unit of external equipment that is connected to a *field-bus*.

driver device

A special component in the Niagara AX framework that virtually represents a *field-device*.

point value

Individual data measurement or setpoint that a *field-device* makes externally available to other field-devices or equipment for read and/or write access using *field-bus* communication.

driver control point

A special component in the Niagara AX framework that virtually represents a *point value*.

frozen property

A special kind of Niagara AX property on a Niagara AX component that (among other benefits) you can easily access from Java source code.

Chapters - Day 2

- [Chapter 9](#)
- [Chapter 10](#)
- [Chapter 11](#)
- [Chapter 12](#)
- [Chapter 13](#)
- [Chapter 14](#)
- [Chapter 15](#)
- [Chapter 16](#)
- [Conclusion](#)

Chapter 9 - Create a Read Request For Your Driver

The developer driver framework will use your driver's read request to automatically read point values from a field-device and introduce them into the rest of the Niagara AX framework. Point values are represented virtually as a control point. Please follow these steps to create a read-request for your driver.

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

1. Review your equipment's protocol documentation again. If you are working directly for the manufacturer of the equipment, then they should be able to provide you with one or more documents that describes the way in which the equipment communicates, plus the structure of the data that the equipment expects to see on the field-bus. If you have purchased this equipment then you will need to negotiate with the equipment's manufacturer in order to gain access to the equipment's protocol.
2. Pick a message from your protocol that retrieves one or more data point values that you are interested in. After reviewing the equipment's protocol documentation, choose a message from the protocol that looks like the simplest message that accomplishes this task.

Some protocols provide several messages that read point values. Some messages read point values as well as details about the point. For now, please only concern yourself with retrieving the actual value of the point, such as a single temperature value or other measurement. Please defer your concerns about retrieving extra information about the data point, such as engineering units (for example - whether it is Fahrenheit or Celsius) until later.

3. Make a Java class that extends **BDdfReadRequest**. Name it **BYourDriverReadRequest**. Create this in the package named **com.yourCompany.yourDriver.comm.req**. Also add an empty slotomatic comment immediately after the opening brace for the class declaration.

To do this, create a text file named `BYourDriverReadRequest.java` in the `jarFileName/src/com/yourCompany/yourDriver/comm/req` folder. Inside the text file, start with the following text:

```
package com.yourCompany.yourDriver.comm.req;

import javax.baja.sys.*;
import javax.baja.io.*;

import com.tridium.ddf.comm.*;
import com.tridium.ddf.comm.req.*;
import com.tridium.ddf.comm.rsp.*;

import com.yourCompany.yourDriver.identify.*;
import com.yourCompany.yourDriver.comm.rsp.*;

public class BYourDriverReadRequest
    extends BDdfReadRequest
{
    /*-
    class BYourDriverReadRequest
    {
    }
    -*/
}
```

4. Override the `toByteArray` method. Build the byte array following your protocol's read message. Inside the body of the `toByteArray` method, you will need to construct a Java byte array and return it. The next step will further describe how to do this.

To add the `toByteArray` method, add the following lines of text after the slotomatic comment of the class:

```
public byte[] toByteArray()
{
}
```

5. Assume that any data you need that is unique to your protocol message for reading data points is the value in a frozen property on another class that extends `BDdfReadParams`. This other class is called the **read parameters** structure. It is very similar to the **device id** structure that was automatically available to the *ping request* during yesterday's lesson. Just like the **ping request** class, the **read request** automatically has access to a copy of the **device id** structure. In addition, the **read request** class automatically also has access to a copy of your driver's **read parameters** structure.

Please do not be overwhelmed at the mention of this. Suffice it to say, you will soon create another class that we will call the **read parameters** structure. On this class, you will define one or more frozen properties that contains the information necessary, other than information about the device itself -- which you already placed in your *device id* structure, to transmit a request to read one or more data values from a field-device.

As mentioned during the lesson for day 1, frozen properties are a special kind of Niagara AX property on a Niagara AX component that you can easily access from Java source code. In subsequent chapters, you will make the class for the **read parameters** structure. For now, please proceed as though you have already created it.

In light of all this discussion, please finish updating the `toByteArray` method to return a byte array that matches the description that your protocol document defines for the message that you chose to be the read request. Please follow this example as a guide:

```
public byte[] toByteArray()
{
    // In the dev driver framework, all requests are automatically
    // Assigned a deviceId when they are created. In addition to the
    // deviceId, read requests are automatically assigned an instance
    // Of a Read Parameters structure when they are created. The dev
    // Driver framework calls the toByteArray method (function) after
    // It creates the read request, therefore this particular request
    // Has already been assigned a device id and read parameters
    // structure. The deviceId will be an instance of BYourDriverDeviceId,
    // The readParameters structure will be an instance of
    // BYourDriverReadParameters.
    // That is how dev driver works! This happens automatically for
    // Your convenience!

    BYourDriverDeviceId deviceId =
        (BYourDriverDeviceId)getDeviceId();

    BYourDriverReadParams readParams =
        (BYourDriverReadParams)getReadParameters();

    final byte SOH = 0x01;
    final byte EOT = 0x04;
    // In this hypothetical example, the protocol document would
    // Indicate that all requests start with a hex 01 byte and
    // All requests end with a hex 04 byte.
    // So, after the hex 01, the protocol expects a number between
    // 0 and 255 to identify the device, followed by some ASCII
    // Characters ("read analog outputs", "read analog inputs".
    // "read digital outputs", or "read digital inputs" in this
    // case), followed by the hex 04 terminator byte.

    // ByteBuffer is a standard Niagara AX utility class in package
    // javax.baja.io
```



```

// Let's use it to help us build the byte array that we will return
ByteBuffer bbuf = new ByteBuffer();
// Writes the hex 01 start character to bbuf, our byte buffer.
bbuf.write(SOH);
// Writes the unit number onto bbuf, our byte buffer.
bbuf.write(deviceId.getUnitNumber());
// Writes the ASCII bytes for the word "read" followed by a space to
// bbuf, our byte buffer.
bbuf.writeBytes("read ");
// Writes the ASCII bytes for the value of the "type string" frozen
// property that will be on our read parameters structure. Writes
// this to bbuf, our byte buffer. For this sample driver, we
// will create on our driver's read parameters structure a frozen
// property called typeString. The value in this property is a
// string. It will either be the string "analog" or "digital".
bbuf.writeBytes(readParams.getTypeString());
// Writes the ASCII byte for a space character. Note, we are
// Following our hypothetical driver's protocol structure.
bbuf.write(' ');
// Writes the ASCII bytes for the value of the "direction" frozen
// Property that will be on our read parameters structure. Writes
// This to bbuf, our buffer of bytes. For this sample driver,
// We will also create on our driver's read parameters structure
// A frozen property called direction. The value in this property
// Is a string. It will either be the string "outputs" or "inputs".
bbuf.writeBytes(readParams.getDirection());
// Writes the byte that according to our hypothetical protocol,
// Indicates the end of the message on the field-bus.
bbuf.write(EOT);
// Converts the byte buffer into an actual Java byte
// Array and returns it.
return bbuf.toByteArray();
}

```

6. Override the `processReceive` method and return a new instance of your read response class (to be created in a subsequent chapter).

To recap part of day 1's lesson, the developer driver framework calls the `toByteArray` method (function), transmits the resulting byte array onto the field-bus, looks for incoming data frames, and passes them to this method (until this method returns a response (not null), throws an exception, or times out).

Please implement the **`processReceive`** method as follows:

```

public BIDdfResponse processReceive(IDdfDataFrame receiveFrame)
    throws DdfResponseException
{
    return new BYourDriverReadResponse(receiveFrame);
}

```

NOTE: When adding methods (functions) please add them to the bottom of your class file (just above the closing squiggly brace). This is to ensure that you do not add your function into the section towards the top of the class that is reserved for the source code that the slotomatic utility automatically generates. Accidentally placing code up there could result in the code being lost when you run the *slot.exe* program.

Chapter 10 - Create a Read Parameters Structure For Your Driver

In the previous chapter, you created a read request. We asked you to construct a byte array and return it from your read request's **toByteArray** method. We asked you to assume that a **read parameters** structure contained any information, other than information about the device itself, that you would need in order to construct a request message from your driver's protocol -- that is, a request message that asks a field-device for one or more data values.

This chapter will guide you through the creation of the **read parameters** structure.

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

1. Review your read request's **toByteArray** method.
2. Make a Java class that extends **BDdfReadParams**. Name it **BYourDriverReadParams**. Create this in a package named **com.yourCompany.yourDriver.identify**. Also add an empty slotomatic comment immediately after the opening squiggly brace for the class declaration.

To do this, create a text file named `BYourDriverReadParams.java` in the `jarFileName/src/com/yourCompany/yourDriver/identify` folder. Inside the text file, start with the following text:

```
package com.yourCompany.yourDriver.identify;

import javax.baja.sys.*;

import com.tridium.ddf.identify.*;

import com.yourCompany.yourDriver.comm.req.*;

public class BYourDriverReadParams
    extends BDdfReadParams
{
    /*-
    class BYourDriverReadParams
    {
    }
    -*/
}
```

3. Add a **properties** section to the slotomatic header and declare properties for any values that you needed in order to make the **toByteArray** method (function) in your read request.

*In our hypothetical example, we required two string properties. In the example **toByteArray** method, we accessed these by calling the **getTypeString** and **getDirection** methods. We designed the code for the **toByteArray** method (function) like this because we planned that when creating the **read parameters** structure, we would add these properties to it. We understood that by naming these two properties **typeString** and **direction** that the Niagara AX slotomatic utility would automatically generate a **getTypeString** method and a **getDirection** method.*

Please make the slotomatic statement on your read parameters structure look something like this:

```

/*-
class BYourDriverReadParams
{
    properties
    {
        typeString : String
        -- This property has nothing to with the dev
        -- driver framework itself. Instead, we need
        -- to construct the toByteArray method of the
        -- driver's read request in following the
        -- driver's protocol to read data values.
        default{["analog"]}
        slotfacets{[MGR_INCLUDE]}

        direction : String
        -- This property has nothing to with the dev
        -- driver framework itself. Instead, we need
        -- to construct the toByteArray method of the
        -- driver's read request in following the
        -- driver's protocol to read data values.
        default{["outputs"]}
        slotfacets{[MGR_INCLUDE]}
    }
}
-*/

```

NOTE: Providing the *slotfacet* of *MGR_INCLUDE* on each of these properties will cause them to automatically appear in your *point manager* for your driver.

4. Override the `getReadRequestType` method and return the `TYPE` of your driver's read request. Please note that we have not yet asked you to run the *slotomatic* utility today, therefore, the java file for your read request class file might not yet have a `TYPE` constant (unless you ran the *slotomatic* utility on your own). The Niagara AX *slotomatic* utility will automatically add the `TYPE` constant to that file, and the java files for the other classes that you will be creating today. Do not worry, we will ask you to run the *slotomatic* utility and the *build* utility at the end of the next chapter.

Add the following code to *BYourDriverReadParams.java*:

```

public Type getReadRequestType(){return BYourDriverReadRequest.TYPE;}

```

Chapter 11 - Create a Read Response For Your Driver

In a previous chapter from today's lesson, you created a read request. We asked you to return an instance of `BYourDriverReadResponse` from your read request's **processReceive** method (function). In this chapter you will learn how to code your read response class.

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

1. Review your equipment's protocol documentation (again), especially where it describes the structure of the field-bus message that will be sent in reply to the read request that you constructed in a previous chapter of today's lesson. If you are working directly for the manufacturer of the equipment, then they should be able to provide you with one or more documents that describes the way in which the equipment communicates, plus the structure of the data that the equipment expects to see on the field-bus. If you have purchased this equipment then you will need to negotiate with the equipment's manufacturer in order to gain access to the equipment's protocol.
2. Determine exactly which data values are returned in the response. If there are more than one then determine exactly how to extract the value for any particular data value in the response.
3. Some protocols provide several messages that read point values. Some messages read point values as well as details about the point. For now, please only concern yourself with retrieving the actual value of the point, such as a single temperature value or other measurement. Please defer your concerns about retrieving extra information about the data point, such as engineering units (for example - whether it is Fahrenheit or Celsius) until later.
4. Make a Java class that extends **BDdfResponse** and implements **BIDdfReadResponse**. Name it **BYourDriverReadResponse**. Create this in a package named **com.yourCompany.yourDriver.comm.rsp**. Also add an empty slotomatic comment immediately after the opening brace for the class declaration.
To do this, create a text file named `BYourDriverReadResponse.java` in the `jarFileName/src/com/yourCompany/yourDriver/comm/rsp` folder. Inside the text file, start with the following text:

```
package com.yourCompany.yourDriver.comm.rsp;

import javax.baja.sys.*;
import javax.baja.status.*;

import com.tridium.ddf.comm.*;
import com.tridium.ddf.comm.rsp.*;
import com.tridium.ddf.comm.req.*;

import com.yourCompany.yourDriver.identify.*;
import com.yourCompany.yourDriver.point.*;

public class BYourDriverReadResponse
    extends BDdfResponse
    implements BIDdfReadResponse
    {
        /*-
         class BYourDriverReadResponse
         {
         }
         -*/
    }
```

5. Add a constructor that takes one parameter that is an `IDdfDataFrame`:

A constructor is a special method (function) that is called when a particular instance of the class is allocated. To put this in perspective, the **processReceive** method that you coded in the previous chapter, will call the constructor on the read response class that you are creating in this chapter. Moreover, the **processReceive** method will pass in the **IDdfDataFrame** that it received from the dev driver framework. An **IDdfDataFrame** is essentially a byte array wrapper. In effect, your read request from the previous chapter will pass the bytes of the response into your read response's constructor.

To add this constructor, add the following lines of text after the slotomatic comment of the class: >

```
public BYourDriverReadResponse(IDdfDataFrame receiveFrame)
{
}
```

You will need to make a copy of the bytes in the given data frame, since it could be a transient part of an internal buffer used in the dev communicator. Declare a byte array on the line that is immediately above the constructor. The byte array will hold a copy of the bytes that are passed to the constructor inside the IDdfDataFrame.

```
private byte[] receiveBytes;
```

Use the following code as the constructor that you already started making:

```
private byte[] receiveBytes; // This is populated by the constructor
private BYourDriverReadParameters readParams; // This is populated by the constructor
/**
 * This constructor is called by the processReceive method of
 * BYourDriverReadRequest.
 *
 * @param a reference to the data frame that matches up with
 * the request that was recently transmitted. The byte array
 * in this frame could be a direct reference to the dev
 * communicator's receiver's internal byte array so this
 * constructor copies the bytes into a safe instance array.
 * @param a reference to the read parameters structure that the
 * read request used to construct its own byte array.
 */
public BYourDriverReadResponse(IDdfDataFrame receiveFrame, BYourDriverReadParams
readParams)
{ // We will use the read parameters reference more for point discovery
  // (to be discussed much later in the tutorial)
  this.readParams = (BYourDriverReadParameters)readParams.newCopy();
  // This copies the bytes of the internal receive buffer
  // Into a copy that is accessible to any other non-static
  // Functions on this class.
  byte[] receiveBuffer = receiveFrame.getFrameBytes();
  int receiveLength = receiveFrame.getFrameSize();
  // Allocates our own byte array to hold the copy
  receiveBytes=new byte[receiveLength];
  // Copies each byte from the receiveBuffer into our own
  // Safe array named receiveBytes.
  for (int i=0; i<receiveLength;i++)
    receiveBytes[i]=receiveBuffer[i];
}
```

6. Also add an empty constructor. An empty constructor might be required with future implementations of the developer driver framework in order to possibly allow client-side proxy copies of the request (if you do not understand what "client-side proxy" means -- *you are not expected to know this* -- then do not despair, please just keep this in the back of your mind and simply add the empty constructor.)

```
/**
 * This empty constructor allows Niagara AX to instantiate
 * a client-side proxy of this request. It is not presently
 * used but could be required in future versions of the
 * developer driver framework.
 */
public BYourDriverReadResponse()
{
}
```

7. Please add a public method (function) named **parseReadValue** that takes one parameter, an **IDdfReadable**, and returns a **BStatusValue**. For now, please make this method return null. We will revisit this in a subsequent chapter of today's

lesson. This method satisfies the **BIDdfReadResponse** interface.

```
public BStatusValue parseReadValue(IDdfReadable readableSource)
{
    return null;
}
```

8. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

Chapter 12 - Create a Proxy Extension For Your Driver

As discussed in the introduction before the first day's lesson, the Niagara AX framework uses something called a **driver control point** to represent a data value that is inside an external unit of equipment. It is now time to take this discussion one step forward.

Niagara AX control points have a frozen property on them named **proxyExt**. Drivers define the external behavior of Niagara AX control points by defining a proxy extension component. The developer driver framework automatically places an instance of your driver's proxy extension onto Niagara AX control point components, as necessary, for your convenience. Therefore, all you need to do is follow this procedure to make this happen automatically:

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

1. Make a Java class that extends **BDdfProxyExt**. Name it **BYourDriverProxyExt**. Create this in a package named **com.yourCompany.yourDriver.point**. Also add an empty slotomatic comment, with an empty **properties** section immediately after the opening brace for the class declaration.

To do this, create a text file named `BYourDriverProxyExt.java` in the `jarFileName/src/com/yourCompany/yourDriver/point` folder. Inside the text file, start with the following text:

```
package com.yourCompany.yourDriver.point;

import javax.baja.sys.*;

import com.tridium.ddf.point.*;
import com.tridium.ddf.identify.*;

import com.yourCompany.yourDriver.identify.*;

public class BYourDriverProxyExt
    extends BDdfProxyExt
{
    /*-
    class BYourDriverProxyExt
    {
        properties
        {
        }
    }
    -*/

    public Type getDeviceExtType()
    { // You will need to return BYourDriverPointDeviceExt
      // After you create it in a subsequent chapter. For
      // Now, please return null to satisfy the compiler.
      return null;
    }
}
```

NOTE: **BDdfProxyExt**, the ancestor of **BYourDriverProxyExt**, has already defined a property called **readParameters** and a property called **pointId**. You need to redefine those two properties and change the default value to be an instance of the corresponding classes in your driver.

In one of the previous chapters of today's lesson, we showed you how to create the **read parameters structure**. We also explained that the **read parameters structure** tells an instance of your driver's *read request* how to construct the byte array that it must return from its **toByteArray** method (function).

Also, in a previous chapter of today's lesson, we showed you how to create the **read response** but had you simply return null from the **parseReadValue** method of your read response. In a subsequent chapter of today's lesson, we will show you how to finish defining the **parseReadValue** method of your read response. It is time that we give you a little more explanation about that.

Please recall that the **parseReadValue** method is passed an instance of **IDdfReadable**. All proxy extensions in the developer driver framework implement this Java interface. In fact, the developer driver framework will make one or more successive calls into your read response's **parseReadValue** method (function). The developer driver framework will call this method on your response one or more times, once for each driver control point that is under the same device from your driver and that shares the equivalent **read parameters structure**. The framework will pass the proxy extension itself, cast as an **IDdfReadable**, into the **parseReadValue** method (function) of your read response.

2. Redefine the **readParameters** property to use your driver's read parameters structure.

*Declare an instance of your driver's read parameters as the default value for the **readParameters** property:*

```
/*-
class BYourDriverProxyExt
{
    properties
    {
        readParameters : BDdfIdParams
        -- This hooks your driver's read parameters structure into the
        -- proxy extension that is placed on control points that are
        -- under devices in your driver. The read parameter's structure
        -- tells the dev driver framework which read request to use to
        -- read the control point. It also tells your read request's
        -- toByteArray method how to construct the bytes for the request.
        default{[new BYourDriverReadParams()]}
        slotfacets{[MGR_INCLUDE]}
    }
}
-*/
```

3. Redefine the **pointId** property to use the point id class from your driver (to be created soon).

Even though you have not yet created it, we will show you how to create the point id in a subsequent chapter of today's lesson.


```

/*-
class BYourDriverProxyExt
{
    properties
    {
        readParameters : BDdfIdParams
        -- This hooks your driver's read parameters structure into the
        -- proxy extension that is placed on control points that are
        -- under devices in your driver. The read parameter's structure
        -- tells the dev driver framework which read request to use to
        -- read the control point. It also tells your read request's
        -- toByteArray method how to construct the bytes for the request.
        default{[new BYourDriverReadParams()]}
        slotfacets{[MGR_INCLUDE]}

        pointId : BDdfIdParams
        -- This tells your read response's parseReadValue method how to
        -- extract the data value for a particular control point.
        default{[new BYourDriverPointId()]}
        slotfacets{[MGR_INCLUDE]}
    }
}
-*/

```

NOTE: Providing the *slotfacet* of *MGR_INCLUDE* on each of these structured properties will cause any of their properties that are flagged with *MGR_INCLUDE* to automatically appear in your *point manager* for your driver.

4. Run slotomatic with the *-mi* switch and resolve any slotomatic compiler errors. Do not perform an actual build on your driver yet (this class is not yet ready to be fully compiled).
5. A couple more classes are required before you will be able to successfully perform a full build on your driver. One of these is the point id, which we mentioned briefly in this chapter. The other is the **point device extension**, which is very simple to create. Next, we will show you how to create these both. After that, we will have you finish creating your proxy extension.

Chapter 13 - Tell Your Read Response How to Read the Value For Your Driver's Proxy Ext

In the previous chapters of today's lesson, we showed you how to create your driver's read response class. We also helped you start creating your driver's proxy extension class. Please recall from the chapter where you created the read response that we asked you to make the **parseReadValue** method (function) return null. Please also recall that in the previous chapter we mentioned that the **IDdfReadable** that is passed to the **parseReadValue** method (function) of your read response will be an instance of your driver's proxy extension. With all of this being explained, it is now time to finish writing the **parseReadValue** method of your driver's **read response**.

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

Please Modify BYourDriverReadResponse.parseReadValue As Follows:

1. Verify that the given **IDdfReadable** is an instance of **BYourDriverProxyExt**. If not then simply return null.
2. Cast the given **IDdfReadable** to **BYourDriverProxyExt**.
3. Get a reference to the **pointId** from the given proxy extension and cast it to **BYourDriverPointId**.
*Do not worry that you have not yet created the **BYourDriverPointId** class. We will show you how to create it in the next chapter.*
4. Plan that any information you would need to determine the current value of the given proxy extension is available from your driver's point id (such as an offset into the response frame's byte array or some information that will allow you to determine said offset).

