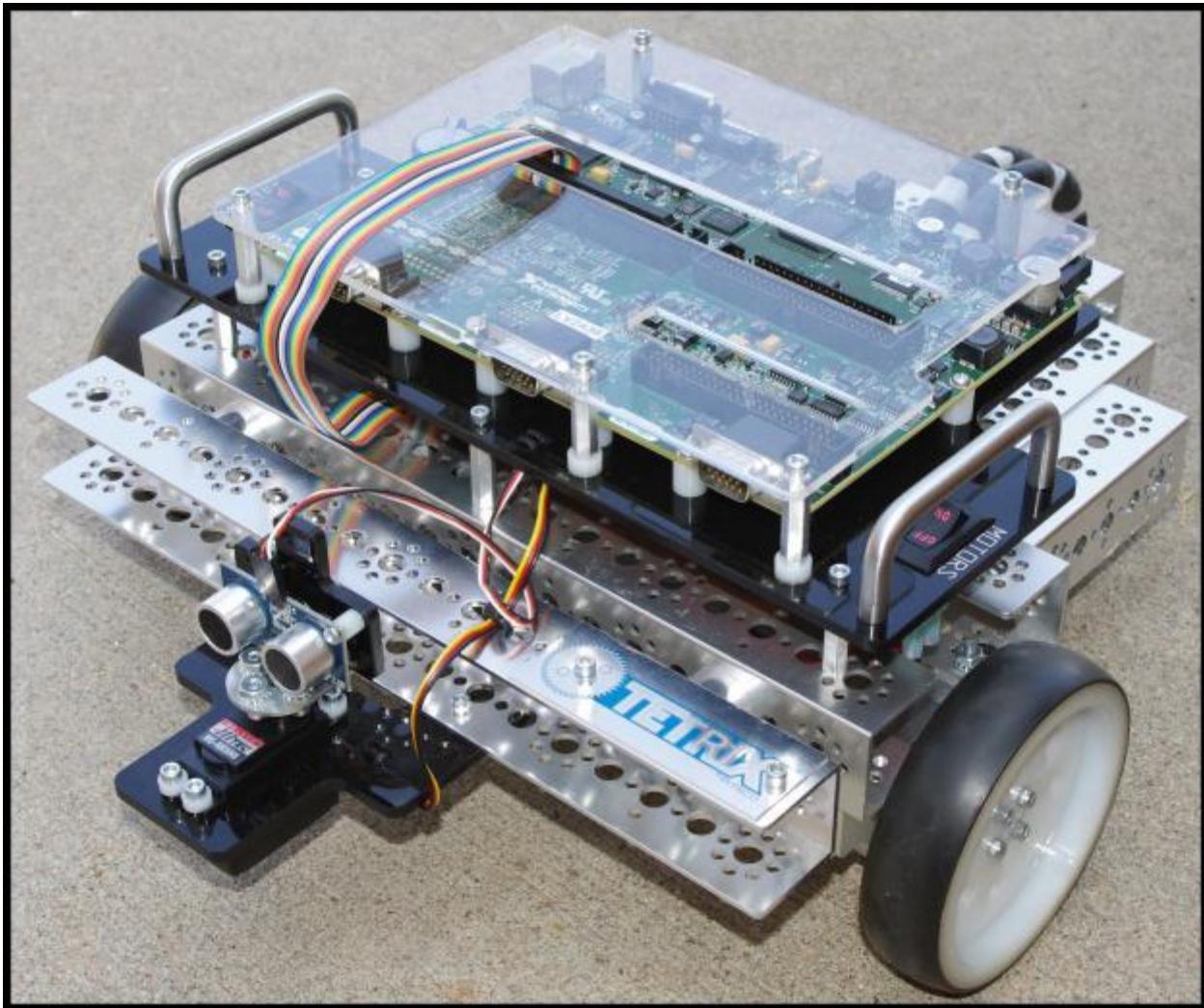


# Mobile Robotics Experiments with DaNI



Developed By:

**DR. ROBERT KING**  
**COLORADO SCHOOL OF MINES**



## Table of Contents

Introduction .....	5
Experiment 1 – LabVIEW and DaNI .....	9
Instructor’s Notes .....	9
Goals.....	9
Required Components.....	9
Background .....	10
Experiment 1-1 DaNI Setup.....	10
Establish Communications between DaNI and a host computer using the Hardware Wizard .....	16
Establish Communications between DaNI and a host without the Hardware Wizard .....	29
Creating a Project without the Hardware Wizard .....	34
Experiment 1-2 DaNI Test .....	38
Experiment 1-3 Evaluate the Operation of an Autonomous Mobile Robot.....	44
Experiment 1-4 Compare Autonomous and Remote Control .....	46
Experiment 2 – Ultrasonic Transducer Characterization.....	53
Instructor’s Notes .....	53
Goal .....	53
Background .....	53
Experiment 2-1 Characterization with the Roaming VI Graph .....	53
Experiment 2-2 Introduction to LabVIEW .....	57
Experiment 2-3 Ultrasonic Transducer Characterization.....	79
Experiment 3 – Motor Control .....	83
Instructor’s Notes .....	83
Goal .....	83
Background .....	83
Experiment 3-1 Open Loop Motor Control .....	83
Experiment 3-2 Closed Loop Motor Control.....	92
Experiment 4 - Kinematics .....	107
Instructor’s Notes .....	107
Goal .....	107
Background .....	107
Experiment 4-1 Turning and Rotating .....	107
Experiment 4-2 User Choice: LabVIEW Case Structure and Enum Data Type .....	112
Experiment 4-3 Using Hierarchical Programming to Drive from Start to Goal .....	117
Experiment 4-4 Steering Frame.....	125
Experiment 4-5 Grouping Steering Frame and Other Data in LabVIEW with Arrays and Clusters .....	129
Experiment 4-6 LabVIEW State Machine Architecture to Drive from Start to Goal with the Steering Frame.....	133
Experiment 5 – Perception with PING))).....	137

Instructor's Notes .....	137
Goal .....	137
Background .....	137
Experiment 5-1 Calibrating PING)))'s Orientation and File IO .....	137
Experiment 5-2 Displaying Perception Data with an XY Graph.....	144
Experiment 5-3 Communicating Perception Data to the Host with Network Streams...	146
Experiment 5-4 Feature Extraction - Identify Edges of an Obstacle .....	154
Experiment 5-5 Obstacle Avoidance.....	156
Experiment 5-6 Follow a Wall .....	156
Experiment 5-7 Gap feature extraction in the Roaming VI .....	156
Experiment 6 – Localization .....	173
Instructor's Notes .....	173
Goal .....	173
Background .....	173
Experiment 6-1 Odometric Localization (Dead Reckoning).....	173
Experiment 6-2 Localize with Range Data .....	175
Experiment 6-3 Occupancy grid map.....	176
Optional Projects and Competitions .....	177
Obstacle avoidance, Localization and Mapping .....	177
Obstacle avoidance, Localization, Mapping, and Object Recognition .....	177
Obstacle Avoidance, Mapping, and Navigation.....	177
Hardware Enhancement.....	178



# Introduction

Robotics and automation are becoming an essential component of engineering and scientific systems and consequently they are very important topics for study by engineering and science students. Furthermore, robotics is built on fundamentals like transducer characterization, motor control, data acquisition, mechanics of drive trains, network communication, computer vision, pattern recognition, kinematics, path planning, and others that are also fundamental to other fields, manufacturing, for instance. Learning these fundamentals can be challenging and fun by doing experiments with a capable mobile robot. The National Instruments (NI) LabVIEW Robotics Kit and LabVIEW provide an active-learning supplement to traditional robotics textbooks and curriculum by providing multiple capabilities in a compact and expandable kit.

National Instruments Corporation, located in Austin Texas, has been providing hardware and software that engineers and scientists use to design, prototype, and deploy systems for test, control, and embedded applications since 1976. The company has offices in over 40 countries, and NI open graphical programming software and modular hardware is used by more than 30,000 companies annually. More information is available at <http://www.ni.com>.

The experiments described herein show how to communicate between a host computer and a robot, how robots communicate with sensors to obtain data from the robot's environment, how to implement algorithms for localization and planning in LabVIEW software, how the robot communicates with actuators to control sensor motion and driving motion, how to implement algorithms for controlling sensor and motion. National Instruments LabVIEW is a graphical programming environment used by millions of engineers and scientists to develop sophisticated measurement, test, and control systems using intuitive graphical icons and wires that resemble a flowchart. It facilitates integration with thousands of hardware devices and provides hundreds of built-in libraries for advanced analysis and data visualization – all for creating virtual instrumentation. The LabVIEW platform is scalable across multiple targets and operating systems.