Just as you created your driver's **read parameters** structure for the purpose of helping you to create the **toByteArray** method in your read request, you will likewise create your driver's **point id** for the purpose of helping you to retrieve a particular control point's value from the corresponding read response.

5. Perform whatever computation and lookup is necessary, from the byte array copy that you made in the constructor of this class, in order to retrieve the correct, current value for the control point that the given proxy extension belongs to. Use the information that you will encode in your **point id** to determine this.
Please use this hypothetical example as a guide:

```
// The following imports are for our particular implementation
// Of the parseReadValue(IDdfReadable) method. Different
// Drivers might need to import these or other classes to
// Custom-define the parseReadValue(IDdfReadable) method.
import java.util.*;

import javax.xml.baja.control.*;
```

```

// We will use this array to optimize our implementation of the
// parseReadValue method.
private String[] rawValues = null;
public BStatusValue parseReadValue(IDdfReadable readableSource)
{ // Verify that the given readableSource is an instance of your
  // driver's proxy extension
  if (readableSource instanceof BYourDriverProxyExt)
  { // Casts the given readableSource into a BYourDriverProxyExt
    BYourDriverProxyExt proxy
      = (BYourDriverProxyExt)readableSource;
    // Gets the point id of the given proxy
    BYourDriverPointId pointId
      = (BYourDriverPointId)proxy.getPointId();
    // In our hypothetical protocol, the read request asks to read
    // either analog or digital outputs or inputs. The field device
    // Has up to 30 analog or digital outputs or inputs. The read
    // Response returns all of the corresponding values that were
    // Requested inside a comma delimited string. The substrings in
    // between the commas are signed integers for analog values or
    // The text "on" or "off" for digital values.
    if (rawValues==null) // Parses each of the values from the
      parseRawValues(); // receive frame into a string array.
    // Calls the getReadValue method that takes an int as a
    // Param. We created this method for our own convenience. The
    // getReadValue is private and not part of the developer driver
    // framework. It exists only to help us keep this method short.
    return getReadValue(proxy,pointId);
  }
  else
  {
    return null;
  }
}

```

```

/**
 * This method is called by the parseReadValue method. It is not a
 * requirement for the dev driver framework. Rather, we decided
 * that we needed this method in order to support our own logic
 * in the parseReadValue method.
 */
private void parseRawValues()
{
  // The response frame contains a hex 02 character, followed by
  // The device's unit number as one byte, followed by an ASCII
  // Comma delimited string, followed by a hex 04 character.

  // Makes a string from the bytes that were in the received frame
  // Starting at location 2 (the third byte since Java arrays
  // Start at index 0). The new string will be parsed starting at
  // Location 2 of the receiveBytes and including as many bytes as
  // The length of the receiveBytes array minus 3 bytes -- those
  // Three bytes being skipped are the hex 02 char, the unit nbr,
  // And the hex 04 terminating char.
  String commaDelimitedString
    = new String(receiveBytes, 2, receiveBytes.length-3);

  // Prepares to count the number of substrings in commaDelimitedString
  int numValues = 0;
  // StringTokenizer is a standard Java utility class that makes it
  // Loops through a string by tokens. Our string tokenizer
  // Will split up the commaDelimitedString based on comma chars.
  StringTokenizer t = new StringTokenizer(commaDelimitedString,",");
  // Loops through the comma delimited string and counts the number
  // Of substrings that are in between commas
  while (t.hasMoreTokens())
  {
    numValues = numValues + 1;
    // Consumes the token to allow us to loop to the next one
    t.nextToken();
  }
}

```

```

    }
    // Allocates an array to hold each individual raw value
    rawValues = new String[numValues];
    // Uses the following int to index into rawValues in the next loop
    int offset = 0;
    // Loops again through all of the substrings that are between commas
    // In the comma delimited string
    t = new StringTokenizer(commaDelimitedString, ",");
    while (t.hasMoreTokens())
    {
        rawValues[offset]=t.nextToken();
        offset = offset+1;
    }
    // NOTE: Now we can retrieve the string for any particular value by
    // Directly indexing into the rawValues array.
}

```

```

/**
 * We call this method ourselves from the parseReadValue method.
 * It is not required by the developer driver framework, instead, we
 * decided to create this method in order to shorten the
 * parseReadValue method.
 *
 * This method looks up the string value for the given offset and
 * returns it as a BStatusValue that is appropriate for the given
 * proxy extension's driver control point.
 *
 * @param a reference to the BYourDriverProxyExt to parse the read
 * value for.
 *
 * @param a reference to the BYourDriverPointId to use to tell us
 * how to parse the read value from the response bytes.
 *
 * @return a BStatusNumeric, BStatusBoolean, BStatusEnum, or
 * BStatusString that appropriately matches the proxy's control
 * point type. If the proxy's control point is a BNumericWritable
 * or BNumericPoint then this will return a BStatusNumeric that
 * represents the present value of the point. If the proxy's
 * control point is a BBooleanWritable or a BBooleanPoint then
 * this returns a BStatusBoolean that represents the current value
 * of the point. If the proxy's control point is a BEnumWritable
 * or BEnumPoint then this returns BStatusEnum that represents the
 * current value of the point. If this proxy's control point is a
 * BStringPoint or BStringWritable then this returns a BStatusString
 * that represents the current value of the point. Sorry for the
 * verbose description.
 */
private BStatusValue getReadValue(BYourDriverProxyExt proxy,
                                  BYourDriverPointId pointId)
{
    // Gets the raw value at the index for the given proxy
    // NOTE: When we create our pointId, we will give it an int property named offset
    String sRawValue = rawValues[pointId.getOffset()];
    // Normalizes the string raw value into an int
    int iRawValue = 0;
    if (sRawValue.equalsIgnoreCase("on"))
        iRawValue = 1;
    else if (sRawValue.equalsIgnoreCase("off"))
        iRawValue = 0;
    else
        iRawValue = Integer.parseInt(sRawValue);

    // Wraps iRawValue into a BStatusValue of the appropriate type
    // For the control point
    BControlPoint controlPoint = proxy.getParentPoint();
    // Checks if the control point is a Numeric Writable or Numeric
    // Point component.
    if (controlPoint instanceof BNumeric)
    { // This is all we need to do! Any scales, offsets, or other

```

```

    // Conversions are handled by other parts of Niagara. We just
    // Need to return a raw representation here.
    return new BStatusNumeric(iRawValue);
}
// Checks if the control point is a Boolean Writable or a Boolean
// Point component. NOTE: We must check boolean before enum because
// a BIBoolean is also a BIEnum!
else if (controlPoint instanceof BIBoolean)
{ // This is all we need to do. Any polarity conversion is
  // specified and handled elsewhere in Niagara AX
  return new BStatusBoolean(iRawValue>0);
}
// Checks if the control point is an Enum Writable or an Enum Point
// component.
else if (controlPoint instanceof BIEnum)
{ // A slight extra step is imperative for Enum Points to preserve
  // any dynamic enum range (text to ordinal mapping).
  BStatusEnum e = ((BEnumPoint)controlPoint).getOut();
  return new BStatusEnum( BDynamicEnum.make(
    (int)Math.round(iRawValue),
    e.getValue().getRange()));
}
else if (controlPoint instanceof BStringPoint)
{
  return new BStatusString(sRawValue);
}
else
{
  throw new IllegalStateException(
    "Unsupported control point type: "+
    controlPoint.getType()+"! Please have my program fixed.");
}
}

```

6. Proceed to the next chapter without performing a build on the response. The response will not successfully compile until you define your driver's point id (**BYourDriverPointId**). The next chapter explains how to do this.

Chapter 14 - Create Your Point Id and Point Folder Class

In the previous chapter of today's lesson you coded the **parseReadValue** method of your read response. In the **parseReadValue** method, you identified whatever information that you needed in order to retrieve the value for any particular point from the bytes of the response's data frame. Now that you have identified this information, creating the point id should be straight-forward!

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

Create Your Point Id Structure For Your Driver

1. Make a Java class that extends **BDdfIdParams**. Name it **BYourDriverPointId**. Create this in a package named **com.yourCompany.yourDriver.identify**. Also add an empty slotomatic comment, with an empty **properties** section immediately after the opening brace for the class declaration. *To do this, create a text file named **BYourDriverPointId.java** in the **jarFileName/src/com/yourCompany/yourDriver/identify** folder. Inside the text file, start with the following text:*

```
package com.yourCompany.yourDriver.identify;

import javax.baja.sys.*;

import com.tridium.ddf.identify.*;

public class BYourDriverPointId
    extends BDdfIdParams
{
    /*-
    class BYourDriverPointId
    {
        properties
        {
        }
    }
    -*/
}
```

2. In the **properties** section of the slotomatic header, declare properties for any information that you needed in order to complete the **parseReadValue** method (function) in your read response (the previous chapter covers this lesson). *In our hypothetical example, we required an integer offset between 0 and 29. In the example **parseReadValue** method, we accessed this by calling the **getOffset** method. We coded the **parseReadValue** method like this because we knew that in this chapter, we would add a property named **offset** to our hypothetical **point id** structure. We knew that by naming this property **offset** that the Niagara AX slotomatic utility would automatically generate a **getOffset** method (function).*

Please make the slotomatic statement on your point id structure look something like this (instead of adding a property named *offset* add one or more properties named appropriately to accommodate your version of the *parseReadValue* method in the previous chapter):

```

/*-
class BYourDriverPointId
{
    properties
    {
        offset : int
        -- This property has nothing to do with the dev
        -- driver framework itself. Instead, we need to
        -- know the location of a point's value when in
        -- the parseReadValue method of yourDriver's
        -- read response
        default{[0]}
        slotfacets{[MGR_INCLUDE]}
    }
}
-*/

```

NOTE: Providing the *slotfacet* of *MGR_INCLUDE* on each of these properties will cause them to automatically appear in your *point manager* for your driver. }

3. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

Now that your driver's point id has been defined, the rest of the classes that you created in today's lesson should now be ready to compile.

Create a Point Folder Component For Your Driver

1. Create a class named **BYourDriverPointFolder** in package **com.yourCompany.yourDriver.point**. To do this, create a text file named *BYourDriverPointFolder.java* in the *jarFileName/src/com/yourCompany/yourDriver/point* folder. Inside the text file, start with the following text:

```

package com.yourCompany.yourDriver.point;

import com.tridium.ddf.*;
import javax.baja.sys.*;

public class BYourDriverPointFolder
    extends BDdfPointFolder
{
    /*-
    class BYourDriverPointFolder
    {
        properties
        {
        }
    }
    -*/
}

```

2. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

NOTE: Like the device folder, the point folder is practically a formality in the Niagara AX framework.

Chapter 15 - Create Your Driver's Point Device Extension Component

In the Niagara AX framework driver control points reside under a special component that called a **point device extension**. The developer driver framework takes care of most of the details for you. All you need to do is define a class for your driver's **point device extension**, add one of your driver's **point device extension** components as a property to BYourDriverDevice, and override a method on BYourDriverProxyExt. From that method you will need to simply return the TYPE constant that belongs to your driver's **point device extension**.

This might sound complicated but it is very simple to do and will only take a matter of minutes. In fact, we believe this will be the simplest chapter in today's lesson!

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

1. Make a Java class that extends **BDdfPointDeviceExt**. Name it **BYourDriverPointDeviceExt**. Create this in a package named **com.yourCompany.yourDriver**. Also add an empty slotomatic comment, with an empty **properties** section immediately after the opening brace for the class declaration. To do this, create a text file named **BYourDriverPointDeviceExt.java** in the *jarFileName/src/com/yourCompany/yourDriver* folder. Inside the text file, start with the following text:

```
package com.yourCompany.yourDriver;

import javax.baja.sys.*;
import javax.baja.util.*;

import com.tridium.ddf.*;

import com.yourCompany.yourDriver.point.*;

public class BYourDriverPointDeviceExt
    extends BDdfPointDeviceExt
{
    /*-
    class BYourDriverPointDeviceExt
    {
        properties
        {
        }
    }
    -*/
}
```

2. Define the follow methods on *BYourDriverPointDeviceExt*:


```

/**
 * Associates BYourDriverPointDeviceExt to BYourDriverDevice.
 */
public Type getDeviceType()
{
    return BYourDriverDevice.TYPE;
}
/**
 * Associates BYourDriverPointDeviceExt to BYourDriverPointFolder.
 */
public Type getPointFolderType()
{
    return BYourDriverPointFolder.TYPE;
}
/**
 * Associates BYourDriverPointDeviceExt to BYourDriverProxyExt
 */
public Type getProxyExtType()
{
    return BYourDriverProxyExt.TYPE;
}
/**
 * This can be left null, depending on how you decide to define
 * the discovery behavior in your driver. We will visit the
 * discovery process in further detail during another day's
 * lesson.
 */
public BFolder getDiscoveryFolder()
{
    return null;
}

```

3. Open the *BYourDriverProxyExt.java* file that you created in the previous chapters of today's lesson.
4. Implement the *getDeviceExtType* method as follows:

```

/**
 * This associates BYourDriverDeviceExt with
 * BYourDriverProxyExt within the Niagara AX
 * framework.
 */
public Type getDeviceExtType()
{
    return BYourDriverPointDeviceExt.TYPE;
}

```

5. Add the following import statement to the top of *BYourDriverProxyExt.java* (along with the other import statements)

```
import com.yourCompany.yourDriver.*;
```

6. Open the **BYourDriverDevice.java** file that you created in the previous day's lesson.
7. Add a property named **points** whose type is **BYourDriverPointDeviceExt**

```
/*-
class BYourDriverDevice
{
  properties
  {
    deviceId : BDdfIdParams
      -- This plugs in an instance of yourDriver's
      -- device id as this device's deviceId
      default {[new BYourDriverDeviceId()]}
    points : BYourDriverPointDeviceExt
      -- Adds the special point device extension
      -- component to the property sheet and the
      -- Niagara AX navigation tree. This special
      -- component will contain and process all
      -- control points for YourDriver
      default{[new BYourDriverPointDeviceExt()]}
  }
}
-*/
```

8. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

Chapter 16 - Testing Your Progress of Day 2

At this point, you are ready to *see* and *test* what you have done for today's lesson.

Run Your Station and Review it From Workbench

1. Open a **Niagara Console**. You typically do this by clicking the Windows Start menu, hovering over programs, then hovering over Niagara 3.2.8 (or whatever version of Niagara AX you are using), and selecting *Console*.
2. In the **Niagara Console** type `station <NewStation>` but replace `<NewStation>` with the name that you assigned to your station in the lesson for Day 1. Press enter after you type this text.

```
d:\Niagara-3.2.x> station TestStation
MESSAGE [12:27:06 22-Mar-07 EDT][sys.registry] Up-to-date [140ms]
MESSAGE [12:27:07 22-Mar-07 EDT][sys] Baja runtime booted ("d:\niagara\niagara-3.2.8")
MESSAGE [12:27:07 22-Mar-07 EDT][sys.registry] Loaded [390ms]
MESSAGE [12:27:08 22-Mar-07 EDT][sys] Loading "d:\niagara\niagara-3.2.8\stations
\TestStation\config.bog"...
MESSAGE [12:27:12 22-Mar-07 EDT][sys] Loaded (4166ms)
MESSAGE [12:27:16 22-Mar-07 EDT][alarm.database] Created
MESSAGE [12:27:16 22-Mar-07 EDT][sys] Services Initialized (210ms)
MESSAGE [12:27:16 22-Mar-07 EDT][sys.mixin] Updated [0ms]
WARNING [12:27:16 22-Mar-07 EDT][platform] Local daemon session not available, station
not started by niagarad
MESSAGE [12:27:16 22-Mar-07 EDT][web.server] HTTP Server started on port 80
MESSAGE [12:27:16 22-Mar-07 EDT][fox] Service started on port 1911
MESSAGE [12:27:16 22-Mar-07 EDT][sys] nre.version: 3.2.8
MESSAGE [12:27:16 22-Mar-07 EDT][sys] *** Station Started (832ms) [11938ms total] ***
niagara>
```

3. Your station program should begin executing. Your station is now online (again)!
4. Run Niagara AX Workbench.
5. Use Niagara AX Workbench to connect to the Niagara AX station that is running in your console window.

As a reminder, here's how to connect to the station from Workbench AX:

- o Click File on the main menu
- o Click Open
- o Click Open Station (fox).
- o A window will appear with the title Open Station.
- o Leave the Host i.p. address blank (it will then resolve to the local host)
- o For **User** enter **admin** (all lower case)
- o For **Password** either leave empty or enter the administrator password that you entered back when you created the station.
- o By default this is likely to be user name admin with no password.
- o Now your station has been opened. Your workstation now appears in the navigation tree with an icon next to it that looks like a desktop PC.

NOTE: For illustrations of this procedure, please review the conclusion for Day 1's lesson.

Visit Your New Point Manager

1. Locate your workstation in the **Nav** tree. Expand your workstation by clicking the plus sign that is next to it in the **Nav** tree.
2. Click the plus sign next to the *station* (the station will likely have an icon of a fox next to it) that is underneath

your workstation.

3. Click the plus sign next to the **Config** component that is under the station. Watch it expand.
4. Click the plus sign next to the **Drivers** component that is under the *Config* component. Watch it expand.
5. Click the plus-sign that is next to the network component that you created in day 1's lesson. Watch the network expand.
6. Click the plus-sign that is next to the device component that you created in day 1's lesson. Watch the device expand.
7. Double-click the special **points** folder under your device. Its title will be *Your Driver Point Manager* where *Your Driver* is the name of your driver module.
8. The **Dev Point Manager** is displayed in right-most two-thirds or so of the Workbench.
9. Notice that there are columns in the **Dev Point Manager** for each property on your *read parameters structure* and *point id* on which you added the *MGR_INCLUDE* facet.

Create A Sample Control Point

1. Click the **New** button to add a control point using the **Dev Point Manager**,
2. Choose a control point type, such as Numeric Point, in the window that pops up.
3. Click ok, and then ok again.
4. Notice that the *driver control point* appears in the **Dev Point Manager** database list.
5. Whenever you view this *driver control point*, such as right now, the station program will attempt to communicate to the field-device by instantiating an instance of your *read request*, calling your *toByteArray* method, sending out the resulting byte array, receiving data frames, matching up the data frames to your *read request*, instantiating a *read response*, and updating the *driver control point* with the value returned by the *parseReadValue* method of your driver's *read response*.
6. Also notice that the driver control point appears in the Niagara AX navigation tree.
7. Double-click the control point itself from within the Niagara AX navigation tree.
8. Notice that you now see the property sheet of your new control point.
9. Click the plus-sign that is next to the property named **proxyExt**.
10. You should now *see* your *read parameters* structure and your *point id*!

Verify That Your Driver is Trying to Perform Communication to Read Your Control Point's Value

1. Right click the station icon in the Navigation tree.
2. Click **Spy**. The main Workbench area changes to become the **Spy** page.
3. Click **logSetup** in the **Spy** page located in the main Workbench area. The main Workbench area displays the **Logs** table.
4. Locate a row for your network or device's communicator and click the circle with a line under it [O] that is in the **Trace** column of the table.
5. Click the **Save to File** link that is located near the top of the **Log** table.
6. Restore your station's Niagara AX console (DOS window) (this is a separate windows application that you have running on your workstation).
7. Verify that the station is printing the bytes to the Niagara AX console (DOS) window that it is trying to send and also those that it is receiving.
8. Repeat any of the steps from today's lesson, if you desire, to test other data points in your driver.

At this point, if you have not already connected your equipment to your Workstation, we encourage you to do so. If your equipment is serial then you should follow your equipment's wiring and set-up instructions to hook your Workstation's serial port onto your equipment's field-bus.

If your equipment communicates wirelessly, such as over a mesh network, you will likely need to connect a serial-to-wireless radio to the serial port. Please consult the documentation for your equipment (or consult the equipment manufacturer directly) if you do not know how to connect your PC to the wireless network. In summary though, you will usually plug a serial-to-wireless radio into your PC's serial port.

If your equipment communicates over Tcp/Ip or Udp/Ip then connect it to the same local area network (LAN) as your PC. Please also make sure that you follow the manufacturer's instructions for configuring the Udp/Ip or Tcp/Ip settings of your equipment. This is necessary so that your equipment can truly communicate over your LAN. As a first step towards troubleshooting the connection you can open a DOS console and attempt to ping your equipment's i.p. address. However, please keep in mind that not all equipment supports the standard ping mechanism used by the DOS console's ping utility. So, pinging your equipment from DOS is just one suggestion that may or may not help.

Another way to troubleshoot a Tcp/Ip or Udp/Ip connection is to place your personal computer and your Tcp/Ip or Udp/Ip equipment onto an Ethernet hub and possibly assigning one or both a static i.p. address. This can help eliminate many of the variables involved in troubleshooting Tcp/Ip connectivity issues. You can also ask your equipment manufacturer if they provide any software that can connect to the equipment. If so, you can install the manufacturer's software on your PC. If your manufacturer's software truly uses Tcp/Ip or Udp/Ip and if you can connect to the equipment using their software on your PC then your Dev Tcp or Dev Udp driver (running in a station on your PC) should be able to connect to the same equipment as well.

Nevertheless, troubleshooting Tcp/Ip or Udp/Ip connections can be rather challenging due to the many possible configurations of a local area network (LAN). For that reason, any further discussion about troubleshooting Tcp/Ip or Udp/Ip connections is beyond the scope of this tutorial.

WARNING: Depending on how well documented and straight-forward your equipment's protocol is, it could take days or weeks for you to test all possible data points. We hope that the developer driver framework alleviates your concerns about the inner workings of Niagara and allows you, instead, to stay focused on your equipment and on your equipment's communication protocol.

Tutorial Day 2 Conclusion

Congratulations! You now have a Niagara AX driver with a network component. You also have a device component that pings and has a device extension component. This allows you to create any driver control point components that you desire and use them to introduce data values from your field-device to the rest of the Niagara AX framework. With these data values, you may now take advantage of all that Niagara AX has to offer, such as easy web page creation, alarming, and data logging.

After today's lesson, you should have a general idea about how to virtually represent and access the data points in your field-device within the Niagara AX framework. In tomorrow's lesson, we will show you how to add support to your driver to allow you to change data values that are in your field-device from the Niagara AX framework. To do this, you will add *write* functionality to your driver's control points. That will allow you to control and schedule your data values from Niagara AX.

Day 3 - Writing Data Points

During the lesson for day 2, you programmed your driver to read control points for your equipment. You created a **read request**, a **read parameters** structure, a **read response**, a **proxy extension**, a **point id**, and a **point device extension**. You made some associations amongst all of these and at the end of the day you had control points with read access to your equipment's data values.

The process for adding write capabilities to your driver's control points is very similar. In fact, you may even find it to be somewhat simpler, since you will be re-using some of the components that you created in day 2's lesson. You will make a **Write Request** and a **Write Parameters** structure. You might also add some more properties to your **point id**. You will then make similar associations with these new items to your proxy extension. After today's lesson, your control points should be capable of reading and writing data values in your field-device.

Chapters - Day 3

- [Chapter 17](#)
- [Chapter 18](#)
- [Chapter 19](#)
- [Chapter 20](#)
- [Chapter 21](#)
- [Conclusion](#)

Chapter 17 - Create a Write Request For Your Driver

The developer driver framework will use your driver's **write request** to automatically update point values in your equipment when the value in the corresponding control point component changes from within Niagara AX. Please follow these steps to create a write-request for your driver.

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

1. Further review your equipment's protocol documentation. If you are working directly for the manufacturer of the equipment, then they should be able to provide you with one or more documents that describes the way in which the equipment communicates, plus the structure of the data that the equipment expects to see on the field-bus. If you have purchased this equipment then you will need to negotiate with the equipment's manufacturer in order to gain access to the equipment's protocol.
2. Pick a message from your protocol that updates one or more of the data point values that you are interested in, preferably one or more of the data point values that you retrieved in day 2's lesson. After reviewing the equipment's protocol documentation, choose a message from the protocol that looks like the simplest message that accomplishes this task.
Some protocols are designed so that one write message can update more than one data point in the equipment. Other protocols are designed with a one-to-one relationship between write messages and point values. The developer driver framework accommodates both of these scenarios.

3. Make a Java class that extends **BDdfWriteRequest**. Name it **BYourDriverWriteRequest**. Create this in a package named **com.yourCompany.yourDriver.comm.req**. Also add an empty slotomatic comment immediately after the opening brace for the class declaration.

To do this, create a text file named `BYourDriverWriteRequest.java` in the `jarFileName/src/com/yourCompany/yourDriver/comm/req` folder. Inside the text file, start with the following text:

```
package com.yourCompany.yourDriver.comm.req;

import com.tridium.ddf.comm.req.*;
import javax.baja.sys.*;

public class BYourDriverWriteRequest
    extends BDdfWriteRequest
{
    /*-
    class BYourDriverWriteRequest
    {
    }
    -*/
}
```

4. Override the `toByteArray` method. Build the byte array following your protocol's write message. Inside the body of the `toByteArray` method, you will need to construct a Java byte array and return it. The next step will further describe how to do this.

To add the `toByteArray` method, add the following lines of text after the slotomatic comment of the class:

```
public byte[] toByteArray()
{
}
```

5. Assume that any data you need in order to construct the write message itself, ignoring any details about

which particular data value to write, is a frozen property on a given **write parameters** structure. Just as in previous lessons, you will create this structure later in the day's lesson. This will help you determine what information needs to be included in the **write parameters** structure. Suffice it to say, you will soon create another component that we will call the **write parameters**. On this component, you will define one or more frozen properties that uniquely identifies a particular write request message on your equipment's field-bus.

As mentioned during the lesson for day 1, frozen properties are a special kind of Niagara AX property on a Niagara AX component that you can easily access from Java source code. In subsequent chapters, you will make the class for the **write parameters** structure. For now, please assume that you have already created it.

Please recall that to create frozen properties on a component, you add a special comment just after the class statement in the Java file. After doing that, you will run a Niagara AX development utility called **slotomatic** that will parse the special comment and add some Java code to your file -- the Java code necessary to add the property to the Niagara AX component that the Java file defines. This automatically generated Java code includes a method (function) called `getMyProperty` (where `myProperty` is the name of the frozen property, as you would have declared in the special comment).

In light of all this discussion, please finish updating the `toByteArray` method to return a byte array that matches the description that your protocol document defined for the message that you chose to be the write request. Please follow this example as a guide:

```
public byte[] toByteArray()
{
    // In the dev driver framework, all requests are automatically
    // Assigned a deviceId when they are created. In addition to the
    // deviceId, write requests are automatically assigned an instance
    // Of a Write Parameters structure when they are
    // Created. The dev driver framework calls the toByteArray method
    // (function) after it creates the write request, therefore this
    // Particular request has already been assigned a device id and
    // a Write parameters structure. The deviceId will
    // Be an instance of BYourDriverDeviceId, the writeParameters
    // Structure will be an instance of BYourDriverWriteParameters,
    // That is how dev driver works! This happens automatically.

    BYourDriverDeviceId deviceId =
        (BYourDriverDeviceId)getDeviceId();

    BYourDriverWriteParams writeParams =
        (BYourDriverWriteParams)getWriteParameters();

    final byte SOH = 0x01;
    final byte EOT = 0x04;
    // In this hypothetical example, the protocol document
    // Indicates that all requests start with a hex 01 byte and
    // All requests end with a hex 04 byte.
    // After the hex 01, the protocol expects a number between
    // 0 and 255 to identify the device, followed by the ASCII
    // Characters "Write " or "Force Write" followed by one or more of the
    // following:
    //   ao{X}={signed int}
    //   do{X}=On
    //   do{X}=Off
    // Where:
    //   {X} = a sequence of ASCII digits '0'-'9' to identify which
    //         analog or digital output to change.
    //
    //   {signed int} = a sequence of ASCII digits '0'-'9' preceded
    //                  optionally by a minus sign
    //
    // If more than one of these sequences are present then they are
    // delimited by commas.
    // The message ends with a hex 04 terminator byte.
```

```

// ByteArrayOutputStream is a standard Java class in package java.io
// Let's use it to help us build the byte array that we will return
ByteArrayOutputStream bos = new ByteArrayOutputStream();
// Writes the hex 01 start character to bos, our Java stream of
// bytes.
bos.write(SOH);
// Possibly writes the ASCII bytes for "force" followed by a space to
// bos, our Java stream of bytes
if (writeParams.getForceWrite())
{
    bos.write('f'); bos.write('o'); bos.write('r'); bos.write('c'); bos.write('e');
    bos.write(' ');
}
// Writes the ASCII bytes for "write" followed by a space to
// bos, our Java stream of bytes
bos.write('w'); bos.write('r'); bos.write('i'); bos.write('t'); bos.write('e');
bos.write(' ');

// Loops through all control points that are to be updated by this request.
// The dev driver framework will try to coalesce all control points
// Under a device that have equivalent write parameters into a single write
// Request. You can get the array of these items by calling the method
// named getWritableSource. It returns an array of IDdfWritable -- these
// will be the proxy extensions of your driver's control points.
IDdfWritable pointsToUpdate[] = getWritableSource();
for (int i=0; i<pointsToUpdate.length;i++)
{ // This is a good thing to check in case dev driver adds support for
  // Writing components that are not proxy extensions
  if (pointsToUpdate[i] instanceof BYourDriverProxyExt)
  { // Casts the IDdfWritable to BYourDriverProxyExt
    BYourDriverProxyExt updateProxy = (BYourDriverProxyExt)pointsToUpdate[i];
    // Gets the read parameters structure, it helped BYourDriverReadRequest
    // know whether to read analog or digital. We can make optimal use of this
    // property by re-using it here also.
    BYourDriverReadParameters proxyReadParams =
        (BYourDriverReadParameters)updateProxy.getReadParameters();
    // Gets the point id structure for the particular point. We previously
    // defined it to contain an offset. We can make optimal use of this offset
    // by re-using it here also.
    BYourDriverPointId proxyPointId =
        (BYourDriverPointId)updateProxy.getPointId();
    // This boolean helps our logic remember whether the point is digital
    boolean digitalProxy=false;
    // The getRawValue method is available for your convenience to convert any
    // IDdfWritable among the writable source into a double precision value.
    double proxyValue = getRawValue(updateProxy);
    if (proxyReadParams.getTypeString().equalsIgnoreCase("analog"))
    { // If updating an analog output on the field-device
      bos.write('a'); bos.write('o');
      digitalProxy=false;
    }
    else if (proxyReadParams.getTypeString().equalsIgnoreCase("digital"))
    { // Else, if updating a digital output on the field-device
      bos.write('d'); bos.write('o');
      digitalProxy=true;
    }
    else // Sanity check
      throw new RuntimeException("Oops! Writing type string '"+
        proxyReadParams.getTypeString()+"' is not supported.");
    // Writes the point index as a string
    bos.write(Integer.toString(proxyId.getOffset()).getBytes());
    bos.write('=');
    // Writes the new value
    if (digitalProxy) // If the point is digital
    {
      if (proxyValue>0) // If the rawValue is greater than 0
      {
        bos.write('O'); bos.write('n'); // Writes "On"
      }
      else // Else, rawValue must be 0
      {
        // Writes "Off"
      }
    }
  }
}

```

```

        bos.write('0'); bos.write('f'); out.write('f');
    }
    // Else, the point is analog
    else // The Java Long.toString method outputs a sequence of ASCII digits
        bos.write(Long.toString(Math.round(proxyValue))) // Possibly preceded by '-'

    if(i+1<pointsToUpdate.length) // If not the last point in the loop
        bos.write(','); // Then this writes a comma
    } // End of if instanceof etc.
} // End of for loop
// Writes the byte that according to our hypothetical protocol,
// Indicates the end of the message on the field-bus.
bos.write(EOT);
// Converts the stream of bytes in our Java stream of bytes into
// An actual Java byte array and returns it.
return bos.toByteArray();
}

```

6. Override the processReceive method and return a new instance of your write response class (to be created in a subsequent chapter).

To recap part of day 1's lesson, the developer driver framework calls the toByteArray method (function), transmits the resulting byte array onto the field-bus, looks for incoming data frames, and passes them to this method (until this method returns a response (not null), throws an exception, or times out).

Please implement the **processReceive** using the following Java code as a guide:

```

public BIDdfResponse processReceive(IDdfDataFrame recieveFrame)
    throws DdfResponseException
{
    // We coded our hypothetical receiver such that received frames always have
    // at least 2 bytes. So lets not worry about checking for that
    String responseStatus = new String( // Constructs a string from bytes
        receiveFrame.getDataBytes(),    // In the receive frame buffer
        2,                               // Starting at index 2, taking
        receiveFrame.getLength()-3);     // All chars except SOH, unitNbr, and EOT
    // According to our hypothetical protocol, the device responds 'OK' if the
    // Write succeeds. Any other string denies the write. In the event of a
    // Denial, the string itself describes the denial in plain English.
    if (responseStatus.equalsIgnoreCase("OK"))
        return new BYourDriverWriteResponse();
    else
        throw new DevResponseException("Equipment Nak - "+responseStatus);
}

```

Chapter 18 - Create a Write Parameters Structure For Your Driver

The developer driver framework tries its best to automatically coalesce all control points under that same device, that need written, and that have equivalent write parameters into a single write request. In a previous chapter of today's lesson, you created your write request. We asked you to assume that you had already created the write request and added any data that you needed to construct the write request itself, not including the information needed to specify the values of individual points.

Please create your **write parameters** structure following these steps as a guide:

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

1. Review your write request's `toByteArray` method.
2. Make a Java class that extends **BDdfIdParams** and implements **BIDdfWriteParams**. Name it **BYourDriverWriteParams**. Create this in a package named **com.yourCompany.yourDriver.identify**. Also add an empty slotomatic comment immediately after the opening brace for the class declaration.

To do this, create a text file named `BYourDriverWriteParams.java` in the `jarFileName/src/com/yourCompany/yourDriver/identify` folder. Inside the text file, start with the following text:

```
package com.yourCompany.yourDriver.identify;

import com.tridium.ddf.identify.*;
import javax.baja.sys.*;

public class BYourDriverWriteParams
    extends BDdfIdParams
    implements BIDdfWriteParams
{
    /*-
    class BYourDriverWriteParams
    {
    }
    -*/
}
```

3. Add a **properties** section to the slotomatic header and declare properties for any values that you needed in order to make the `toByteArray` method (function) in your write request.

*In our hypothetical example, we required one boolean property named "forceWrite". In the example `toByteArray` method, we accessed this by calling the **getForceWrite** method. We coded the **toByteArray** like this because we knew that in this chapter, we would add this property to our hypothetical **write parameters** structure. We knew that by naming the properties **forceWrite** that the Niagara AX slotomatic utility would automatically generate a **getForceWrite** method.*

Please make the slotomatic statement on your read parameters structure look something like this:

```

/*-
class BYourDriverWriteParams
{
    properties
    {
        forceWrite : boolean
        -- This property has nothing to with the dev
        -- driver framework itself. Instead, we need
        -- to construct the toByteArray method of the
        -- driver's write request in following the
        -- driver's protocol to write data values.
        -- In this hypothetical protocol, if we do not
        -- forceWrite then the equipment's internal
        -- program could overwrite any change that
        -- Niagara might make to a data value.
        default{[true]}
    }
}
-*/

```

4. Override the `getWriteRequestType` method and return the TYPE of your driver's write request. The Niagara AX slotomatic utility automatically adds the TYPE constant to your Java file when you run the utility.

Add the following code to *BYourDriverWriteParams.java*:

```

public Type getWriteRequestType(){return BYourDriverWriteRequest.TYPE;}

```

5. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

Redefine the writeParameters property on BYourDriverProxyExt

1. Open *BYourDriverProxyExt.java*.
2. Redefine the 'writeParameters' property by modifying the slotomatic header of *BYourDriverProxyExt.java* as follows:

```

class BYourDriverProxyExt
{
    properties
    {
        readParameters : BDdfIdParams
        -- This hooks your driver's read parameters structure into the
        -- proxy extension that is placed on control points that are
        -- under devices in your driver. The read parameter's structure
        -- tells the dev driver framework which read request to use to
        -- read the control point. It also tells your read request's
        -- toByteArray method how to construct the bytes for the request.
        default{[new BYourDriverReadParams()]}
        slotfacets{[MGR_INCLUDE]}

        pointId : BDdfIdParams
        -- This tells your read response's parseReadValue method how to
        -- extract the data value for a particular control point.
        default{[new BYourDriverPointId()]}
        slotfacets{[MGR_INCLUDE]}
    }
}

```

```
writeParameters : BDdfIdParams
-- This hooks your driver's write parameters structure into the
-- proxy extension that is placed on control points that are
-- under devices in your driver. The write parameter's structure
-- tells the dev driver framework which write request to use to
-- write the control point. It also tells your write request's
-- toByteArray method how to construct the bytes for the request.
default{[new BYourDriverWriteParams()]}
slotfacets{[MGR_INCLUDE]}

}
}
-*/
```

NOTE: Providing the *slotfacet* of *MGR_INCLUDE* on each of these structured properties will cause any of their properties that are flagged with *MGR_INCLUDE* to automatically appear in your *point manager* for your driver.

3. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

Chapter 19 - Create a Write Response For Your Driver

The **write response** is typically a very simple class to develop. Most driver protocols return back a small amount of data, just enough to indicate whether the field-device accepted the write request. Many drivers can get away with essentially having an empty implementation for the **write response**.

Please note that in our examples for the previous chapters in today's lesson, we coded the **processReceive** method of our hypothetical **write request** to throw a **DdfResponseException** if the field-device denied the driver's request to write data values. If the field device accepted the driver's request, then we return an empty instance of **BYourDriverWriteResponse**.

Create your driver's **write response** as follows:

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

1. Make a Java class that extends **BDdfResponse**. Name it **BYourDriverWriteResponse**. Create this in a package named **com.yourCompany.yourDriver.comm.rsp**. Also add an empty slotomatic comment immediately after the opening brace for the class declaration.

To do this, create a text file named `BYourDriverWriteResponse.java` in the `jarFileName/src/com/yourCompany/yourDriver/comm/rsp` folder. Inside the text file, start with the following text:

```
package com.yourCompany.yourDriver.comm.rsp;

import com.tridium.ddf.comm.rsp.*;
import javax.baja.sys.*;

public class BYourDriverWriteResponse
    extends BDdfResponse
{
    /*-
    class BYourDriverWriteResponse
    {
    }
    -*/
}
```

NOTE: If you do not define any constructors at all then Java automatically creates an empty constructor for you. By not defining any constructors for **BYourDriverWriteResponse** in this example, we implicitly create an empty constructor with full, public access.

2. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

Chapter 20 - Update Your Point Id

Please recall that in a previous chapter in today's lesson you created your driver's **write request**. In that chapter, you coded up a method named **toByteArray** on your **write request**. We asked you to assume that any information that you needed to construct the byte array, without even specifying any information about any particular data values, was available on your **write parameters** structure. Obviously, however, you also needed to specify information about the particular data value to update. In our hypothetical example, the **point id** already contained a property called *offset* that we added during the lesson for day 2. In our hypothetical example, that was all we needed from the **point id**. However, this will not necessarily be the case for all drivers.

If you required more information in order to create the byte array that you returned from the **toByteArray** method of your request, then please add frozen properties for that information to the **BYourDriverPointId** component that you created on day 2.

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

Please follow this easy procedure:

1. Open the `BYourDriverPointId.java` file that you created on day 2.
2. Add any additional properties that you need to the **properties** section of the slotomatic header.
3. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

Chapter 21 - Create an Auto Request and Response For Your Driver

You only need to worry about this chapter if your driver features the notion of **overriding** data points. Most equipment runs an internal program that updates some or all of the field-device's data values. Some protocols require a point to be placed into a special mode if and when it is controlled by an entity other than the field-device itself. In our hypothetical protocol such a mode takes effect by transmitting the ASCII string "force" in the byte array of the write request. Field-devices that have a notion of **overriding** points will usually need to be specially informed if and when the particular data value is no longer being controlled by the external entity. The dev driver framework makes this easy for you to implement in your driver.

To do this, you will create a **point-auto** request and possibly a **point-auto** response. Please follow these steps if this is necessary for your driver:

While following the examples in this chapter, please replace the text *jarFileName*, *yourDriver* and *yourCompany* as previously described in the [Preparation](#) section):

1. Make another Java class that extends **BDdfWriteRequest**. Name it **BYourDriverPointAutoRequest**. Create this in the package named **com.yourCompany.yourDriver.comm.req**. Also add an empty slotomatic comment immediately after the opening brace for the class declaration.

To do this, create a text file named `BYourDriverPointAutoRequest.java` in the `jarFileName/src/com/yourCompany/yourDriver/comm/rsp` folder. Inside the text file, start with the following text:

```
package com.yourCompany.yourDriver.comm.req;

import com.tridium.ddf.comm.req.*;
import javax.baja.sys.*;

public class BYourDriverPointAutoRequest
    extends BDdfWriteRequest
{
    /*-
    class BYourDriverPointAutoRequest
    {
    }
    -*/
}
```

2. Override the `toByteArray` method. Build the byte array following your protocol's message that un-forces the data point. Please review your driver's protocol once again, if necessary, to find such a message. If you cannot find such a message then you can probably skip this chapter!

Inside the body of the `toByteArray` method, you will need to construct a Java byte array and return it. The next step will further describe how to do this.

Assume that any data you need in order to construct the auto (unforce) message itself, ignoring any details about which particular data value to auto (unforce), is a frozen property on the **write parameters** structure (the same structure that your driver's **write request** uses to construct the byte array that it returns from its **toByteArray** method). If you need more information than your **write parameters** presently provides, then you will need to update your **write parameters** structure and add frozen properties for the required information.