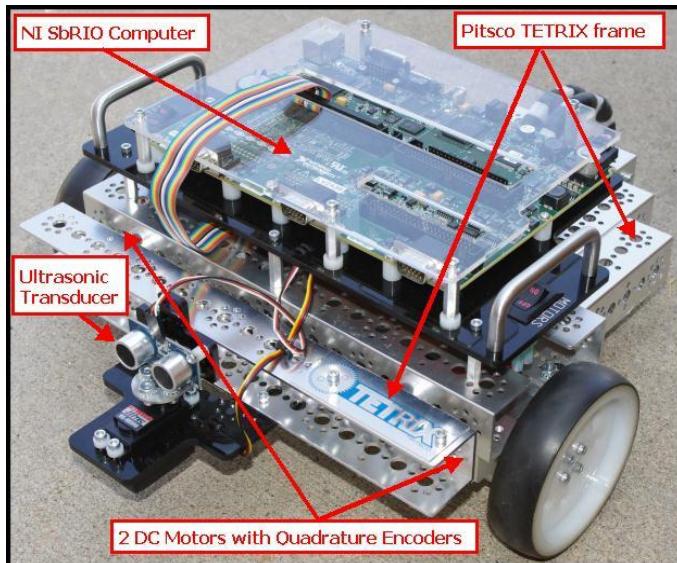
The NI LabVIEW robotics kit includes DaNI: an assembled robot with frame, wheels, drive train, motors, transducers, computer, and wiring. The hardware can be studied, reverse engineered, and modified by students. However, the major focus of the experiments is robot perception and control fundamentals that are implemented in LabVIEW software developed on a remote host computer and downloaded to the robot computer. To accomplish this goal, the experiments teach robotics fundamentals and LabVIEW programming simultaneously. The experiments are organized into subject matter areas, each containing introductory sections entitled Instructor's Notes, Goal, Required Components, and Background. These sections serve as a preview of the material and provide the requisite information.

There are several texts available that explain LabVIEW programming and several that explain robotics fundamentals. This document integrates the two with experiments in robotics with LabVIEW and DaNI. The experiments do not repeat the fundamental and theoretical material in

traditional introduction to robotics texts and courses. Rather, they help students discover robotics concepts in an active learning environment and show students how to implement robotics fundamentals. The robotics fundamentals for the experiments were drawn from *Introduction to Autonomous Mobile Robots*, 2nd edition, by Roland Siegwart, Illah R. Nourbakhsh, and Davide Scaramuzza 2011, ISBN 978-0-262-01535-6 hereafter referred to as Siegwart et al (2011).

Previous programming experience is not required to do these experiments. The experiments gradually build programming skills in LabVIEW. LabVIEW can be a very good first programming language as it is graphical, so students will visualize the logic of their programs. Students will use a LabVIEW program written by NI engineers in the first experiment and learn to write a simple LabVIEW program in the second experiment. Further experiments in the series will sequentially introduce more sophisticated LabVIEW programming techniques along with more sophisticated robotics fundamentals.

DaNI 2.0, the hardware portion of the LabVIEW Robotics Starter Kit that is used in these experiments, is an out-of-the-box mobile robot platform with sensors, motors, and an NI 9632 Single-Board Reconfigurable I/O (sbRIO) computer mounted on top of a Pitsco TETRIX erector robot base as shown in Figure 0-1.