In light of this discussion, please code the `toByteArray` method to return a byte array that matches the description that your protocol document defines for the message that you identify as the *un-force* or *relinquish-auto* request. Please follow this example as a guide:

```

public byte[] toByteArray()
{
    // In the dev driver framework, all requests are automatically
    // Assigned a deviceId when they are created. In addition to the
    // deviceId, write requests are automatically assigned an instance
    // Of a Write Parameters structure when they are created, this point
    // auto request is an instance of BDdfWriteRequet. The dev driver
    // Framework calls the toByteArray method (function) after it
    // Creates the write request, therefore this particular request has
    // already been assigned a device id, and a Write parameters
    // structure. The deviceId will be an instance of BYourDriverDeviceId,
    // the writeParameters structure will be an instance of
    // BYourDriverWriteParameters,
    // That is how dev driver works! This happens automatically.

    BYourDriverDeviceId deviceId =
        (BYourDriverDeviceId)getDeviceId();

    BYourDriverWriteParams writeParams =
        (BYourDriverWriteParams)getWriteParameters();

    final byte SOH = 0x01;
    final byte EOT = 0x04;
    // In this hypothetical example, the protocol document
    // Indicates that all requests start with a hex 01 byte and
    // All requests end with a hex 04 byte.
    // After the hex 01, the protocol expects a number between
    // 0 and 255 to identify the device, followed by the ASCII
    // Characters "Unforce ", followed by one or more of the
    // following:
    //   ao{X}
    //   do{X}
    // Where:
    //   {X} = a sequence of ASCII digits '0'-'9' to identify which
    //         analog or digital output to change.
    //
    // If more than one of these sequences are present then they are
    // delimited by commas.
    // The message ends with a hex 04 terminator byte.

    // ByteArrayOutputStream is a standard Java class in package java.io
    // Let's use it to help us build the byte array that we will return
    ByteArrayOutputStream bos = new ByteArrayOutputStream();
    // Writes the hex 01 start character to bos, our Java stream of
    // bytes.
    bos.write(SOH);
    // Writes the ASCII bytes for "unforce" followed by a space to
    // bos, our Java stream of bytes
    bos.write('u'); bos.write('n');
    bos.write('f'); bos.write('o'); bos.write('r'); bos.write('c'); bos.write('e');
    bos.write(' ');

    // Loops through all control points that are to be auto 'ed (unforced) by this request.
    // The dev driver framework will try to coalesce all control points that need updated
    // Under a device that have equivalent write parameters into a single write
    // Request. You can get the array of these items by calling the method
    // named getWritableSource. It returns an array of IDdfWritable -- these
    // will be the proxy extensions of your driver's control points.
    IDdfWritable pointsToUpdate[] = getWritableSource();
    for (int i=0; i<pointsToUpdate.length;i++)
    { // This is a good thing to check in case dev driver adds support for
      // Auto'ing components that are not proxy extensions
      if (pointsToUpdate[i] instanceof BYourDriverProxyExt)
      { // Casts the IDdfWritable to BYourDriverProxyExt
        BYourDriverProxyExt updateProxy = (BYourDriverProxyExt)pointsToUpdate[i];
        // Gets the read parameters structure, it helped BYourDriverReadRequest
        // know whether to read analog or digital. We can make optimal use of this
        // property by re-using it here also.
        BYourDriverReadParameters proxyReadParams =
            (BYourDriverReadParameters)updateProxy.getReadParameters();
        // Gets the point id structure for the particular point. We previously
        // defined it to contain an offset. We can make optimal use of this offset
        // by re-using it here also.
        BYourDriverPointId proxyPointId =

```

```

        (BYourDriverPointId)updateProxy.getPointId();
        if (proxyReadParams.getTypeString().equalsIgnoreCase("analog"))
        { // If updating an analog output on the field-device
            bos.write('a'); bos.write('o');
        }
        else if (proxyReadParams.getTypeString().equalsIgnoreCase("digital"))
        { // Else, if updating a digital output on the field-device
            bos.write('d'); bos.write('o');
        }
        else // Sanity check
            throw new RuntimeException("Oops! Auto'ing type string '"+
                proxyReadParams.getTypeString()+"' is not supported.");
        // Writes the point index as a string
        bos.write(Integer.toString(proxyId.getOffset()).getBytes());

        if(i+1<pointsToUpdate.length) // If not the last point in the loop
            bos.write(','); // Then this writes a comma
        } // End of if instanceof etc.
    } // End of for loop
    // Writes the byte that according to our hypothetical protocol,
    // Indicates the end of the message on the field-bus.
    bos.write(EOT);
    // Converts the stream of bytes in our Java stream of bytes into
    // An actual Java byte array and returns it.
    return bos.toByteArray();
}

```

3. Override the `processReceive` method and return a new instance of your auto response class (to be created in a subsequent step). You can re-use your **write response** here, if you determine that it provides the same exact functionality that you would need for your **point-auto response**. This will most likely be the case if you left your **write response** essentially as an empty class that extends **BDdfWriteRequest**.

To recap part of day 1's lesson, the dev driver framework calls the `toByteArray` method (function), transmits the resulting byte array onto the field-bus, looks for incoming data frames, and passes them to this method (until this method returns a response (not null), throws an exception, or times out).

Please implement the **processReceive** using the following Java code as a guide. You may notice that this hypothetical example is nearly identical to the **processReceive** method that was in the example for the **read request**!

```

public BIDdfResponse processReceive(IDdfDataFrame recieveFrame)
    throws DdfResponseException
{ // We coded our hypothetical receiver such that received frames always have
  // at least 2 bytes. So lets not worry about checking for that
  String responseStatus = new String( // Constructs a string from bytes
      receiveFrame.getDataBytes(), // In the receive frame buffer
      2, // Starting at index 2, taking
      receiveFrame.getLength()-3); // All chars except SOH, unitNbr, and EOT
  // According to our hypothetical protocol, the device responds 'OK' if the
  // Request succeeds. Any other string denies the write. In the event of a
  // Denial, the string itself describes the denial in plain English.
  if (responseStatus.equalsIgnoreCase("OK"))
      return new BYourDriverWriteResponse(); // <-In this case our auto
                                              // response has the
                                              // equivalent functionality
                                              // as BYourDriverWriteResponse
                                              // So we can re-use it.
  else
      throw new DevResponseException("Equipment Nak During Auto - "+responseStatus);
}

```

4. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).
5. Create your point-auto response, if you were not able to re-use your **write response**.
*As a guide, please follow the instructions in the earlier chapter of today's lesson where we showed you how to create your driver's **write response**.*
6. Associate your auto request to your **write parameters** class.

Do this by changing the class declaration on *BYourDriverWriteParams* to specify that it also implements the ***BIDdfAutoParams*** interface. Then define a method called ***getDevAutoRequestType*** on ***BYourDriverWriteRequest***

Follow this example as a guide:

```
package com.yourCompany.yourDriver.identify;

import com.tridium.ddf.identify.*;
import javax.baja.sys.*;

public class BYourDriverWriteParams
    extends BDdfIdParams
    implements BIDdfWriteParams, BIDdfAutoParams
{
    /*-
    class BYourDriverWriteParams
    {
        properties
        {
            forceWrite : boolean
            -- This property has nothing to with the dev
            -- driver framework itself. Instead, we need
            -- to construct the toByteArray method of the
            -- driver's write request in following the
            -- driver's protocol to write data values.
            -- In this hypothetical protocol, if we do not
            -- forceWrite then the equipment's internal
            -- program could overwrite any change that
            -- Niagara might make to a data value.
            default{[true]}
        }
    }
    -*/
    public Type getWriteRequestType(){return BYourDriverWriteRequest.TYPE;}
    public Type getAutoRequestType(){return BYourDriverPointAutoRequest.TYPE;}
}
```

7. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

Chapter 22 - Testing Your Progress of Day 3

In today's lesson you created a **write request**, a **write response**, and a **write parameters** structure. You may have also created a **point-auto request** and a **point-auto response**. You may have also added some frozen properties to your **point id**. By doing all of this, you have supplied enough information for the developer driver framework to automatically update the data values in your field-device if and when a corresponding control point component (with a proxy extension from your driver) is controlled or scheduled from within Niagara AX.

We encourage you to follow the same procedure that we specified at the conclusion of yesterday's lesson to test the control points in your driver. To test writing your driver's control points, you may right-click any writable control point (for example, a Numeric Writable) under your device's **point device extension** from the workbench (provided that it is connected to your test station), expand the **actions** menu, choose **emergency override**, enter a value in the window that pops up, and click **Ok**. To then **auto** your control point, right click the point again but choose **Emergency Auto** from the **actions** menu.

Writable Driver Control Points - Overview

Driver control points that are writable update data values in a field-device. Writable control points feature an input priority scheme. They feature 16 control inputs. If you recall back to the discussion before day 1, we introduced you to the concept of **logic links**. Niagara AX installation professionals can link to 14 of the control point's inputs (inputs 2-7 and inputs 9-16). They can design logic that feeds any or no value through any links that they create to any of the available priority inputs.

Inputs 1 and 8, as well as the fallback value, are reserved for the following items that appear on the control point's **actions** menu.

Emergency Override

Sets the control value for the most critical priority -- priority level 1.

Emergency Auto

Clears the control value for the most critical priority -- priority level 1.

Override

Sets the intermediate control value -- priority level 8.

Auto

Clears the intermediate control value -- priority level 8.

Set

Sets the fallback value. This value takes effect if no value is being supplied to any of inputs 1-16.

Driver control points automatically use your driver to set the data value in the field-device equal to the input value that is provided at the priority of the index closest to input one. If no values are provided to any of the sixteen input properties then the driver control point will set the data value equal to the fallback value (if one is specified) on the property sheet of the control point.

If no fallback value is provided and if no input values are provided, then the driver control point will not modify the particular data value in your field-device. Whenever a transition occurs within the station that causes this scenario to occur, the driver control point will ask your driver to **auto** the data value in the field-device. Your driver may or may not need to take any special action during this scenario. If you specified a **point auto request** then the dev driver framework will automatically transmit it through your field-bus in this scenario.

We hope that this discussion will allow you to test the *write* and *auto* functionality that you created during today's lesson.

NOTE: You will need to follow the procedure outline in the conclusion for Day 2. However, make sure that you add a writable control point from your driver's point manager. Writable control points are Numeric Writable,

Boolean Writable, Enum Writable, and String Writable. We recommend that you start testing with a Numeric Writable and/or a Boolean Writable.

After adding a writable, driver control point to your station, right-click the writable control point, hover over the **actions** side-menu, choose the **Emergency Override** action, and specify a value on the dialog that appears. The developer driver framework and the Niagara AX framework will work together to transmit the byte array from an instance of your driver's write request, as described during today's lesson.

After running tests to write a value to one your driver's control points, please consider running tests to *auto* your driver's control point. To do this, right-click the writable control point, hover over the **actions** side-menu, and choose the **Emergency Auto** action. The developer driver framework and the Niagara AX framework will work together to transmit the byte array from an instance of your driver's *auto* request, as described during today's lesson.

Please refer to the conclusion of Day 2 for illustrations about how to use your driver's point manager to create driver control points.

Tutorial Day 3 Conclusion

Congratulations! Your driver is now more than half-way finished. In the lessons to follow we will focus on a feature of the Niagara AX framework and of the dev driver framework called auto-discovery.

In the lessons so far, you added a **network** component to your test station. Then you added a **device** component to your **network** and some **control point** components to your **device**.

To add your device, you double-clicked the **network** component. Then you clicked **New** from the **Ddf Device Manager**. After that, you visited the property sheet for your device and specified the proper values for the **deviceId** (possibly also the Tcp/Ip address and port -- if applicable).

To add your control points, you double-clicked the **point device extension** component under your **device**. Then you clicked **New** from the **Ddf Point Manager**. After that, you specified the proper values for the **readParameters**, **writeParameters**, and **pointId** structures on your control points.

This required that you, yourself, know the exact information necessary to identify the **device** and the **control points**.

Wouldn't it be much better if your station could automatically query your field-bus and find these for you? It usually can! In tomorrow's lesson, we will show you what to do to your driver to make this happen.

Day 4 - Add Device Discovery Capabilities to Your Driver

Discovery is a special procedure that a Niagara AX installation technician follows to add driver **device** components and **control point** components to his or her Niagara AX station. When configuring stations to use your driver, Niagara AX installation technicians will double-click the **drivers** folder below the **config** folder in the **station**. Then the technicians will click the **New** button and choose your driver's **network** from the drop down list box on the window that appears.

After that, technicians will double click your network to visit your driver's device manager. By using the developer driver framework, you can automatically provide a fully-functional device manager when the technician double-clicks your **network**. After that, technicians *prefer* to click the **Discover** button from your driver's **Device Manager**. After clicking the **Discover** button, the station performs special communication on your field-bus and figures out which devices are online. Then Niagara updates the top half of the **Device Manager** with a list of available devices. Technicians will then select one or more devices from the top half and click the **Add** button. That adds one of your driver's **device** components for each of the selected items.

If you follow the steps in today's lesson and update your driver accordingly, then the developer driver framework on which your driver is built, will automatically perform all details for the **device discovery** process.

In the previous day's lessons, you programmed your driver to ping a field-device in order to determine whether it is online. You also programmed your driver to read and to write data values in your equipment using Niagara AX control points.

To ping your field-device, you created a **device** component, a **device id** structure, and a **ping request**. In Java code you made some special associations between these. That allowed the developer driver framework to do the rest.

Similarly, to read and write data values from your field-device, you created a **proxy extension** component, a **read parameters** structure, a **write parameters** structure, a **point id** structure, a **read request**, a **read response**, a **write request**, a **write response**, and possibly a **point-auto request** and a **point-auto response**. In your Java code, you made some special associations between these. That allowed the developer driver framework to do the rest.

The process of adding support to your driver for auto-discovery of devices will be rather similar.

Chapters - Day 4

- [Chapter 23](#)
- [Chapter 24](#)
- [Chapter 25](#)
- [Chapter 26](#)
- [Conclusion](#)

Chapter 23 - Create Your Device Discovery Parameters

NOTE: This class is analogous to your driver's read parameters and write parameters classes.

Some protocols, especially those with field-gateways, have special messages that a driver can transmit to request all devices or maybe just a group of devices, depending on the size-limitation of the particular messages's response. For these types of drivers the developer driver framework will attempt to loop through all possible combinations of the driver's special request that retrieves information about the existence of multiple devices.

Other protocols, especially serial master-slave protocols, do not feature a message that can request information about whether multiple devices are online. For these types of drivers, you will need to take some special steps that will cause the developer driver framework to attempt to **ping** all possible combinations of the **device id**.

Review your driver's protocol. Try to find documentation about a message that your driver can transmit in order to receive information about the existence of more than one device.

A. If your driver's protocol features a message that can retrieve information about multiple field-devices:

1. Determine what information the request's byte array will need.
2. Create a class named **BYourDriverDeviceDiscoverParams** that extends **BDdfDiscoveryParams** in the package **com.yourDriver.identify**.

To do this, create a text file named `BYourDriverDeviceDiscoverParams.java` in the `jarFileName/src/com/yourCompany/yourDriver/identify` folder. Inside the text file, start with the following text:

```
package com.yourCompany.yourDriver.comm.identify;

import com.tridium.ddf.identify.*;
import javax.baja.sys.*;

public class BYourDriverDeviceDiscoverParams
    extends BDdfDiscoverParams
{
    /*-
    class BYourDriverDeviceDiscoverParams
    {
        properties
        {
        }
    }
    -*/
}
```

Based on your driver's protocol document, please be prepared to add properties to the slotomatic statement to help you construct the byte array in the device discovery request (you will create the device discovery request in the next chapter)

3. Define the **getFirst**, **getLast**, **getNext**, and **isAfter** methods. These are essential and allow the developer driver auto discovery process to automatically loop through all possible combinations of your discovery parameters.

getFirst

The **getFirst** method should return an instance of `BYourDriverDeviceDiscoverParams` that represents the data that would be placed in the protocol request to request information about the first device or series of devices. If your protocol has no way of identifying a definite *first* device then you will need to be creative here and return an instance of `BYourDriverDeviceDiscoverParams` with all properties set to special values, that you will determine, such as `Integer.MIN_VALUE` for int properties.

getLast

The **getLast** method should return an instance of `BYourDriverDeviceDiscoverParams` that represents the data in the protocol request that would request information about the last device or series of devices. This might not be possible to accurately define for some protocols. If your protocol does not have a way of identifying the last series of devices, then you will need to be creative here and return an instance of `BYourDriverDeviceDiscoverParams` with all properties set to special values, that you will create, such as `Integer.MAX_VALUE` for int properties.

getNext

The **getNext** method should return an instance of `BYourDriverDeviceDiscoverParams` that represents the data in the protocol request that would request the next device or series of devices, with respect to the particular instance of `BYourDriverDeviceDiscoverParams`. This method should review the values of some or all of the properties that you add to `BYourDriverDeviceDiscoverParams`. This method should return a new instance of `BYourDriverDeviceDiscoverParams` whose property values are incremented in such a way that the new instance would represent the data needed in the *toByteArray* method of the device discovery request in order to ask the field-gateway for information about the next device or set of devices.

isAfter

The **isAfter** method will be passed an instance of `BYourDriverDeviceDiscoverParams`. You should review the property values of the given instance of `BYourDriverDeviceDiscoverParams` and return true if the current instance would request information about a device or series of devices that would be *after* the device or series of devices that the given instance of `BYourDriverDeviceDiscoverParams` would identify. If the current instance represents a device or series of devices that is before or equal to the given instance, then this method should return false.

We understand that this can be a bit tricky, since the definitions of *first*, *last*, *next*, and *after* can be somewhat abstract in some protocols. Please feel free to skip ahead and revisit this later if you wish.

4. Your device discover params class also needs to define the `getDiscoverRequestType` method and return the Niagara-AX **TYPE** for your device discovery request class (you will create this in the next chapter).

```

/**
 * The implementing class needs to return a Type that represents
 * the discover request from the driver whose discoverId can be an
 * instance of this object's class. If the class supports more than
 * one discover request type, then this should return the type that is
 * the most appropriate for the particular instance of the implementing
 * class.
 *
 * This is fundamental to dev driver's auto-discovery features.
 *
 * @return review method description.
 */
public Type getDiscoverRequestType()
{
    return null; // For now! Soon we will revisit this and return
                // BYourDriverDeviceDiscoveryRequest.TYPE after
                // Creating it.
}

```

5. Update your driver's device id structure, make it implement BIDdfDiscoveryLeaf. Implement the following methods that BIDdfDiscoveryLeaf requires.

NOTE: BIDdfDiscoveryLeaf objects appear in the "discovered list" (top half) of the device manager.

Your driver's device id also needs to import javax.baja.registry.*, com.tridium.ddf.discover.*, and import com.yourCompany.yourDriver.*;

```

package com.yourCompany.yourDriver;

import javax.baja.sys.*;
import javax.baja.registry.*;

import com.tridium.ddf.identify.*;
import com.tridium.ddf.comm.req.*;
import com.tridium.ddf.discover.*;

import com.yourCompany.yourDriver.*;

public class BYourDriverDeviceId
    extends BDdfDeviceId
    implements BIDdfDiscoveryLeaf
{
    ...

    ...

    ...

    /**
     * When a control point is added to the station from the Dev
     * Point Manager, it is given this name by default (possibly
     * with a suffix to make it unique).
     * @return
     */
    public String getDiscoveryName()
    {
        return "NewDevice";
    }

    /**
     * Descendants need to return an array of TypeInfo objects corresponding
     * to all valid Niagara Ax types for this discovery object. This is
     * important when the end-user clicks 'Add' from the user interface for
     * the manager.
     */
}

```

```

    * For this discovery object, please return a list of the types
    * which may be used to model it as a BComponent in the station
    * database. If the discovery object represents a device in your
    * driver then method should return an array with size of
    * at least one, filled with TypeInfo's that wrap the Niagara AX
    * TYPE's for your driver's device components.
    *
    * The type at index 0 in the array should be the type which
    * provides the best mapping. Please return an empty array if the
    * discovery cannot be mapped.
    */
    public TypeInfo[] getValidDatabaseTypes()
    {
        return new TypeInfo[]{BYourDriverDevice.TYPE.getTypeInfo()};
    }
}

```

6. Revisit the BYourDriverDeviceDiscoverParams from earlier in this chapter. Define the getDiscoveryLeafType method and return the Niagara-AX TYPE of your device id class.

```

/**
 * The implementing class needs to return a Type that will be the
 * discovery leaves in the ddf manager. For the device discovery
 * process, the discovery leaf TYPE is the device id TYPE.
 *
 * @return BYourDriverDeviceId.TYPE
 */
public Type getDiscoveryLeafType()
{
    return BYourDriverDeviceId.TYPE
}

```

B. If your driver is a serial driver, uses a master-slave protocol, and provides no message that can retrieve information about multiple field-devices:

You should modify your device id, ping params, ping request, and ping response to also serve in the auto-discovery procedure.

1. Open the **BYourDriverDeviceId.java** file from Day 1 of this tutorial.
2. Modify the class declaration statement and declare that BYourDriverDeviceId implements BIDdfDiscoverParams and BIDdfDiscoveryLeaf.

```

public class BYourDriverDeviceId
    extends BDdfIdParams
    implements BIDdfDiscoverParams, BIDdfDiscoveryLeaf
{
    ...
}

```

3. Your driver's device id structure, acting also as the device discover params structure, needs to satisfy the BIDdfDiscoverParams interface and define the **getFirst**, **getLast**, **getNext**, and **isAfter** methods in order to allow the developer driver auto discovery process to automatically loop through all possible combinations of your discovery parameters.

For example, in the hypothetical protocol that we used in the days past, our device had a unitNbr. Let's assume that the unitNbr could range from 0 to 50. The **getFirst** method would return an instance of BYourDriverDeviceId with a unitNbr of 0. The **getLast** method would return an instance of BYourDriverDeviceId with a unitNbr of 50. The **getNext** method would return an instance of BYourDriverDeviceId with a unitNbr that is one more than the current instances's unitNbr: *this.getUnitNumber()+1*. Etc...

Your driver's device id also needs to satisfy the BIDdfDiscoveryLeaf interface and implement the **getDiscoveryName** and **getDatabaseTypes** methods.

Your driver's device id also needs to import javax.baja.registry.*; com.tridium.ddf.discover.*; import com.yourCompany.yourDriver.*; and com.yourCompany.yourDriver.comm.req.*;

```
package com.yourCompany.yourDriver;

import javax.baja.sys.*;
import javax.baja.registry.*;

import com.tridium.ddf.identify.*;
import com.tridium.ddf.comm.req.*;
import com.tridium.ddf.discover.*;

import com.yourCompany.yourDriver.*;
import com.yourCompany.yourDriver.comm.req.*;

public class BYourDriverDeviceId
    extends BDdfDeviceId
    implements BIDdfDiscoverParams, BIDdfDiscoveryLeaf
{
    /*-
    class BYourDriverDeviceId
    {
        properties
        {
            unitNumber : int
                -- This is the unitNumber in our hypothetical protocol.
                default{[0]}
        }
    }
    -*/

    private static final int FIRST_UNIT_NBR = 0;
    private static final int LAST_UNIT_NBR = 50;

    /**
     * Niagara AX requires a public, empty constructor, so that it can perform
     * Its own introspection operations.
     */
    public BYourDriverDeviceId(){}
    /**
     * This constructor is for our own convenience in the methods getFirst,
     * getNext, etc.
     */
    public BYourDriverDeviceId(int unit)
    {
        setUnitNumber(unit);
    }
    public BIDdfDiscoverParams getFirst()
    {
```

```

        return new BYourDriverDeviceId(FIRST_UNIT_NBR);
    }

    public BIDDfDiscoverParams getLast()
    {
        return new BYourDriverDeviceId(LAST_UNIT_NBR);
    }

    public BIDDfDiscoverParams getNext()
    {
        int nextUnitNumber = getUnitNumber()+1;
        if (nextUnitNumber>LAST_UNIT_NBR) // Circles back to first
            return getFirst();
        else
            return new BYourDriverDeviceId(nextUnitNumber);
    }

    public boolean isAfter(BIDDfDiscoverParams anotherId)
    {
        return this.getUnitNumber() >
            ((BYourDriverDeviceId)anotherId).getUnitNumber();
    }

    /**
     * When a control point is added to the station from the Dev
     * Point Manager, it is given this name by default (possibly
     * with a suffix to make it unique).
     * @return
     */
    public String getDiscoveryName()
    {
        return "NewDevice";
    }

    /**
     * Descendants need to return an array of TypeInfo objects corresponding
     * to all valid Niagara Ax types for this discovery object. This is
     * important when the end-user clicks 'Add' from the user interface for
     * the manager.
     *
     * For this discovery object, please return a list of the types
     * which may be used to model it as a BComponent in the station
     * database. If the discovery object represents a device in your
     * driver then method should return an array with size of
     * at least one, filled with TypeInfo's that wrap the Niagara AX
     * TYPE's for your driver's device components.
     *
     * The type at index 0 in the array should be the type which
     * provides the best mapping. Please return an empty array if the
     * discovery cannot be mapped.
     */
    public TypeInfo[] getValidDatabaseTypes()
    {
        return new TypeInfo[]{BYourDriverDevice.TYPE.getTypeInfo()};
    }
}

```

4. Your driver's device id structure, acting also as the device discover params structure, also needs to define the getDiscoverRequestType method (inherited from the interface BIDDfDiscoverParams) and return the BYourDriverPingRequest that you created in one of the previous day's lessons. This tutorial will soon show you how to update BYourDriverPingRequest to serve as the device discovery request too.

```

/**
 * The implementing class needs to return a Type that represents
 * the discover request from the driver whose discoverId can be an
 * instance of this object's class. If the class supports more than
 * one discover request type, then this should return the type that is
 * the most appropriate for the particular instance of the implementing
 * class.
 *
 * This is fundamental to dev driver's auto-discovery features.
 *
 * @return review method description.
 */
public Type getDiscoverRequestType()
{
    return BYourDriverPingRequest.TYPE;
}

```

5. Your driver's device id structure, acting also as the device discover params structure, also needs to define the `getDiscoverRequestTypes` method (inherited from the interface `BIDdfDiscoverParams`) and return an array of size 1, including whatever is returned by the `getDiscoverRequestType` method:

```

/**
 * Some drivers might require multiple, completely different requests
 * to be used to discover devices. This should not be the case here though.
 *
 * @return review method description.
 */
public Type[] getDiscoverRequestTypes()
{
    return new Type[]{getDiscoverRequestType()};
}

```

6. Your driver's device id structure, acting also as the device discover params structure, also needs to define the `getDiscoveryLeafType` method (inherited from the interface `BIDdfDiscoverParams`) and return its own type (since it is acting as both the discover params and as the discovery leaf):

```

/**
 * Some drivers could hypothetically group discovery objects. This
 * method returns the Type that will ultimately form the bottom-
 * most, non-grouped discovery objects. The columns of the device
 * manager will be determined from this type.
 *
 * In this case, the discovery leaf type is also BYourDriverDeviceId.
 *
 * @return review method description.
 */
public Type getDiscoveryLeafType()
{
    return getType();
}

```

NOTE: If your driver communicates *directly* over Tcp/Ip or Udp/Ip and *not through a Tcp/Ip or Udp/Ip gateway* then a discovery is probably not possible. You may be able to do a Udp/Ip multi-cast discovery, depending on your equipment. However, that is beyond the scope of this tutorial. You will probably have to skip the chapters that pertain to device discovery (today's lesson).

Chapter 24 - Create Your Device Discovery Request

In the previous chapter you reviewed your driver's protocol in search of a protocol request that retrieves information about multiple field-devices. If you were able to find such a request, we will ask you to create a corresponding request class for your driver. If you were unable to find such a request, and if your driver uses a serial, master-slave protocol, then we will ask you update your driver's ping request and ping response classes to make them also serve as the discover request and discover response.

A. If you created a `BYourDriverDeviceDiscoverParams` class in the previous chapter:

1. Create a class named **`BYourDriverDeviceDiscoverRequest`** that extends **`BDdfDiscoveryRequest`** in the package **`com.yourCompany.yourDriver.comm.req`**
2. Define the **`toByteArray`** method. In the **`toByteArray`** method, call `getDiscoverParameters()` and cast the result to `BYourDriverDeviceDiscoverParams`. Most of the information that you need will be available in this instance of your driver's **`BYourDriverDeviceDiscoverParams`** class. You created this class in the previous chapter.

```
package com.yourCompany.yourDriver.comm.req;

import com.tridium.ddf.identify.*;
import com.tridium.ddf.comm.req.*;
import javax.baja.sys.*;

public class BYourDriverDeviceDiscoverRequest
    extends BDdfDiscoveryRequest
{
    /*-
    class BYourDriverDeviceDiscoverRequest
    {
    }
    -*/

    public byte[] toByteArray()
    {
        BYourDriverDeviceDiscoverParams devDscvParams =
            (BYourDriverDeviceDiscoverParams)getDiscoverParameters();

        return new byte[]{
            //...
            // Substitute value1 and value2 with your own properties
            (byte)devDscvParams.getValue1(),
            (byte)devDscvParams.getValue2(),
            //...
        }
    }
}
```

3. Update **`BYourDriverDeviceDiscoverParams`** (from the previous chapter) - Finish defining the following method:

```
public Type getDiscoverRequestType()
{
    return BYourDriverDeviceDiscoverRequest.TYPE
}
```


B. If you modified the `BYourDriverDeviceId` class in the previous chapter to serve in the device discovery:

1. Modify the declaration of **`BYourDriverPingRequest`** and make it implement **`BIDdfDiscoverRequest`**. Also make it import the following additional packages:

```
...

import com.tridium.ddf.discover.*;
import com.tridium.ddf.identify.*;

...

public class BTestDriverPingRequest
    extends BDdfPingRequest
    implements BIDdfDiscoverRequest
{
    ...
}
```

2. Add the following property to the slotomatic statement of **`BYourDriverDevicePingRequest`**

```
/*-
class BYourDriverPingRequest
{
    properties
    {
        ...
        discoverParameters : BDdfIdParams
            -- This provides the necessary data that the toByteArray method
            -- Needs in order to construct the byte array.
            -- NOTE: During auto-discovery, the auto discovery job loops
            -- through all possible combinations of discoverParameters. Each
            -- pass through the loop, the next discoverParameters value for
            -- your driver is passed to this property. When you implement
            -- the toByteArray method, you may cast this to your own
            -- discoveryParameters class (that is what it will ultimately be).
            default{[new BYourDriverDeviceId()]}
        ...
    }
}
-*/
```

3. Add the following methods to `BYourDriverDevicePingRequest`. Please add them exactly as-is. You shouldn't need to modify these methods, except maybe to change the comments.

```
/**
 * The setDiscoverer method will be passed an instance of
 * IDdfDiscoverer. You need to retain the reference on
 * the instance and return it (whenever requested) from
 * the getDiscoverer method.
 */
IDdfDiscoverer discoverer = null;
/**
 * The BDdfAutoDiscoveryJob will pass an inner instance
 * of itself to the setDiscoverer method. In there, you
 * need to save away the reference. In here, please return
 * the most recent reference that was passed to the
 * setDiscoverer method.
 */
public IDdfDiscoverer getDiscoverer(){return discoverer;}
/**
 * The BDdfAutoDiscoveryJob will pass an inner instance
 * of itself here. Please save away the reference. Other
 * than that, you should not need to concern yourself
 * with this.
 */
public void setDiscoverer(IDdfDiscoverer discoverer)
{
    this.discoverer=discoverer;
}
```

NOTES:

- You already defined the toByteArray method during one of the previous day's lessons. You should not need to change it any further.
- In this scenario, since you are using the ping request as the discover request too, the values of the deviceId and discoverParameters properties will be equivalent. Since your toByteArray method has already been implemented to use the *deviceId* then that will suffice.

Chapter 25 - Create Your Device Discovery Response

1. Depending on whether you created a special discover request **BYourDriverDeviceDiscoverRequest** or whether you are having your driver's ping request **BYourDriverPingRequest** also serve as the discovery request, you should follow either **A** or **B** but not both.

A. If you created a class named **BYourDriverDeviceDiscoverRequest** in the previous chapters of the day's lesson:

Create a class named **BYourDriverDeviceDiscoveryResponse** that extends **BDdfResponse** and implements **BIDdfDiscoverResponse**. Create this class in the package **com.yourCompany.yourDriver.comm.rsp**.

```
package com.yourCompany.yourDriver.comm.rsp;

import com.tridium.ddf.comm.*;
import javax.baja.sys.*;

public class BYourDriverDeviceDiscoverResponse
    extends BDdfResponse
    implements BIDdfDiscoverResponse
{
    /*-
    class BYourDriverDeviceDiscoverResponse
    {
    }
    -*/
}
```

B. If you decided to have your driver's ping request and device id also serve as the discover request/discover parameters:

Update the class declaration for **BYourDriverPingResponse** and declare it so that it implements **BIDdfDiscoverResponse**. Also add a statement to import **com.tridium.ddf.discover.***.

```
package com.yourCompany.yourDriver.comm.rsp;

import com.tridium.ddf.comm.*;
import com.tridium.ddf.discover.*;

import javax.baja.sys.*;

public class BYourDriverPingResponse
    extends BDdfResponse
    implements BIDdfDiscoverResponse
{
    ...
}
```

2. In either case, please proceed as follows.
3. Define the **getDiscoveryChildren** method on the device discovery response. This is a requirement of the **BIDdfDiscoverResponse** interface.

```

/**
 * This method parses the response byte array and returns an
 * array of BYourDriverDeviceId objects describing the devices
 * that this response is able to identify. This is called during
 * the auto discovery process.
 */
public BIDdfDiscoveryObject[] parseDiscoveryObjects(Context c)
{
    return null;
}

```

4. The `parseDiscoveryObjects` method should return an array of `BIDdfDiscoveryObject` structures that describe the field-devices that are identified in the response bytes. We recommend that you return an array of `BYourDriverDeviceId` objects. It just so happens that `BYourDriverDeviceId` objects already implement `BIDdfDiscoveryObject` (since *BYourDriverDeviceId* extends *BDdfIdParams* which implements *BIDdfDiscoveryObject*).

```

/**
 * In our hypothetical protocol, we know that by receiving any
 * response to the ping request than the deviceId of the transaction
 * represents a device that is online.
 */
public BIDdfDiscoveryObject[] parseDiscoveryObjects(Context c)
{ // Returns an array of size one, containing just the deviceId
  // Of the response. Please note that for drivers that are not
  // Re-using the ping request as a discovery request, then this
  // Method should parse through the response bytes and make an
  // Array of BYourDriverDeviceId whose length corresponds to
  // The number of device entries that you determine are present
  // In the response bytes. Then assign each BYourDriverDeviceId
  // Entry in the array based on what you are able to parse from
  // The response bytes.
  return new BIDdfDiscoveryObject[]{
    (BIDdfDiscoveryObject)deviceId().newCopy()
  };
}

```

Chapter 26 - Create Your Manager Device Discovery Preferences

In the previous chapters of today's lesson, you defined a device discovery parameters object, device discovery request, and device discovery response. This chapter will tie all these classes together so that the developer driver's device manager can use them to discover the devices on your field-bus.

1. Create a class named **BYourDriverDeviceDiscoveryPreferences** that extends **BDdfAutoDiscoveryPreferences**. Create this in the package **com.yourCompany.yourDriver.discover**.
2. Redefine the **timeOut** property and specify the default amount of time that your driver should wait after transmitting your device discovery request before timing out.
3. Redefine the **retryCount** property and specify the default number of retries that your driver should attempt after a request times-out (before giving up on that particular request).
4. NOTE: Sometimes during a discovery process it could be helpful to specify shorter time-outs and less retries so that the entire device discovery process can complete sooner.
5. Redefine the **min** property and make the default be a copy of whatever your device discovery params class returns from its **getFirst** method.
6. Redefine the **max** property and make the default be a copy of whatever your device discovery params class returns from its **getLast** method.
7. **OPTIONAL:** Redefine the **doNotAskAgain** property and declare the default value as **true** if you do not want the integrator to receive a special prompt when he or she clicks the **Discover** button on the device manager. If you set this to true then the discovery process will automatically loop from the **min** to the **max** that you also specify here. Once you finish today's lesson, we encourage you to try this both ways and decide which way makes the most sense for your driver.
8. NOTE: Your **min** and **max** properties will be instances of the device id if you re-use your ping request as the discovery request. If not, your min and max will be instances of the **BYourDriverDeviceDiscoverParams** object that you created during today's lesson.

```
package com.yourCompany.yourDriver.discover;

import com.tridium.ddf.identify.*;
import com.tridium.ddf.discover.auto.*;

import com.yourCompany.yourDriver.identify.*;

import javax.baja.sys.*;

public class BYourDriverDeviceDiscoveryPreferences
    extends BDdfAutoDiscoveryPreferences
{
    /*-
    class BYourDriverDeviceDiscoveryPreferences
    {
        properties
        {
            timeout : BRelTime
                -- This is the amount of time to wait per field-bus request before timing out
                default{[BRelTime.makeSeconds(3)]}
                slotfacets{[BFacets.make(BFacets.make(BFacets.SHOW_MILLISECONDS,BBoolean.TRUE),
                    BFacets.MIN,BRelTime.make(0))]}

            retryCount : int
                -- This is the number of discovery field-message retransmissions
                -- per request.
                default{[1]}
                slotfacets{[BFacets.make(BFacets.MIN,BInteger.make(0))]}

            min : BDdfIdParams
                -- This is the id of the lowest device for your driver to attempt to
                -- learn by default
                default{[(BDdfIdParams)new BYourDriverDeviceId().getFirst()]}

            max : BDdfIdParams
                -- This is the id of the highest device for your driver to attempt to
                -- learn by default
```

```

        default{[(BDdfIdParams)new BYourDriverDeviceId().getLast()]}
    }
}
-*/
}

```

9. Redefine the **discoveryPreferences** property on BYourDriverNetwork and specify the default value to be an instance of BYourDriverDeviceDiscoveryPreferences.

- o Add the following import statements to the top of **BYourDriverNetwork** or **BYourDriverGatewayNetwork**:

```

import com.tridium.ddf.discover.*;
import com.yourCompany.yourDriver.discover.*;

```

- o For a **serial network**, your serial network component's slotomatic header should look something like this:

```

/*-
class BYourDriverSerialNetwork
{
    properties
    {
        communicator : BValue
        -- This plugs in an instance of yourDriver's
        -- communicator onto the serial network component.
        -- The Niagara station's platform will communicate
        -- over a serial port that is configured on this
        -- serial network. You can look at the property
        -- sheet of this communicator to review the exact
        -- settings.
        default{[ new BYourDriverCommunicator() ]}    }
        discoveryPreferences : BDdfDiscoveryPreferences
        -- This saves the last set of discovery preferences
        -- that the user provides on the device manager. It
        -- is also used as the default for the first Time
        -- that the user is prompted for a discovery.
        default{[ new BYourDriverDeviceDiscoveryPreferences()]}

    }
}
-*/

```

- o For a **TCP gateway network**, your gateway network component's slotomatic header should look something like this:

```

/*-
class BYourDriverGatewayNetwork
{
    properties
    {
        communicator : BValue
        -- This plugs in an instance of yourDriver's
        -- communicator onto the gateway network component.
        -- The Niagara station's platform will communicate
        -- directly to the corresponding gateway unit on the
        -- field-bus.
        default{[ new BYourDriverCommunicator() ]}    }
        discoveryPreferences : BDdfDiscoveryPreferences
        -- This saves the last set of discovery parameters
        -- that the user provides on the device manager. It
        -- is also used as the default for the first Time
        -- that the user is prompted for a discovery.
        default{[ new BYourDriverDeviceDiscoveryPreferences()] }
    }
    -*/

```

- For a **Tcp/Ip network** that is not a Tcp/Ip gateway network, device discovery is probably not possible.

SIDENOTE: If the network supports UDP Multi Cast capabilities then an auto-learn might still be possible. However, UDP Multi Cast is a subject that is beyond the scope of this tutorial.

10. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

Day 4 Conclusion

Congratulations! Your driver is now almost finished. In today's lesson you created a **device discovery parameters** structure, a **device discovery request**, a **device discovery response**, and a **device discovery preferences** structure. In Java code, you made some associations between these and your driver's network. By doing this, your driver should now support device discovery.

To discover field-devices on your field-bus:

1. **Run a station and view your network's Dev Device Manager**

- Run a station that has your driver's network component under the *drivers* folder.
- Open a Workbench.
- Connect to your station from Workbench.
- Expand the station in the *Nav* tree.
- Expand the *Config* component in the *Nav* tree.
- Expand the *Drivers* component in the *Nav* tree.
- Double-click your driver's network that you added in the previous day's lessons.
- Verify that the **Dev Device Manager** appears.
- You should see a **Discover** button at the bottom of the **Dev Device Manager**.

2. **Discover the devices on your field-bus.**

- Click the **Discover** button.
- Provided that you did not set the default value of the **doNotAskAgain** property on **BYourDriverDeviceDiscoveryPreferences** to *true* then you should see a window appear with property-sheet-like representation of **BYourDriverDeviceDiscoveryPreferences**. If you did decide to set the default of the **doNotAskAgain** property to **True** then the discovery procedure will instantly begin within the station.
- Click **Ok**.
- The discovery should now be occurring in the station.
- The job progress bar at the top of the **Dev Device Manager** should provide visual feedback about the completion status of the discovery process.

In your station, the driver will loop through all possible combinations of **BYourDriverDeviceParams** from the **min** to **max** that was specified on the window that popped up after you clicked the **Discover** button.

If all went well then you should see one or more rows in the top half of the **Dev Device Manager**. This area of the **Dev Device Manager** is called the *discover pane*. The columns in the *discover pane* should correspond to the properties in **BYourDriverDeviceId** that are defined the **MGR_INCLUDE** facet.

3. **Add one or more device components to your network.**

- Select a row in the *discover pane* and click the **Add** button.
- A window should appear that allows you to optionally edit some information.
- Click **Ok**.
- An instance of **BYourDriverDevice** should appear under your driver's network in your station.

Tomorrow we will show you how to update your driver to support discovery of data points in your device. The procedure for point-discovery is very similar to the procedure that you followed today for adding device-discovery support to your driver.

Day 5 - Add Point Discovery Capabilities to Your Driver

As you learned in yesterday's lesson, **discovery** is a special procedure that a Niagara AX installation technician follows to easily add driver **device** components and **control point** components to his or her Niagara AX station. Yesterday's lesson instructed you on how to design your driver to automatically add device components to your driver's network. You did this by defining four Niagara AX components in Java and making associations amongst them and your driver's network:

1. Device discover parameters
2. Device discover request
3. Device discover response
4. Device discovery preferences

If you are creating a driver that follows a serial, master-slave protocol then you likely modified the *device id* that you previously created and made it serve also as the *device discover parameters*. You likewise modified the *ping request* and *ping response* that you previously created and made them serve also as the *device discover request* and *device discover response*.

Alternatively, you may have decided to create a separate *device discovery parameters*, *device discovery request*, and *device discovery response* classes.

In both cases, you created a new class for the *device discovery preferences* structure.

By doing this, the developer driver framework uses these classes and associations in your driver to perform the discovery feature on your network's **Ddf Device Manager**.

If you follow the steps in today's lesson and update your driver accordingly, then the developer driver framework on which your driver is built, will automatically perform all details for the **point discovery** process. After following today's procedure, your driver will support discovery of data points within your device. Today's procedure is very similar to yesterday's procedure, although slightly more complicated.

Chapters - Day 5

- [Chapter 27](#)
- [Chapter 28](#)
- [Chapter 29](#)
- [Chapter 30](#)
- [Chapter 31](#)
- [Conclusion](#)

Chapter 27 - Create Your Point Discovery Parameters

NOTE: This class is analogous to your driver's read parameters, write parameters, and device discovery parameters classes that you created during the previous day's lessons. Just as you may have been able to use your driver's *device id* as the *device discovery parameters* class, you may likewise be able to use your driver's *read parameters* structure as the *point discovery parameters* structure.

DISCUSSION: Some protocols feature a message that can be sent to the field-device to ask the field-device for a list of data points. In that case, the field-devices sometimes return a list of information about all data points in the device. Alternatively, the request might ask the device for a particular group of data points that fall into a particular category or classification. If your driver's protocol supports such a message, then you will create five new classes today. If not, then you will create two new classes and modify your driver's *read parameters* structure, *read request*, and *read response* to also serve in the point discovery process. The latter scenario is similar to how the previously day's lesson instructed you to configure your *device id*, *ping request*, etc. to also serve the device discovery mechanism.

Please follow one of the next two paths (A or B, but not both).

A. If your driver's protocol features a message that can be sent to the field-device to ask it for a list data points:

1. Determine what information the request's byte array will need.
2. Create a class named **BYourDriverPointDiscoverParams** that extends **BDdfDiscoveryParams** in the package **com.yourCompany.yourDriver.identify**.

To do this, create a text file named `BYourDriverPointDiscoverParams.java` in the `jarFileName/src/com/yourCompany/yourDriver/identify` folder. Inside the text file, start with the following text:

```
package com.yourCompany.yourDriver.identify;

import com.tridium.ddf.identify.*;
import javax.baja.sys.*;

import com.yourCompany.yourDriver.comm.req.*;
import com.testCompany.yourDriver.discover.*;

public class BYourDriverPointDiscoverParams
    extends BDdfDiscoverParams
{
    /*-
    class BYourDriverPointDiscoverParams
    {
        properties
        {
        }
    }
    -*/
}
```

Based on your driver's protocol document, please be prepared to add properties to the slotomatic statement to help you construct the byte array in the point discovery request (you will create the point discovery request in the next chapter). You may return to this class and add properties to the slotomatic statement for any data values you require to construct the byte array for your point discovery request (to be created in the next chapter).

3. Define the **getFirst**, **getLast**, **getNext**, and **isAfter** methods. These are essential and allow the developer driver auto discovery process to automatically loop through all possible combinations of your discovery parameters.