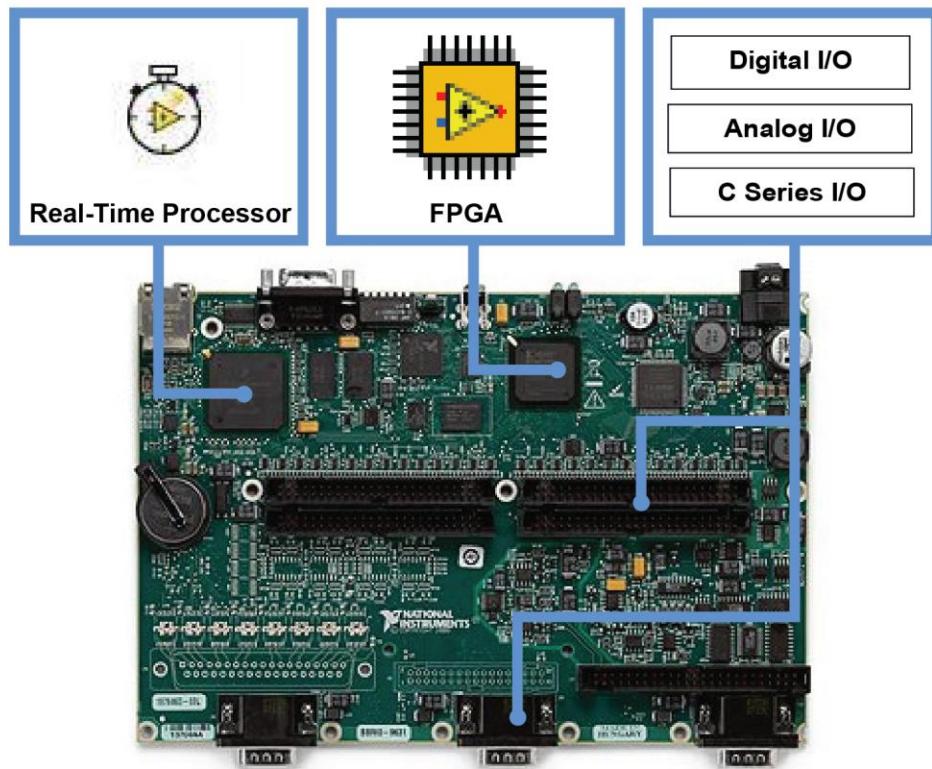


**Figure 0-1 DaNI 2.0 Main Components**

The kit is produced by PITSCO Education who provides kits, teacher guides, and classroom tools. More information is available at <http://www.pitsco.com>.

The NI Single-Board RIO, shown in Figure 0-2, is an embedded deployment platform that integrates a real-time processor, reconfigurable field-programmable gate array (FPGA), and

analog and digital I/O on a single board. This board is programmable with LabVIEW Real-Time, LabVIEW FPGA and LabVIEW Robotics software modules.



**Figure 0-2 The 9632 NI Single-Board RIO includes a real-time processor, FPGA, and built-in digital and analog I/O.**

The 2.0 starter kit includes ultrasonic and optical encoder sensors, but the reconfigurable I/O capability allows you to expand the kit to experiment with a variety of sensors including:

- LIDAR
- Radar
- Infrared
- Compass
- Gyroscopes
- Inertial Measurement Unit
- Global Position System
- Camera (CCD and CMOS)

Refer to [ni.com](http://ni.com) for information about the sbRIO and about connecting these sensors to the sbRIO and developing LabVIEW programs to acquire data from and to control them.



# **Experiment 1 – LabVIEW and DaNI**

## **Instructor's Notes**

This set of experiments was developed with DaNI 2.0, MS Windows 7, and LabVIEW 2011. Students will set up the software for DaNI and experiment with a prebuilt program that executes a vector field histogram (VFH) obstacle avoidance algorithm based on feedback from the included ultrasonic transducer. Students will study the VFH later in the experiment set. They will just observe the results in this experiment.

It is important that the battery be fully charged before beginning this experiment. Student should carefully read the battery information below so the battery has adequate life for the experiments.

This experiment may require students to assemble the robot and load software on a host computer unless that has been completed prior to beginning the experiment. Host computer administrator privileges are required to install the software. A Hardware Wizard is available to facilitate network and hardware configuration. The host computer must be configured for DHCP if the Hardware Wizard is used. If it has a static IP address, it will not be able to use the Hardware Wizard to download software. Instructions are provided in the following to configure communications and download software without using the Hardware Wizard but this extends the length and difficulty of the experiment. Even if the Hardware Wizard is used, the instructions for configuring without it are educational.

## **Goals**

Setup and test DaNI and a host computer. Test the software installation. Compare autonomous and remote control. Investigate DaNI's mobility platform. Introduce the LabVIEW project and the roles of development computer and onboard computer in robot software development and control.

## **Required Components**

Robotics Starter Kit 2.0 containing instruction sheets, DVD, charger, Ethernet crossover cable, and preassembled DaNI robot.

Host computer with Microsoft Windows Operating System. Administrator access privileges required for adding software unless software is preloaded. The required preloads are: LabVIEW and the Robotics, RealTime, and FPGA modules from the kit DVD.

An indoor roaming environment including objects for obstacles (ultrasonic transducer targets) like a cardboard boxes, furniture, laptop bag etc.