NOTE: Yesterday's lesson explained how to do this for the device discovery process.

getFirst

The **getFirst** method should return an instance of `BYourDriverPointDiscoverParams` that represents the data that would be placed in the protocol request to request information about the first data point or series of data points. If your protocol has no way of identifying a definite *first* data point then you will need to be creative here and return an instance of `BYourDriverPointDiscoverParams` with all properties set to special values, that you will determine, such as `Integer.MIN_VALUE` for int properties.

getLast

The **getLast** method should return an instance of `BYourDriverPointDiscoverParams` that represents the data in the protocol request that would request information about the last device or series of devices. This might not be possible to accurately define for some protocols. If your protocol does not have a way of identifying the last series of data points, then you will need to be creative here and return an instance of `BYourDriverPointDiscoverParams` with all properties set to special values, that you will create, such as `Integer.MAX_VALUE` for int properties.

getNext

The **getNext** method should return an instance of `BYourDriverPointDiscoverParams` that represents the data in the protocol request that would request the next data point or series of data points, with respect to the particular instance of `BYourDriverPointDiscoverParams`. This method should review the values of some or all of the properties that you add to `BYourDriverPointDiscoverParams`. This method should return a new instance of `BYourDriverPointDiscoverParams` whose property values are incremented in such a way that the new instance would represent the data needed in the *toByteArray* method of the point discovery request in order to ask the field-device for information about the next data point or set of data points.

isAfter

The **isAfter** method will be passed an instance of `BYourDriverPointDiscoverParams`. You should review the property values of the given instance of `BYourDriverPointDiscoverParams` and return true if the current instance would request information about a data point or series of data points that would be *after* the data point or series of data points that the given instance of `BYourDriverPointDiscoverParams` would identify. If the current instance represents a data point or series of data points that are before or equal to the given instance, then this method should return false.

We understand that this can be a bit tricky, since the definitions of *first*, *last*, *next*, and *after* can be somewhat vague in some protocols. Please feel free to skip ahead and revisit this later if you wish.

As a reference, please see the sample **getFirst**, **getLast**, **getNext**, and **isAfter** methods that are illustrated below, for part B.

4. Define the **getDiscoverRequestType** method and make it return an instance of your driver's point discovery request. You will create the point discovery request later during today's lesson.

```
public Type getDiscoverRequestType()
{
    return BYourDriverDiscoveryRequest.TYPE;
}
```

5. Define the **getDiscoveryLeafType** method and make it return a reference to your driver's point discovery leaf *TYPE*. *The point discovery leaf will be created in a subsequent chapter of today's lesson.*

```
/**
 * This tells the developer driver framework that
 * instances of BYourDriverDiscoveryLeaf will be
 * placed into the discovery list of the point
 * manager to represent each data point that the
 * driver discovers.
 */
public Type getDiscoveryLeafType()
{
    return BYourDriverPointDiscoveryLeaf.TYPE;
}
```

B. If your driver's protocol does not feature a message that can be sent to the field-device to ask it for a list of data points then you will modify your driver's *read parameters* to also serve as the point discover parameters.

1. Open the **BYourDriverReadParams.java** file from day 2 of this tutorial.
2. Modify the class declaration statement and declare that **BYourDriverReadParams** implements **BIDdfDiscoverParams**.

```
public class BYourDriverReadParams
    extends BDdfReadParams
    implements BIDdfDiscoverParams
{
    ...
}
```

3. Your driver's read parameters structure, acting also as the point discover params structure, needs to satisfy the **BIDdfDiscoverParams** interface and define the **getFirst**, **getLast**, **getNext**, and **isAfter** methods in order to allow the developer driver auto discovery process to automatically loop through all possible combinations of your discovery parameters.
Please review the discussion concerning these methods as described in part A (in this chapter - above)

EXAMPLE: In the hypothetical protocol that we used back during day 2 of this tutorial, the read request retrieves data point values by asking the hypothetical field device for the values of all points of a certain type and direction (for example, analog inputs, analog outputs, digital inputs, or digital outputs).

In that scenario, the **getFirst** method would return an instance of `BYourDriverReadParameters` with a type string of *analog* and a direction string of *inputs*. The **getNext** method would return an instance of `BYourDriverReadParameters` with a type string of *analog* but with a direction string of *outputs*. That instance's **getNext** method would return an instance of `BYourDriverReadParameters` with a type string of *digital* and a direction string of *inputs*. That instance's **getNext** method would return an instance of `BYourDriverReadParameters` with a type string of *digital* but with a direction string of *outputs*. Finally, the **getLast** method will return an instance of `BYourDriverReadParameters` with a type string of *digital* and a direction string of *outputs*.

NOTE: In this example, we have imposed our own ordering for all of the possible combinations of the sample read parameters structure.

Please define your own version of these methods using the following example as a guide.

NOTE: Your driver's *read parameters* structure also needs to import packages:

- `javax.baja.registry.*`
- `com.tridium.ddf.discover.*`
- `com.yourCompany.yourDriver.discover.*`

```
package com.yourCompany.yourDriver.identify;

import javax.baja.sys.*;
import javax.baja.registry.*;

import com.tridium.ddf.identify.*;
import com.tridium.ddf.discover.*;

import com.yourCompany.yourDriver.comm.reg.*;
import com.yourCompany.yourDriver.discover.*;

public class BYourDriverReadParams
    extends BDdfReadParams
    implements BIDdfDiscoverParams
{
    ...
    ...
    ...

    /**
     * Niagara AX requires a public, empty constructor, so that it can perform
     * Its own introspection operations.
     */
    public BYourDriverReadParams(){}
    /**
     * This constructor is for our own convenience in the methods getFirst,
     * getNext, etc.
     */
    public BYourDriverReadParams(String typeString, String direction)
    {
        setTypeString(typeString);
        setDirection(direction);
    }
}
```

```

public BIDdfDiscoverParams getFirst()
{
    return new BYourDriverReadParams("analog","inputs");
}

public BIDdfDiscoverParams getLast()
{
    return new BYourDriverReadParams("digital","outputs");
}

public BIDdfDiscoverParams getNext()
{ // In all truth, our typeString and direction properties essentially
  // Could be combined into a single enumeration. Let's at least treat
  // It that was to help make this method simple.
  switch (getConvenientNumber()) // getConvenientNumber is a private
  {                               // method that we created below
      case 0: // Current = analog inputs  Next = analog outputs
          return new BYourDriverReadParams("analog","outputs");
      case 1: // Current = analog outputs Next = digital inputs
          return new BYourDriverReadParams("digital","inputs");
      case 2: // Current = digital inputs  Next = digital outputs
          return new BYourDriverReadParams("digital","outputs");
      //case 3:
      default: // Current = digital outputs Next = analog inputs
          return new BYourDriverReadParams("analog","inputs");
  }
}

public boolean isAfter(BIDdfDiscoverParams anotherId)
{ // In all truth, our typeString and direction properties essentially
  // Could be combined into a single enumeration. Let's at least treat
  // It that was to help make this method simple.
  return this.getConvenientNumber() > ((BYourDriverReadParams)anotherId).
getConvenientNumber();
}

/**
 * We created up this method to help with the isAfter and getNext methods.
 *
 * This method allows us to effectively enumerate all possible, reasonable
 * instances of this class.
 */
private int getConvenientNumber()
{ // This creates an "ordering" on all reasonable instances of this class.
  if (getTypeString().equals("analog") &&
      getDirection().equals("inputs"))
  {
      return 0;
  }
  else if (getTypeString().equals("analog") &&
           getDirection().equals("outputs"))
  {
      return 1;
  }
  else if (getTypeString().equals("digital") &&
           getDirection().equals("inputs"))
  {
      return 2;
  }
  else // "digital outputs" or anything else (invalid as it
        // might be)
  {
      return 3;
  }
}

...

```

```

    ...
    ...
}

```

4. Define the getDiscoverRequestType method and make it return a reference to your driver's read request *TYPE*.

REMINDER: In this scenario, your driver's read request will also serve during the point discovery process.

```

/**
 * The read request will also serve during the
 * point discovery process.
 */
public Type getDiscoverRequestType()
{
    return BYourDriverReadRequest.TYPE;
}

```

5. Define the getDiscoverRequestTypes method and make it return an array of size one, with the one element being a reference to your driver's read request *TYPE*.

REMINDER: In this scenario, your driver's read request will also serve during the point discovery process.

```

public Type[] getDiscoverRequestTypes()
{
    return new Type[]{BYourDriverReadRequest.TYPE};
}

```

6. Define the getDiscoveryLeafType method and make it return a reference to your driver's point discovery leaf *TYPE*. *The point discovery leaf will be created in a subsequent chapter of today's lesson.*

```

/**
 * This tells the developer driver framework that
 * instances of BYourDriverDiscoveryLeaf will be
 * placed into the discovery list of the point
 * manager to represent each data point that the
 * driver discovers.
 */
public Type getDiscoveryLeafType()
{
    return BYourDriverPointDiscoveryLeaf.TYPE;
}

```


Chapter 28 - Create Your Point Discovery Leaf

In this chapter you will create a class to serve as the **discovery leaf** for your driver's **Point Manager**.

Review and Discussion

Discovery Leaf - The discovery leaf is part of the developer driver framework that defines the contents of a row in the *Discovered* list of the *Device Manager* or *Point Manager*.

Device Id - In yesterday's lesson, you updated the class, *BYourDriverDeviceId*, to allow it serve as the discovery leaf for your driver's *Device Manager*. In today's lesson, you will create a dedicated class to serve as the discovery leaf for your driver's *Point Manager*.

Point Manager - The *Point Manager* appears in the Workbench when the point device extension, usually named "points" under a driver device component is double-clicked from within the *Nav Tree* of the Niagara AX Workbench. The *Discovered* list shows a spreadsheet-like listing of all data points in the field-device. This listing is really a list of all data points that an installation professional *may* (at his or option) import into the station to use with Niagara logic, web pages, etc.

Point Discovery Leaf - For point discovery, you will create a new class to serve as the discovery leaf.

As you may recall from the lesson on days 2 and 3, developer driver control points feature a *point id*, *read parameters*, and *write parameters* structure. Since developer driver control points feature three structures that define how to read, write, and parse the particular data point value, implementing the *point discovery leaf* is not quite as simple as re-using any one class (as was the case for re-using the *device id* as the device discovery leaf).

Please follow these steps to create your point discovery leaf:

1. Create a class named `BYourDriverPointDiscoveryLeaf` that extends `com.tridium.ddf.discover.BDdfPointDiscoveryLeaf`. Create this in the package `com.yourCompany.yourDriver.discover`. To do this, create a text file named `BYourDriverPointDiscoveryLeaf.java` in the `jarFileName/src/com/yourCompany/yourDriver/discover` folder. Inside the text file, start with the following text:

```
package com.yourCompany.yourDriver.discover;

import com.tridium.ddf.identify.*;
import com.tridium.ddf.discover.*;
import com.yourCompany.yourDriver.identify.*;

import javax.baja.sys.*;

public class BYourDriverPointDiscoveryLeaf
    extends BDdfPointDiscoveryLeaf
{
    /*-
    class BYourDriverPointDiscoveryLeaf
    {
        properties
        {
        }
    }
    -*/
}
```

2. Re-define the `readParameters`, `writeParameters`, and `pointId` properties of `BYourDriverPointDiscoveryLeaf`. The type needs to remain `BDdfIdParams` (from the package `com.tridium.ddf.identify`) for each. However, define the defaults for each as being the corresponding class that you created for your driver during days 2 and 3 of this tutorial. In addition, add the slotfacet `MGR_INCLUDE` to each. Use the following example as a guide:

```

package com.yourCompany.yourDriver.discover;

import com.tridium.ddf.identify.*;
import com.tridium.ddf.discover.*;
import com.yourCompany.yourDriver.identify.*;

import javax.baja.sys.*;

public class BYourDriverPointDiscoveryLeaf
    extends BDdfPointDiscoveryLeaf
{
    /**-
    class BYourDriverPointDiscoveryLeaf
    {
        properties
        {
            pointId : BDdfIdParams
                default{[new BYourDriverPointId()]}
                slotfacets{[MGR_INCLUDE]}
            readParameters : BDdfIdParams
                default{[new BYourDriverReadParams()]}
                slotfacets{[MGR_INCLUDE]}
            writeParameters : BDdfIdParams
                default{[new BYourDriverWriteParams()]}
                slotfacets{[MGR_INCLUDE]}
        }
    }
    -*/
}

```

3. Implement the **getDiscoveryName** method and return a Java string that will serve as the recommended name. Please note that Niagara AX installation professionals will *see* an instance of **BYourDriverPointDiscoveryLeaf** in the *Discovered* spreadsheet-like list for each data point that your driver finds. If the installation professional chooses to work further with a particular data point, then he or she will click the **Add** button to *add* that data point to the database and thereby create a fully-qualified driver control point for the item. The string that you return from this method will be assigned to the newly created driver control point, by default. The framework will automatically add suffixes, as necessary, to ensure unique names in the database. You may return a simple, static, string or you may compute the return string based on other information about the discovered point, as the following example shows:

```

/**
 * When a control point is added to the station from the
 * Point Manager, it is given this name by default (possibly
 * with a suffix to make it unique).
 */
public String getDiscoveryName()
{ // For the test driver, let's return a rather descriptive name
    String typeString = ((BYourDriverReadParams)getReadParameters()).getTypeString();
    String directionString = ((BYourDriverReadParams)getReadParameters()).getDirection();
    int offset = ((BYourDriverPointId)getPointId()).getOffset();
    // Capitalize the first letter of the typeString and the directionString
    typeString = Character.toUpperCase(typeString.charAt(0)) +
        typeString.substring(1);
    directionString = Character.toUpperCase(directionString.charAt(0)) +
        directionString.substring(1);
    // Concatenates everything together and returns the result, for example,
    // AnalogOutputs_1
    return typeString + directionString + '_' + offset;
}

```

Chapter 29 - Create Your Point Discovery Request

NOTE: This class is similar to your driver's read request, write request, ping request, and device discovery request classes. **Please follow either A or B but not both:**

1. **If your driver's protocol features a message that can be sent to the field-device to ask it for a list data points:**

- Create a class named **BYourDriverPointDiscoverRequest** that extends **BDdfDiscoveryRequest** in the package **com.yourCompany.yourDriver.comm.req**

*To do this, create a text file named **BYourDriverPointDiscoverRequest.java** in the `jarFileName/src/com/yourCompany/yourDriver/comm/req` folder. Inside the text file, start with the following text:*

```
package com.yourCompany.yourDriver.comm.req;

import javax.baja.sys.*;

import com.tridium.ddf.identify.*;
import com.tridium.ddf.comm.*;
import com.tridium.ddf.comm.req.*;
import com.tridium.ddf.comm.rsp.*;

import com.yourCompany.yourDriver.identify.*;

public class BYourDriverPointDiscoverRequest
    extends BDdfDiscoveryRequest
{
    /*-
    class BYourDriverPointDiscoverRequest
    {
    }
    -*/
}
```

- Define the **toByteArray** method.
In the **toByteArray** method, call `getDiscoverParameters()` and cast the result to **BYourDriverPointDiscoverParams**. Most of the information that you need will be available in this instance of your driver's **BYourDriverPointDiscoverParams** class. You created this class previously during today's lesson.

```

package com.yourCompany.yourDriver.comm.req;

import javax.baja.sys.*;

import com.tridium.ddf.identify.*;
import com.tridium.ddf.comm.*;
import com.tridium.ddf.comm.req.*;
import com.tridium.ddf.comm.rsp.*;

import com.yourCompany.yourDriver.identify.*;
import com.yourCompany.yourDriver.comm.rsp.*;

public class BYourDriverPointDiscoverRequest
    extends BDdfDiscoveryRequest
{
    /*-
    class BYourDriverPointDiscoverRequest
    {
    }
    -*/

    public byte[] toByteArray()
    {
        BYourDriverPointDiscoverParams ptDscvParams =
            (BYourDriverPointDiscoverParams)getDiscoverParameters();

        return new byte[]{
            //...
            // Substitute value1 and value2 with your own properties
            // As necessary to construct the outgoing frame that your
            // Driver's protocol defines for requesting all points or
            // A series (group) of points from a field-device.
            (byte)ptDscvParams.getValue1(),
            (byte)ptDscvParams.getValue2(),
            //...
        };
    }
}

```

HINT: If you require information about the field-device in order to construct the byte array that the **toByteArray** method returns then simply call the **getDeviceId** method and cast the result to **BYourDriverDeviceId**!

In fact, all requests have access to the device id in this fashion.

2. If your driver's protocol does not feature a message that can be sent to the field-device to ask it for a list of data points then you should modify your driver's *read request* to also serve as the point discover request.
 - o Modify the declaration of **BYourDriverReadRequest** and make it implement **BIDdfDiscoverRequest**. Also make it import the following additional packages:

```

...

import com.tridium.ddf.discover.*;
import com.tridium.ddf.identify.*;

...

public class BYourDriverReadRequest
    extends BDdfReadRequest
    implements BIDdfDiscoverRequest
{
    ...
}

```

- Add the following property to the slotomatic statement of **BYourDriverReadRequest**

```

/*-
class BYourDriverReadRequest
{
    properties
    {
        ...
        discoverParameters : BDdfIdParams
            -- This provides the necessary data that the toByteArray method
            -- Needs in order to construct the byte array. Since this class
            -- Is the read request and also serves as the discovery request,
            -- Then this property's value will be a copy of the value of the
            -- Read Parameters property.
            default{[new BYourDriverReadParams()]}
        ...
    }
}
-*/

```

- Add the following methods to BYourDriverReadRequest. Please add them exactly as-is. You shouldn't need to modify these methods, except maybe to change the comments.

```

/**
 * The setDiscoverer method will be passed an instance of
 * IDdfDiscoverer. You need to retain the reference on
 * the instance and return it (whenever requested) from
 * the getDiscoverer method.
 */
IDdfDiscoverer discoverer = null;
/**
 * The BDdfAutoDiscoveryJob will pass an inner instance
 * of itself to the setDiscoverer method. In there, you
 * need to save away the reference. In here, please return
 * the most recent reference that was passed to the
 * setDiscoverer method.
 */
public IDdfDiscoverer getDiscoverer(){return discoverer;}
/**
 * The BDdfAutoDiscoveryJob will pass an inner instance
 * of itself here. Please save away the reference. Other
 * than that, you should not need to concern yourself
 * with this.
 */
public void setDiscoverer(IDdfDiscoverer discoverer)
{
    this.discoverer=discoverer;
}

```

NOTES:

- You already defined the `toByteArray` method during one of the previous day's lessons. You should not need to change it any further. It pulls the data necessary to construct the outgoing frame from the *read parameters* structure.
- The *discover parameters* structure will be assigned a copy of the *read parameters* structure. This is a special case since you are using the *read request* also as a discovery request.

Chapter 30 - Create Your Point Discovery Response

NOTE: This class is similar to your driver's read response, write response, ping response, and device discovery response classes. Depending on whether you created a special discover request (**BYourDriverPointDiscoverRequest**) or whether your driver's read request (**BYourDriverReadRequest**) is also serving as the point discovery request, you should follow either **A** or **B** but not both.

A. If you created a class named **BYourDriverPointDiscoverRequest** previously during the day's lesson:

1. Create a class named **BYourDriverPointDiscoverResponse** that extends **BDdfResponse** and implements **BIDdfDiscoverResponse**. Create this class in the package **com.yourCompany.yourDriver.comm.rsp**.

To do this, create a text file named `BYourDriverPointDiscoverResponse.java` in the `jarFileName/src/com/yourCompany/yourDriver/comm/rsp` folder. Inside the text file, start with the following text:

```
package com.yourCompany.yourDriver.comm.rsp;

import java.util.*;

import javax.baja.sys.*;

import com.tridium.ddf.comm.*;
import com.tridium.ddf.comm.rsp.*;
import com.tridium.ddf.discover.*;

import com.yourCompany.yourDriver.discover.*;
import com.yourCompany.yourDriver.identify.*;

public class BYourDriverPointDiscoverResponse
    extends BDdfResponse
    implements BIDdfDiscoverResponse
{
    /*-
    class BYourDriverPointDiscoverResponse
    {
    }
    -*/
}
```

2. Define the `getDiscoveryChildren` method. This satisfies the **BIDdfDiscoverResponse** interface.

```
/**
 * This method parses the response byte array and returns an
 * array of BYourDriverPointDiscoveryLeaf objects describing
 * the data points that this response is able to identify.
 * This is called during the auto discovery process.
 */
public BIDdfDiscoveryObject[] parseDiscoveryObjects(Context c)
{
    return null;
}
```

3. Declare an empty constructor.

```
public BYourDriverPointDiscoverResponse()
{
}
```

4. Declare a constructor that takes as parameters any data that you will need to construct the return array for the **parseDiscoveryObjects** method.

```
/**
 * This constructor does not necessarily need to take an IDdfDataFrame
 * as a parameter. It could take any parameters that you wish to pass
 * to the response from the request's processReceive method. The data
 * that is passed to this constructor will be saved on instance variables
 * and used in the parseDiscoveryObjects method to construct the return
 * array.
 */
public BYourDriverPointDiscoverResponse(IDdfDataFrame receiveFrame)
{
    // TODO: Make a copy of any bytes that you need from the receiveFrame
    //       since the receive frame could be part of an internal buffer
    //       of the receiver.
}
```

5. Revisit your driver's point discovery request and implement the **processReceive** method. The **processReceive** method should return an instance of **BYourDriverPointDiscoverResponse**. When invoking the constructor of **BYourDriverPointDiscoverResponse**, pass in the received data frame and/or any data that you will need to construct the return array for the **parseDiscoveryObjects** method of **BYourDriverDiscoverResponse**.

```
package com.yourCompany.yourDriver.comm.req;

import java.util.*;

import javax.baja.sys.*;

import com.tridium.ddf.identify.*;
import com.tridium.ddf.comm.*;
import com.tridium.ddf.comm.req.*;
import com.tridium.ddf.comm.rsp.*;

import com.yourCompany.yourDriver.identify.*;

public class BYourDriverPointDiscoverRequest
    extends BDdfDiscoveryRequest
{
    /*-
    class BYourDriverPointDiscoverRequest
    {
    }
    -*/

    public byte[] toByteArray()
    {
        BYourDriverPointDiscoverParams ptDscvParams =
            (BYourDriverPointDiscoverParams) getDiscoverParameters();

        return new byte[]{
            //...
            // Substitute value1 and value2 with your own properties
            // As necessary to construct the outgoing frame that your
            // Driver's protocol defines for requesting all points or
            // A series (group) of points from a field-device.
            (byte) ptDscvParams.getValue1(),
            (byte) ptDscvParams.getValue2(),
            //...
        };
    }
}
```



```

public BIDdfResponse processReceive(IDdfDataFrame receiveFrame)
    throws DdfResponseException
{ // TODO: Pass any data to the response that it will need to
  // implement its parseDiscoveryChildren method.
  return new BYourDriverPointDiscoverResponse(receiveFrame);
}
}

```

6. Finish implementing the **parseDiscoveryObjects** method on **BYourDriverPointDiscoveryResponse**. The **parseDiscoveryObjects** method should return an array of **BIDdfDiscoveryObject** structures that describe the data points that are identified in the response bytes. More specifically, you should return an array of **BYourDriverPointDiscoveryLeaf** objects. You already defined your class *BYourDriverPointDiscoveryLeaf* during today's lesson. **BYourDriverPointDiscoveryLeaf** happens to implement **BIDdfDiscoveryObject**.

NOTE: Our hypothetical example is really a candidate for the other scenario being covered in this chapter. However, to illustrate this line item, please use the following implementation of the *parseDiscoveryObjects* method as a guide:

```

/**
 * In our hypothetical protocol, the request returns a comma-
 * separated list of analog inputs, analog outputs, digital
 * inputs, or digital outputs.
 */
public BIDdfDiscoveryObject[] parseDiscoveryObjects(Context c)
{
    // In our hypothetical protocol, the read request asks to read
    // either analog or digital outputs or inputs. The field device
    // Has up to 30 analog or digital outputs or inputs (the exact number
    // can vary). The read response returns all of the corresponding
    // analog input, analog output, digital input, or digital output values
    // that were requested -- inside a comma delimited string. The substrings in
    // between the commas are signed integers for analog values or
    // The text "on" or "off" for digital values. The actual values are not
    // important though. The position is the most important as it provides
    // us with vital information as to how to read and/or write the individual
    // data point.
    if (rawValues==null) // Parses each of the values from the
        parseRawValues(); // receive frame into a string array.
    BYourDriverPointDiscoveryLeaf[] discoveredPoints =
        new BYourDriverPointDiscoveryLeaf[rawValues.length];
    // Loops once for each data point in the response data.
    for (int i=0; i<discoveredPoints.length; i++)
    {
        discoveredPoints[i] = new BYourDriverPointDiscoveryLeaf();
        // Sets the typeString property of the discovery leaf
        ((BYourDriverReadParams)discoveredPoints[i].getReadParameters()).
            setTypeString(readParams.getTypeString());
        // Sets the direction property of the discovery leaf
        ((BYourDriverReadParams)discoveredPoints[i].getReadParameters()).
            setDirection(readParams.getDirection());
        // We know that by returning a value for this index that the data point
        // Exists in the hypothetical device
        ((BYourDriverPointId)discoveredPoints[i].getPointId()).setOffset(i);
        // TODO: We could update the writeParameters on the discovery leaf too
        // If we had anything there to update
    }
    return discoveredPoints;
}

```

B. If you decided to have your driver's read request and read parameters structure also serve as the discover request/discover parameters:

1. Update the class declaration for **BYourDriverReadResponse**.
 - Declare that it implements **BIDdfDiscoverResponse**.
 - Add a statement to import com.tridium.ddf.discover.*;
 - Add a statement to import com.yourCompany.yourDriver.discover.*;

```
package com.yourCompany.yourDriver.comm.rsp;

import javax.baja.sys.*;
import javax.baja.status.*;

import com.tridium.ddf.comm.*;
import com.tridium.ddf.comm.rsp.*;
import com.tridium.ddf.comm.req.*;
import com.tridium.ddf.discover.*;

import com.yourCompany.yourDriver.identify.*;
import com.yourCompany.yourDriver.point.*;
import com.yourCompany.yourDriver.discover.*;

public class BYourDriverReadResponse
    extends BDdfResponse
    implements BIDdfReadResponse, BIDdfDiscoverResponse
{
    ...
}
```

2. Define the `getDiscoveryChildren` method. This satisfies the **BIDdfDiscoverResponse** interface.

```
/**
 * This method parses the response byte array and returns an
 * array of BYourDriverPointDiscoveryLeaf objects describing
 * the data points that this response is able to identify.
 * This is called during the auto discovery process.
 */
public BIDdfDiscoveryObject[] parseDiscoveryObjects(Context c)
{
    return null;
}
```

3. The `parseDiscoveryObjects` method should return an array of **BIDdfDiscoveryObject** structures that describe the data points that can be identified in the response bytes. You should return an array of **BYourDriverPointDiscoveryLeaf** objects. You already defined this class (*BYourDriverPointDiscoveryLeaf*) and it happens to implement **BIDdfDiscoveryObject**. Please use the following implementation of the *parseDiscoveryObjects* method as a guide:

```

/**
 * In our hypothetical protocol, the request returns a comma-
 * separated list of analog inputs, analog outputs, digital
 * inputs, or digital outputs.
 */
public BIDDfDiscoveryObject[] parseDiscoveryObjects(Context c)
{
    // In our hypothetical protocol, the read request asks to read
    // either analog or digital outputs or inputs. The field device
    // Has up to 30 analog or digital outputs or inputs (the exact number
    // can vary). The read response returns all of the corresponding
    // analog input, analog output, digital input, or digital output values
    // that were requested -- inside a comma delimited string. The substrings in
    // between the commas are signed integers for analog values or
    // The text "on" or "off" for digital values. The actual values are not
    // important though. The position is the most important as it provides
    // us with vital information as to how to read and/or write the individual
    // data point.
    if (rawValues==null) // Parses each of the values from the
        parseRawValues(); // receive frame into a string array.

    BYourDriverPointDiscoveryLeaf[] discoveredPoints =
        new BYourDriverPointDiscoveryLeaf[rawValues.length];
    // Loops once for each data point in the response data.
    for (int i=0; i<discoveredPoints.length; i++)
    {
        discoveredPoints[i] = new BYourDriverPointDiscoveryLeaf();
        // Sets the typeString property of the discovery leaf
        ((BYourDriverReadParams)discoveredPoints[i].getReadParameters()).
            setTypeString(readParams.getTypeString());
        // Sets the direction property of the discovery leaf
        ((BYourDriverReadParams)discoveredPoints[i].getReadParameters()).
            setDirection(readParams.getDirection());
        // We know that by returning a value for this index that the data point
        // Exists in the hypothetical device
        ((BYourDriverPointId)discoveredPoints[i].getPointId()).setOffset(i);
        // TODO: We could update the writeParameters on the discovery leaf too
        // If we had anything there to update
    }
    return discoveredPoints;
}

```

C. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

Chapter 31 - Create Your Point Discovery Preferences

In the previous chapters of today's lesson, you defined a point discovery parameters object, point discovery leaf, point discovery request, and a point discovery response. This chapter will tie all of these classes together so that the developer driver's point manager can use them to discover the data points available on a field-device.

NOTE: This class is similar to the device discovery preferences class from yesterday's lesson.

1. Create a class named **BYourDriverPointDiscoveryPreferences** that extends **BDdfAutoDiscoveryPreferences**. Create this in the package **com.yourCompany.yourDriver.discover**. To do this, create a text file named *BYourDriverPointDiscoveryPreferences.java* in the jarFileName/src/com/yourCompany/yourDriver/discover folder. Inside the text file, start with the following text:

```
package com.yourCompany.yourDriver.discover;

import com.tridium.ddf.identify.*;
import com.tridium.ddf.discover.auto.*;

import com.yourCompany.yourDriver.identify.*;

import javax.baja.sys.*;

public class BYourDriverPointDiscoveryPreferences
    extends BDdfAutoDiscoveryPreferences
{
    /*-
    class BYourDriverPointDiscoveryPreferences
    {
        properties
        {
        }
    }
    -*/
}
```

2. In the slotomatic *properties* section, redefine the following properties:

timeOut

Specify the default amount of time that your driver should wait after transmitting your point discovery request before timing out.

retryCount

Specify the default number of retries that your driver should attempt after a request times-out (before giving up on that particular request).

NOTE: Sometimes during a discovery process it could be helpful to specify shorter time-outs and less retries so that the entire point discovery process can complete sooner.

min

Specify the default to be a copy of whatever your point discovery parameters class returns from its **getFirst** method (or whatever your read request parameters class returns from its **getFirst** method, if you chose to have it serve in the discovery process).

max

Specify the default to be a copy of whatever your point discovery params class returns from its

getLast method (or whatever your read request parameters class returns from its **getLast** method, if you chose to have it serve in the discovery process).

NOTE: Your **min** and **max** properties will be instances of the read request id if you re-use your read request as the discovery request. If not, your min and max will be instances of the **BYourDriverPointDiscoverParams** object that you created during today's lesson.

doNotAskAgain (*optional*)

Declare the default value as **True** if you do not want the integrator to receive a special prompt when he or she clicks the **Discover** button on the device manager. If you set this to true then the discovery process will automatically loop from the **min** to the **max** that you also specify here. Once you finish today's lesson, we encourage you to try this both ways and decide which way makes the most sense for your driver.

Please use the following source as a guide:

```
package com.yourCompany.yourDriver.discover;

import com.tridium.ddf.identify.*;
import com.tridium.ddf.discover.auto.*;

import com.yourCompany.yourDriver.identify.*;

import javax.baja.sys.*;

public class BYourDriverPointDiscoveryPreferences
    extends BDdfAutoDiscoveryPreferences
{
    /*-
    class BYourDriverPointDiscoveryPreferences
    {
        properties
        {
            timeout : BRelTime
                -- This is the amount of time to wait per field-bus request before timing out
                default{[BRelTime.makeSeconds(3)]}
                slotfacets{[BFacets.make(BFacets.make(BFacets.SHOW_MILLISECONDS,BBoolean.TRUE),
                    BFacets.MIN,BRelTime.make(0))]}

            retryCount : int
                -- This is the number of discovery field-message retransmissions
                -- per request.
                default{[1]}
                slotfacets{[BFacets.make(BFacets.MIN,BInteger.make(0))]}

            min : BDdfIdParams
                -- This is the id of the lowest data point for your driver to attempt to
                -- learn by default
                default{[(BDdfIdParams)new BTestDriverReadParams().getFirst()]}

            max : BDdfIdParams
                -- This is the id of the highest point for your driver to attempt to
                -- learn by default
                default{[(BDdfIdParams)new BTestDriverReadParams().getLast()]}

        }
    }
    -*/
}
```

3. Redefine the **discoveryPreferences** property on the **BYourDriverPointDeviceExt** class that you created back during the lesson for day 2. Specify the default value to be an instance of

BYourDriverPointDiscoveryPreferences.

NOTE: Please also add statements to import com.yourCompany.yourDriver.discover.*; and com.tridium.ddf.discover.*;

```
package com.yourCompany.yourDriver;

import javax.baja.sys.*;
import javax.baja.util.*;

import com.tridium.ddf.*;
import com.tridium.ddf.discover.*;

import com.yourCompany.yourDriver.point.*;
import com.yourCompany.yourDriver.discover.*;

public class BYourDriverPointDeviceExt
    extends BDdfPointDeviceExt
{
    /*-
    class BYourDriverPointDeviceExt
    {
        properties
        {
            discoveryPreferences : BDdfDiscoveryPreferences
            -- This saves the last set of discovery parameters that the user provided.
            -- It also allows the dev driver framework to automatically learn points
            flags{hidden}
            default{[ new BYourDriverPointDiscoveryPreferences()]}
        }
    }
    -*/
}
```

4. Run slotomatic and perform a full build on your driver (as described in [Chapter 2](#)).

Day 5 Conclusion

Congratulations! You have followed all of the steps of the [Developer Driver Tutorial](#)! In today's lesson you created a **point discover parameters** structure, a **point discovery leaf**, a **point discover request**, a **point discover response**, and a **point discovery preferences** structure. In Java code, you made some associations between these and your driver's point-device-extension. By doing this, your driver now supports point discovery.

To discover data points that are in a device:

1. Run a station and view your network's **Point Manager**

- Run a station that has your driver's network component under the *drivers* folder and that has at least one of your driver's device components under your driver's network component.
- Open a Workbench.
- Connect to your station.
- Double click the station.
- Expand the network in the navigation tree.
- Expand the device in the navigation tree that is under your network.
- In the navigation tree, double click the *Points* folder that is under the device.
NOTE: The *Points* folder is your driver's *point-device-extension*.
- Verify that the **Point Manager** appears.
- You should see a **Discover** button at the bottom of the **Point Manager**.

2. Discover the points on your field-bus.

- Click the **Discover** button.
- Provided that you did not set the default value of the **doNotAskAgain** property on **BYourDriverPointDiscoveryPreferences** to *true* then you should see a window appear with a property-sheet-like representation of **BYourDriverPointDiscoveryPreferences**.
- Click **Ok**.
- The discovery should now be occurring in the station.
- The job progress bar at the top of the **Point Manager** should provide visual feedback about the completion status of the discovery process.

In your station, the driver will loop through all possible combinations of **BYourDriverPointDiscoverParams** from the **min** to **max** that was specified on the window that popped up after you clicked the **Discover** button (or if you configured the default for the *doNotAskAgain* property to be *True* then the driver will loop automatically from the **min** to the **max**).

If all went well then you should see one or more rows in the top half of the **Point Manager**. This area of the **Point Manager** is called the *discover pane*. The columns in the *discover pane* should correspond to the properties that are declared with the **MGR_INCLUDE** facet on the *read parameters*, *write parameters*, and *point id* structures that you defined on *BYourDriverPointDiscoveryLeaf*.

3. Add one or more driver control point components to your network.

- Select a row in the *discover pane* and click the **Add** button.
- A window should appear that shows you the driver control point and its initial values that are about to be added to the database. You will be allowed to modify any data values that you desire. However, the defaults should be exactly as you declared in your discovery response's *getDiscoveryChildren* method.
- Click **Ok**.
- A Niagara AX control point, with an fully configured instance of your driver's proxy extension, should appear under your device's "Points" folder in your station.

The driver should automatically start polling the control point and updating the control point's *out* value.

Appendix 1 - Driver Actions

This appendix describes how to add an action to your driver's device or proxy extension in Niagara AX such that when the action is invoked, either by a Niagara-AX user or by some Niagara-AX logic, your driver instructs the field device to perform a special function.

For example, some driver protocols feature a *reboot* message that when sent to a field-device over the field-bus, the field-device will reboot. If your protocol features such a message then you can add a *reboot* action to your driver's device class. By following these steps, when the *reboot* action is invoked from Niagara-AX, your driver will transmit a *reboot* request to corresponding field-device causing the field-device to reboot.

Please note that the *reboot* example is just one of many possibilities. Your driver's protocol might not necessarily support *reboot* but it might support some other special behavior. Another somewhat typical example is a *change time* feature. In any event, the procedure for implementing this in your driver is as follows:

1. Add an action to your device.

- Add an action to the slotomatic statement in the java file for your driver's device.
 - Save the java file.
 - Run the slotomatic utility against your driver
 - Add an empty *do...* method to your device's java file. For example, if you called the action *something* in the slotomatic statement, then add a Java method called *doSomething* to your device's java file. The method needs to be defined with the *public* identifier.
-

2. Define a request and a response to tell the field-device what to do.

- Create a request class and a response class.
 - Make the request class extend **BDdfResponse**.
 - Call *getDeviceId* in the *toByteArray* method for the request.
 - Cast the result as an instance of your device's *device id* class.
 - Following your equipment's protocol, construct the byte array to instruct the appropriate field-device to do something.
 - Make the response extend **BDdfResponse**.
 - From the *processReceive* method of the request, return a *BDdfResponse* if the given data frame is as expected.
-

3. Have your *do...* method (for the new action that you added to your device) place an instance of the request onto the device's communicator.

- Call *getDdfCommunicator().communicate(new BYourDriverDoSomethingRequest());* from the *do...* method that defines the action's behavior.
 - Note that the call to *communicate* is a non-blocking call. Please read further for an explanation of how to perform special processing on the response.
-

4. To update your device after receiving the response to the *Do Something* request:

- Give the request a constructor that takes the device as a parameter.
- In your device's *do...* method, pass an instance of the device (using the Java *this* keyword) to the request.
- Make the request class implement the **BIDdfCustomResponse** interface from the *com.tridium.ddf.comm.rsp* package in the *devDriver* jar.

- Declare a *processResponse* method on the request. The developer driver framework will automatically call this method when the *do something* response is received for the *do something* request. The *do something* response is passed in as a parameter.
- In the *processResponse* method, you may use the reference to your device (the reference that you passed to the request's constructor) and update your device accordingly.
- Declare a *processTimeout* method on the response. This method is automatically called if the *do something* request times out. You may add any Java code here that you wish (or you may leave the method empty).
- Declare a *processErrorResponse* method on the response. This method is automatically called if the *do something* request's *processRecieve* method throws a *DdfResponseException*. You may add any Java code here that you wish (or you may leave the method empty).
- Declare a *processLateResponse* method on the response. This method is automatically called if the *do something* response is received after the request times out. Please note that this can only happen if your driver uses a multiple-transaction-communicator. You may add any Java code here that you wish (or you may leave the method empty).

Appendix 2 - Device Manager Buttons

Q: How do I add a button to the device manager?

A: `com.tridium.ddf.ui.device.BDdfDeviceMgrAgent`

- Make a class that extends `com.tridium.ddf.ui.device.BDdfDeviceMgrAgent`
- In your driver's module-include file, declare your device manager agent class as an **agent** on your driver's network.
- The **BDdfDeviceMgrAgent** class implements the interface **BIDdfDeviceMgrAgent**. Here are the method declarations for **BIDdfDeviceMgrAgent**. Please review these.

```
/**
 * This name can be either just a name or a lexicon key that defines the button text
 * and the
 * optional button label.
 */
public String getUiName();
/**
 * This method is called when the user clicks the
 * corresponding button on the device manager for this
 * agent. The developer may define any functionality
 * here.
 *
 * NOTE: This will execute on the client-side proxy's
 * virtual machine. Any access to the server-side
 * host will therefore have to be through properties,
 * actions, etc.
 *
 * @param a reference to the device manager
 * @param a reference to the network that the device manager is
 * operating upon
 *
 * @return an undo/redo command artifact or null
 */
public CommandArtifact doInvoke(BDdfDeviceManager deviceManager, BDdfNetwork network);

/**
 * This method is called when the ddf device manager
 * is created. It allows the developer to specify the MGR_CONTROLLER flags
 * that govern whether a button, menu item, toolbar item, etc. is created
 * for this agent.
 *
 * @return
 */
public int getFlags();

/**
 * The developer should review the given BDdfDeviceManager and consider
 * updating (eg. enable/disable) the given agentCommand and/or any other
 * commands on the manager's controller. The method is called anytime
 * there is a change of state on the device manager (eg. discovery list
 * selection change, database list selection change, database component
 * event, learn mode changed, etc.)
 *
 * For example (to enable the agent's UI widget(s) if one database item is selected):
 * agentCommand.setEnabled(deviceManager.getController().getSelectedRows().length ==
1);
 *
 * For example (to enable the agent's UI widget(s) if one or more database items are
selected):
 * agentCommand.setEnabled(deviceManager.getController().getSelectedRows().length > 0);
```

```

*
* For example (to enable the agent's UI widget(s) if zero database items are selected):
*   agentCommand.setEnabled(deviceManager.getController().getSelectedRows().length ==
0);
*
* @param deviceManager
*
* @param agentCommand this is a special instance of IMgrCommand. It is
*   a reference to the corresponding GUI command (button, menu item, and/or
*   toolbar button) on the device manager.
*/
public void update(BDdfDeviceManager deviceManager, DdfDeviceMgrAgentCommand
agentCommand);

```

- NOTE: **BDdfDeviceMgrAgent** provides default implementations of the **update** and **getFlags** methods. Therefore, you really only need to define the **getUiName** and **doInvoke** methods.
- **BDdfDeviceMgrAgent**'s default implementation of **update** does nothing. You should override this if you need to enable or disabled the agent's button or otherwise change the appearance of the device manager (to enable or disable other buttons, etc). Remember, the *update* method is called often (basically whenever there is any detectable change on anything related to the device manager).
- **BDdfDeviceMgrAgent**'s default implementation of **getFlags** always returns **MgrController.BARS**. This places your agent onto the device manager as a button, menu item, and toolbar item (if you specify a lexicon key that defines a toolbar icon from your **getUiName** method).
- Following this procedure will place a button, menu item, and (or) toolbar button onto the device manager (depending on the value that the *getFlags* method returns). When either of these are clicked on the Device Manager, your agent's **doInvoke** method will be called (on the client-side) Java virtual machine. This is where you may define any behavior that should happen as a result.

NOTE: In order to make complete use of a *Device Manager Agent*, some familiarity with aspects of the core **bajauri** and **baja driver** framework will be required. For example, the value returned by the *getFlags* method needs to be a Java int with various flags bit wise or'd from the *javax.baja.workbench.mgr.MgrController* class. These flags can be:

- **MgrController.MENU_BAR** causes the agent to be represented as an item in the Device Manager's main menu.
- **MgrController.ACTION_BAR** causes the agent to be represented as a button along with the other buttons towards the bottom of the Device Manager (such as the *Add*, *Edit*, and *Discover* buttons).
- **MgrController.TOOL_BAR** causes the agent to be represented as a toolbar item, provided that the agent's *getUiName* returns a *lexicon* base key **and** the lexicon base key contains a *.icon* definition (this is explained further below)
- **MgrController.BARS** causes the agent to be represented as all of the above. This is equivalent to **MENU_BAR | ACTION_BAR | TOOL_BAR**.

Example

Here is an example that places an item onto the menu bar, action bar, and toolbar.

BTestDeviceMgrButton.java

```

package com.testCompany.testDriver.ui;

import javax.baja.sys.Sys;
import javax.baja.sys.Type;
import javax.baja.ui.CommandArtifact;

import com.tridium.ddf.BDdfNetwork;
import com.tridium.ddf.ui.DdfMgrControllerUtil.DdfDeviceMgrAgentCommand;
import com.tridium.ddf.ui.device.BDdfDeviceManager;
import com.tridium.ddf.ui.device.BDdfDeviceMgrAgent;

public class BTestDeviceMgrButton
    extends BDdfDeviceMgrAgent
{
    /*-
    class BTestDeviceMgrButton
    {
    }
    -*/

    /*+ ----- BEGIN BAJA AUTO GENERATED CODE ----- +*/
    /*@ $com.testCompany.testDriver.ui.BTestDeviceMgrButton(1190303087)1.0$ @*/
    /* Generated Fri Jun 01 10:40:57 EDT 2007 by Slot-o-Matic 2000 (c) Tridium, Inc. 2000 */

    //////////////////////////////////////
    // Type
    //////////////////////////////////////

    public Type getType() { return TYPE; }
    public static final Type TYPE = Sys.loadType(BTestDeviceMgrButton.class);

    /*+ ----- END BAJA AUTO GENERATED CODE ----- +*/

    /**
     * This name can be either just a name or a lexicon key that defines the button text
    and the
     * optional button label.
     */
    public String getUiName()
    {
        return "DeviceMgr.TestButton";
    }

    /**
     * This method is called when the user clicks the
     * corresponding button on the device manager for this
     * agent. The developer may define any functionality
     * here.
     *
     * NOTE: This will execute on the client-side proxy's
     * virtual machine. Any access to the server-side
     * host will therefore have to be through properties,
     * actions, etc.
     *
     * @param a reference to the device manager
     * @param a reference to the network that the device manager is
     * operating upon
     *
     * @return an undo/redo command artifact or null
     */
    public CommandArtifact doInvoke(BDdfDeviceManager deviceManager, BDdfNetwork network)
    {
        System.out.println("'Test' button clicked on the device manager!");
        return null;
    }
}

```

```

/**
 * The developer should review the given BDdfDeviceManager and consider
 * updating (eg. enable/disable) the given agentCommand and/or any other
 * commands on the manager's controller. The method is called anytime
 * there is a change of state on the device manager (eg. discovery list
 * selection change, database list selection change, database component
 * event, learn mode changed, etc.)
 *
 * For example (to enable the agent's UI widget(s) if one database item is selected):
 * agentCommand.setEnabled(deviceManager.getController().getSelectedRows().length ==
1);
 *
 * For example (to enable the agent's UI widget(s) if one or more database items are
selected):
 * agentCommand.setEnabled(deviceManager.getController().getSelectedRows().length > 0);
 *
 * For example (to enable the agent's UI widget(s) if zero database items are selected):
 * agentCommand.setEnabled(deviceManager.getController().getSelectedRows().length ==
0);
 *
 * @param deviceManager
 *
 * @param agentCommand this is a special instance of IMgrCommand. It is
 * a reference to the corresponding GUI command (button, menu item, and/or
 * toolbar button) on the device manager.
 */
public void update(BDdfDeviceManager deviceManager, DdfDeviceMgrAgentCommand
agentCommand)
{
    // To disable the agent's button, toolbar button, and menu-item, I would need to
    // Do this:
    // agentCommand.setEnabled(false);
}
}

```

module-include.xml

The testDriver's module-include.xml file defines the BTestDeviceMgrButton as follows:

```

<types>
  <!-- Type Example:
  <type name="YourClass" class="com.yourDriver.BYourClass"/>
  -->
  <type name="TestDeviceMgrButton" class="com.testCompany.testDriver.ui.
BTestDeviceMgrButton">
    <agent>
      <on type="myTest:TestDriverNetwork"/>
    </agent>
  </type>

  ...
  <type name="TestDriverNetwork" class="com.testCompany.testDriver.BTestDriverNetwork"/>
  ...
</types>

```

This defines *BTestDeviceMgrButton* as an agent on the *BTestDriverNetwork* component. When initializing itself, the ddf device manager that BTestNetwork automatically receives (since it extends *com.tridium.ddf.BDdfNetwork*) reviews the network on which it is operating (*BTestNetwork* in this case) and adds action buttons, toolbar buttons, and menu items for each type in the driver whose class implements *BIDdfDeviceMgrAgent*

provided that the type is registered as an agent on the corresponding network type for the driver.

module.lexicon

```
DeviceMgr.TestButton.label=Test Device
DeviceMgr.TestButton.icon=module://myTest/images/myTest.png
DeviceMgr.TestButton.accelerator=CTRL+SHIFT+ALT+T
DeviceMgr.TestButton.description=This is a test button on the device manager
```

Please notice that the *getUiName* method returns "DeviceMgr.TestButton". The *module.lexicon* (which is a text, properties file) defines entries for:

- DeviceMgr.TestButton.label
- DeviceMgr.TestButton.icon
- DeviceMgr.TestButton.accelerator
- DeviceMgr.TestButton.description

Here is a description and explanation of each:

DeviceMgr.TestButton.label=Test Device

Defines the **text** that will be displayed inside the corresponding button on the device manager and for the corresponding menu item on the device manager.

DeviceMgr.TestButton.icon=module://myTest/images/myTest.png

OPTIONAL. Defines the **icon** that will be displayed inside the corresponding button, menu item, and toolbar button on the device manager. This identifies a sixteen-by-sixteen image of *png* format.

DeviceMgr.TestButton.accelerator=CTRL+SHIFT+ALT+T

OPTIONAL. Defines the **hot key** that will cause the corresponding button, menu item, and toolbar button to be invoked from the device manager.

DeviceMgr.TestButton.description=This is a test button on the device manager

OPTIONAL. Niagara will use the **description** as it deems necessary to provide a hint to the end-user explaining the function of the corresponding button, menu item, and toolbar button. For example, Niagara might place this description into the Workbench *status* line.

The *getUiName* method, however, does not have to return a base into the lexicon. Alternatively, the *getUiName* method can return a single key into the lexicon. If that is the case, then the device manager will use the corresponding text directly as the label for the button and the menu text. This manner does not define an icon, however, so no toolbar button will be generated (even if the *getFlags* method dictates otherwise).

Finally, if the String returned by the *getUiName* method is not found in your driver's lexicon at all then the String itself will be used as the button and menu item's label (no toolbar button will be generated).

Appendix 3 - Point Manager Buttons

Q: How do I add a button to the point manager?

A: `com.tridium.ddf.ui.point.BDdfPointMgrAgent`

- The procedure for this is very similar to the procedure for adding a button to the device manager as described in Appendix 2.
- Make a class that extends `com.tridium.ddf.ui.point.BDdfPointMgrAgent`
- In your driver's module-include file, declare your point manager agent class as an **agent on** your driver's device or point-device-extension.
- The **BDdfPointMgrAgent** class implements the interface **BIDdfPointMgrAgent**. Here are the method declarations for **BIDdfDeviceMgrAgent**. Please review these.

```
/**
 * This name can be either just a name or a lexicon key that defines the button text
 * and the
 * optional button label.
 */
public String getUiName();
/**
 * This method is called when the user clicks the
 * corresponding button on the point manager for this
 * agent. The developer may define any functionality
 * here.
 *
 * NOTE: This will execute on the client-side proxy's
 * virtual machine. Any access to the server-side
 * host will therefore have to be through properties,
 * actions, etc.
 *
 * @param a reference to the point manager
 * @param a reference to the network that is above the point device extension
 * that the point manager is operating upon.
 * @param a reference to the device that is above the point device ext
 * that the point manager is operating upon
 * @param a reference to the point-device-ext that the point manager
 * is operating upon.
 *
 * @return an undo/redo command artifact or null
 */
public CommandArtifact doInvoke(BDdfPointManager deviceManager, BDdfNetwork network,
BDdfDevice device, BDdfPointDeviceExt ptDevExt);

/**
 * This method is called when the ddf point manager
 * is created. It allows the developer to specify the MGR_CONTROLLER flags
 * that govern whether a button, menu item, toolbar item, etc. is created
 * for this agent.
 *
 * @return
 */
public int getFlags();

/**
 * The developer should review the given BDdfPointManager and consider
 * updating (eg. enable/disable) the given agentCommand and/or any other
 * commands on the manager's controller. The method is called anytime
 * there is a change of state on the point manager (eg. discovery list
 * selection change, database list selection change, database component
 * event, learn mode changed, etc.)
 */
```



```

* For example (to enable the agent's UI widget(s) if one database item is selected):
*   agentCommand.setEnabled(pointManager.getController().getSelectedRows().length == 1);
*
* For example (to enable the agent's UI widget(s) if one or more database items are
selected):
*   agentCommand.setEnabled(pointManager.getController().getSelectedRows().length > 0);
*
* For example (to enable the agent's UI widget(s) if zero database items are selected):
*   agentCommand.setEnabled(pointManager.getController().getSelectedRows().length == 0);
*
* @param pointManager
*
* @param agentCommand this is a special instance of IMgrCommand. It is
* a reference to the corresponding GUI command (button, menu item, and/or
* toolbar button) on the device manager.
*/
public void update(BDdfPointManager pointManager, DdfPointMgrAgentCommand agentCommand);

```

- NOTE: **BDdfPointMgrAgent** provides default implementations of the **update** and **getFlags** methods. Therefore, you really only need to define the **getUiName** and **doInvoke** methods.
- **BDdfPointMgrAgent**'s default implementation of **update** does nothing. You should override this if you need to enable or disabled the agent's button or otherwise change the appearance of the device manager (to enable or disable other buttons, etc). Remember, the *update* method is called often (basically whenever there is any detectable change on anything related to the point manager).
- **BDdfPointMgrAgent**'s default implementation of **getFlags** always returns `MgrController.BARS`. This places your agent onto the device manager as a button, menu item, and toolbar item (if you specify a lexicon key that defines a toolbar icon from your **getUiName** method).
- Following this procedure will place a button, menu item, and (or) toolbar button onto the point manager (depending on the value that the *getFlags* method returns). When either of these are clicked on the Point Manager, your agent's **doInvoke** method will be called (on the client-side) Java virtual machine. This is where you may define any behavior that should happen as a result.

NOTE: In order to make complete use of a *Point Manager Agent*, some familiarity with aspects of the core **bajauri** and **baja driver** framework will be required. For example, the value returned by the *getFlags* method needs to be a Java int with various flags bit wise or'd from the *javax.baja.workbench.mgr.MgrController* class. These flags can be:

- `MgrController.MENU_BAR` causes the agent to be represented as an item in the Point Manager's main menu.
- `MgrController.ACTION_BAR` causes the agent to be represented as a button along with the other buttons towards the bottom of the Point Manager (such as the *Add*, *Edit*, and *Discover* buttons).
- `MgrController.TOOL_BAR` causes the agent to be represented as a toolbar item, provided that the agent's *getUiName* returns a *lexicon* base key **and** the lexicon base key contains a *.icon* definition (this is explained further below)
- `MgrController.BARS` causes the agent to be represented as all of the above. This is equivalent to `MENU_BAR | ACTION_BAR | TOOL_BAR`.

Example

Please refer to the example provided in appendix 2. Although that example illustrates how to implement an agent on the device manager, everything required of the point manager is analogous to everything that is required for the device manager.

Appendix 4 - Exclusive Communications Access

Q: How do I gain exclusive access to the field-bus, in order to perform something special in my driver?

A: Override the **grantAccess** method in your driver's communicator.

The **grantAccess** method is called each time that a request is pulled off of the communicator's queue but before any processing occurs on the request. If the **grantAccess** method returns **false** for the request, then the request is placed back onto the tail of the communicator's queue, asynchronously, after a short period of time. You may define the **grantAccess** method to return **true** only for certain requests, depending on some internal state or scenario that you are maintaining or that you detect in your driver.

Please note that by default, the **grantAccess** method simply returns true. That allows all requests to proceed normally. To gain exclusive access to your field-bus, you may define any restrictions in your overridden version of this method.

Example

The following example allows only certain types of requests to be processed (transmitted onto the field-bus) when certain conditions are in effect. If those certain conditions are not in effect, then this example allows all requests to be processed (transmitted onto the field-bus).

```
/**
 * This method is called for each request that is dequeued.
 *
 * Overridden in Sample driver to provide exclusive access to the
 * communicator when doing terminal mode sessions and backups and/or
 * restores of the controllers.
 *
 * If returning true then the request will be processed
 * immediately (transmitted and scheduled for response checking). If
 * the developer returns false then the request will be placed back
 * in the back of the communicator queue.
 *
 * @param ddfRequest this is a request that was just pulled off of the
 * communicator queue. However, this request has not yet been processed.
 * This request will be processed immediately if this method returns
 * true. This request will be processed later if this method returns
 * false.
 *
 * @return true, unless the backup, restore, or terminal mode is active, and
 * the request is a Reload, ReloadLine, or Keystroke command.
 */
protected boolean grantAccess(BIDdfRequest ddfRequest)
{
    BSampleNetwork network = (BSampleNetwork)getParent();
    // In terminal mode, the end-user has visited a Vt100 terminal display from
    // within Workbench AX. He or she may type at the keyboard and the corresponding
    // keys are passed through to the field-device. While in this mode, routine
    // driver pinging, polling, etc. needs to temporarily stop.
    if(network.isTerminalModeActive())
        return ddfRequest instanceof BSampleKeystrokeRequest;

    // If the driver is backing-up the field-device, then routing pinging, polling,
    // etc. needs to temporarily stop so as not to confuse the field-device while
    // the back-up is occurring. The backup procedure uses BSampleKeystrokeRequest
    // to communicate to the field-device. Therefore, if the field-device is being
    // backed-up, then this only allows BSampleKeystrokeRequest requests to be
    // processed.
    if(network.getBackupModeActive()) // Backup procedure uses BSampleKeystrokeRequest
```

```
        return ddfRequest instanceof BSampleKeystrokeRequest; // To perform the backup

// If the driver is reloading the field-device then only those request types that
// are used during the reload process will be allowed to proceed. Any others will
// be denied access to the field-bus.
if(network.isReloadModeActive())
{
    return (ddfRequest instanceof BSampleReloadNetRequest) ||
           (ddfRequest instanceof BSampleReloadLineRequest) ||
           (ddfRequest instanceof BSampleKeystrokeRequest);
}

// If none of the above scenarios are in effect, then any request type may
// be granted access to the field-bus at this exact moment in time.
return true;
}
```

Appendix 5 - Defining Tags For Outgoing Requests and Incoming Frames

The `BDdfReceiver` features a **computeTag** method. Although this method might not be overridden for every driver, overriding it can be very helpful. The **computeTag** method is used to optimize the algorithm that matches incoming data frames with previously transmitted requests.

Requests (that extend **BDdfRequest** or implement **BIDdfRequest**) also feature a **getTag** method that can be overridden. Any incoming data frame whose tag, as computed by the **computeTag** method "equals" the tag that is returned from an outstanding request will be passed to the outstanding request's **processReceive** method.

By default, the **getTag** method of `BDdfRequest` returns `BString.DEFAULT`. Likewise, the default **computeTag** method from `BDdfReceiver` also returns `BString.DEFAULT`. This naturally causes all incoming data frames to be passed to the **processReceive** method of all outstanding requests.

By overriding these two methods and defining tags, the developer driver framework will automatically pass the appropriate incoming data frames to the appropriate outstanding requests.

If the **tag** for an incoming data frame does not match up with the **tag** of a recently transmitted request, then the incoming data frame will be routed to the driver's unsolicited receive handler. This is described further in the next appendix.

Appendix 6 - Processing Unsolicited Received Data Frames

General Discussion

Being built upon the developer driver framework, when a driver places a request on its communicator, the byte array representation of the request is transmitted onto the driver's field-bus. Subsequently, all data frames that are received before the request times out will be passed to the request's *processReceive* method. If the request defines the *getTag* method and if the communicator component's receiver defines the *computeTag* method, then only those incoming data frames whose tag matches the tag of the recently communicated request will be passed to the request's *processReceive* method.

If the developer does not implement the *getTag* method then all received frames will be passed to the recently transmitted request until the request times out. Otherwise, only those data frames with a tag that matches the request's tag will be passed to the request. In both scenarios, all such data frames are passed to the request until the request times out, returns a completed response from its *processReceive* method, **or** throws a *DdfResponseException* from its *processReceive* method.

There are two scenarios to consider where incoming data frames could be treated as unsolicited data frames:

1. In the case where request and receive frame tags are in use, any incoming data frame whose tag does not match the recently transmitted request (or requests, if the driver uses a multiple transaction communicator) will be passed to the communicator's unsolicited handler. Also any incoming data frame whose tag does indeed match the outstanding request(s) tag but the outstanding request(s)'s *processReceive* method neither returns a completed response nor throws a *DdfResponseException* (in other words, it returns null or throws a *RuntimeException*). then the incoming data frame will be passed to the unsolicited manager.
2. In the case where the driver does not define tags for its requests or incoming data frames, then any incoming data frame whereby when passed to the outstanding request (or all outstanding requests in the event of a multiple-transaction-communicator) and no outstanding request returns a response or throws a *DdfResponseException* when passed the incoming data frame (in other words, if the outstanding request(s) return *null* from the *processReceive* method or throw a *RuntimeException* when passed the data frame), then the data frame will be passed to the communicator's unsolicited handler

That being said, if a driver needs to process unsolicited data frames then the developer needs to create a class that extends **BDdfUnsolicitedMgr**. The developer should override the **processUnsolicitedFrame** method. The **processUnsolicitedFrame** method is passed the unsolicited data frame as a parameter. From there the developer may perform any custom processing that he or she deems necessary.

The **BDdfUnsolicitedManager** implements its own queue and thread. It automatically queues up unsolicited data frames for the communicator and processes each unsolicited frame on its own thread. Therefore, the driver's unsolicited manager's *processUnsolicitedFrame* runs on the driver's own, dedicated, unsolicited-receive handler thread.

After creating the unsolicited manager class, the developer needs to *associate* the unsolicited receive handler with his or her driver's communicator. This is accomplished by redefining the **unsolicitedMgr** property on the driver's communicator component. In fact, the **BDdfCommunicator** itself defines the **unsolicitedMgr** in its slotomatic header as follows:

```

unsolicitedMgr : BDdfUnsolicitedMgr
-- The simplest of drivers will not need unsolicited support
default{[new BDdfNullUnsolicitedMgr()]}

```

Notice that the type of the **unsolicitedMgr** is **BDdfUnsolicitedMgr**. Also notice that the default value for the **unsolicitedMgr** is an instance of a new **BDdfNullUnsolicitedMgr**.

As a side-note, the **BDdfNullUnsolicitedMgr** is a class that extends **BDdfUnsolicitedMgr** and provides an empty **processUnsolicitedFrame** method. Furthermore the **BDdfNullUnsolicitedMgr** removes the queue and the thread from the unsolicited manager, thereby serving a *null* version of an unsolicited manger. Since the developer will extend **BDdfUnsolicitedMgr** and not **BDdfNullUnsolicitedMgr**, his or her unsolicited manager will inherit its own queue and thread for processing unsolicited data frames.

The developer should redefine the **unsolicitedMgr** as follows, in his or her communicator (replacing *YourDriver* with the name of the developer's driver):

```

unsolicitedMgr : BDdfUnsolicitedMgr
-- Plugs in an instance of BYourDriverUnsolicitedMgr as the
-- unsolicited handler for BYourDriverCommunicator
default{[new BYourDriverUnsolicitedMgr()]}

```

Depending on the situation, there are endless possibilities that the developer may need to support in processing the unsolicited data frame. For example, if the developer needs the transmit a request onto the field-bus as a result of the unsolicited data frame, then his or her code in the unsolicited manager class may call the *getDdfCommunicator* method to gain a reference to the driver's communicator component. (*BDdfUnsolicitedMgr* provides the *getDdfCommunicator* method as a convenience.) With the communicator component, the developer may call the **communicate** method and pass it a new instance of a custom **BDdfRequest** for the driver.

This is just one small example out of countless scenarios. We wish you pleasant development!

Appendix 7 - Accessing the Point Id From the Read Request

Q. From the *toByteArray* method of my read request, how do I access the point ID? I can easily access the read parameters but not the point ID!

A. There is not necessarily a single point ID for the read request.

Discussion: Many protocols feature requests whose responses return the values for more than one data point. To accommodate polling in this fashion, devDriver combines (groups) all driver control points that have the equivalent *Read Parameters* into the same *Read Request*. The relationship between driver control points to a read request is potentially many-to-one. Because of this, a read request does not necessarily have just one *pointId*, it could have many *pointIds*.

In the event that many driver control points are automatically placed in the same read request, all of them will have the same *Read Parameters* (they all could have different *pointIds*). For this reason, the *pointId* is generally used from the *parseReadValue* method on the read response (as opposed to the read request). The philosophy is that the *readParameters* structure should have enough information to formulate the outgoing request. Then on the response side, the *pointId* should have any extra information (if there is any) that is required to index into the response and extract the data value for the particular data point.

Some protocols have a read request that retrieves many data values, but requires that each desired value be specifically requested. For example, in this scenario a request might state "*Please Read Inputs 1, 2, 3, 4, 5, 6, 7*". The response would then return the values that were requested (Inputs 1, 2, 3, 4, 5, 6, 7). In this scenario, you can get the array of point IDs that are assigned to the read request (see line item two just below). However, even in this scenario, it still might be better to add a property to the *readParameters* structure (perhaps an enumeration whose possible values would be *inputs1_7*, *inputs8_15*, etc.). Doing so would take maximum advantage of devDriver's ability to combine as many driver control points as possible into a single read request.

Final Answer:

To access the point ID for the read request, there are two options:

1. Move the particular property that you need to access from the *pointId* into the *readParameters*. This step is generally the suggested course of action because it takes maximum advantage of devDriver's ability to combine as many driver control points as possible into a single read request.
2. From your *toByteArray* method call *getReadableSource()*. This returns an array of *IDdfReadable*. These are the proxy extensions for all of the driver control points that share the same read parameters structure and therefore share the same read request. Then loop through the array of *IDdfReadables*, verify that each is an instance of your driver's proxy ext class (future support of virtual points anticipated), cast each to your driver's proxy ext class, call *getPointId()* upon each, cast the result to your driver's point id class, and directly access the point id. Please beware that this step might not allow devDriver to combine as many driver control points as possible into a single read request.

Note: Niagara will allow the user of the driver to add two identical control points to the database. This would be accomplished by choosing a row in the *Discovered* list of the point manager and simply adding it two or more times to the database. In this scenario, the devDriver framework will likewise group each of the identical database points into the same read request. This optimizes throughput on the field-bus by using one single read request to update each of the identical data points. Please keep this in mind while choosing option one or option two from above. If you choose option two but then always index into location zero of the readable source array (in a hard-coded manner) then please consider switching to use option one instead.