Linear distance measuring tool like a ruler, meter stick, or tape measure.

Angle measuring tool like a protractor.

A video capture device, smart phone or camera, is useful but not required.

A long (~ 3 m) cat 5 Ethernet cable for network connection or crossover cable for direct host connection is useful but not required. See explanation below for the connection and cable options.

## Background

This experiment is meant to be the first in a university engineering or science class. As such, students should have a background in physics. Little knowledge of robotics or LabVIEW programming is required for this first experiment, but if students have studied an introductory chapter in a textbook, like Siegwart et al (2011), students will have a better context for the material presented.

Additional information about DaNI and NI robotics is available at [ni.com/robotics](http://ni.com/robotics).

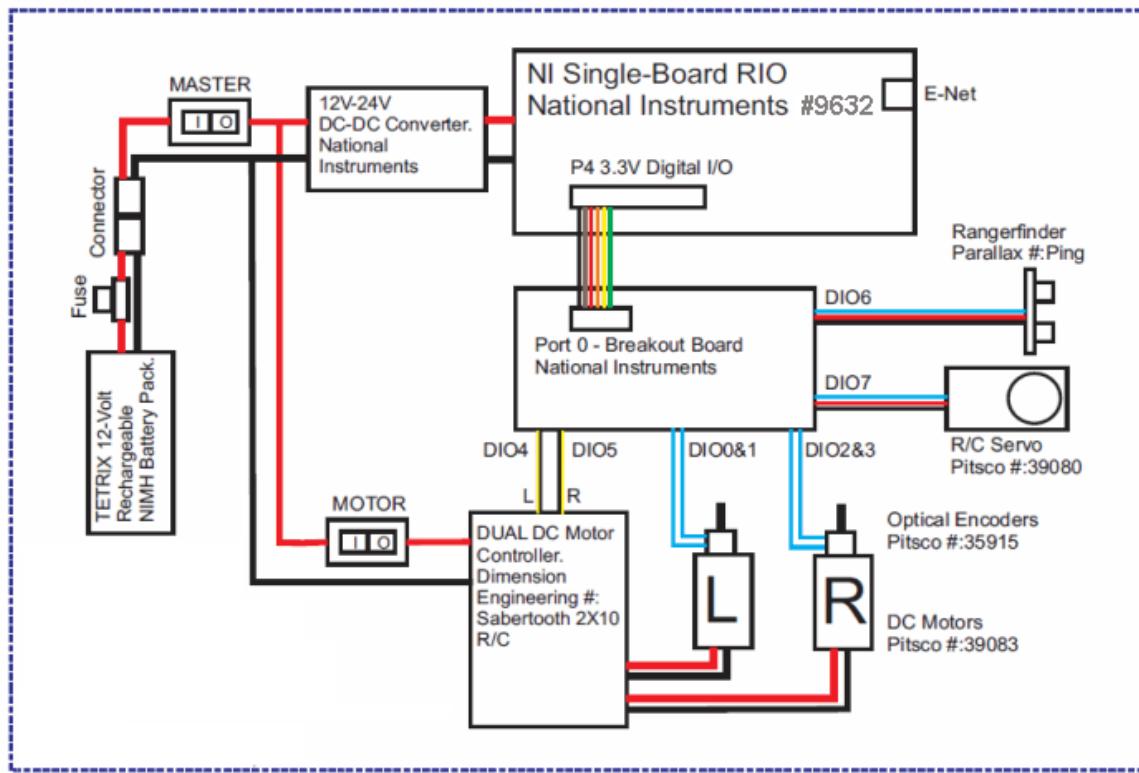
Students should study: the LabVIEW Robotics Starter Kit Safety Guide by navigating to the LabVIEW\readme directory on the DVD that is packaged with the kit and opening StarterKit\_Safety\_Guide.pdf

## Experiment 1-1 DaNI Setup

Study the robot components and connections. Figure 0-1 presents a block diagram of the connections to some major components.

The NI Robotics Starter Kit 2.0 contains hardware that requires special caution when you unpack, handle, and operate it. Refer to the LabVIEW Robotics Starter Kit Safety Guide for important information about protecting yourself from injury and protecting the Starter Kit hardware from damage. Access the LabVIEW Robotics Starter Kit Safety Guide by navigating to the LabVIEW\readme directory on the DVD that is packaged with the kit and opening

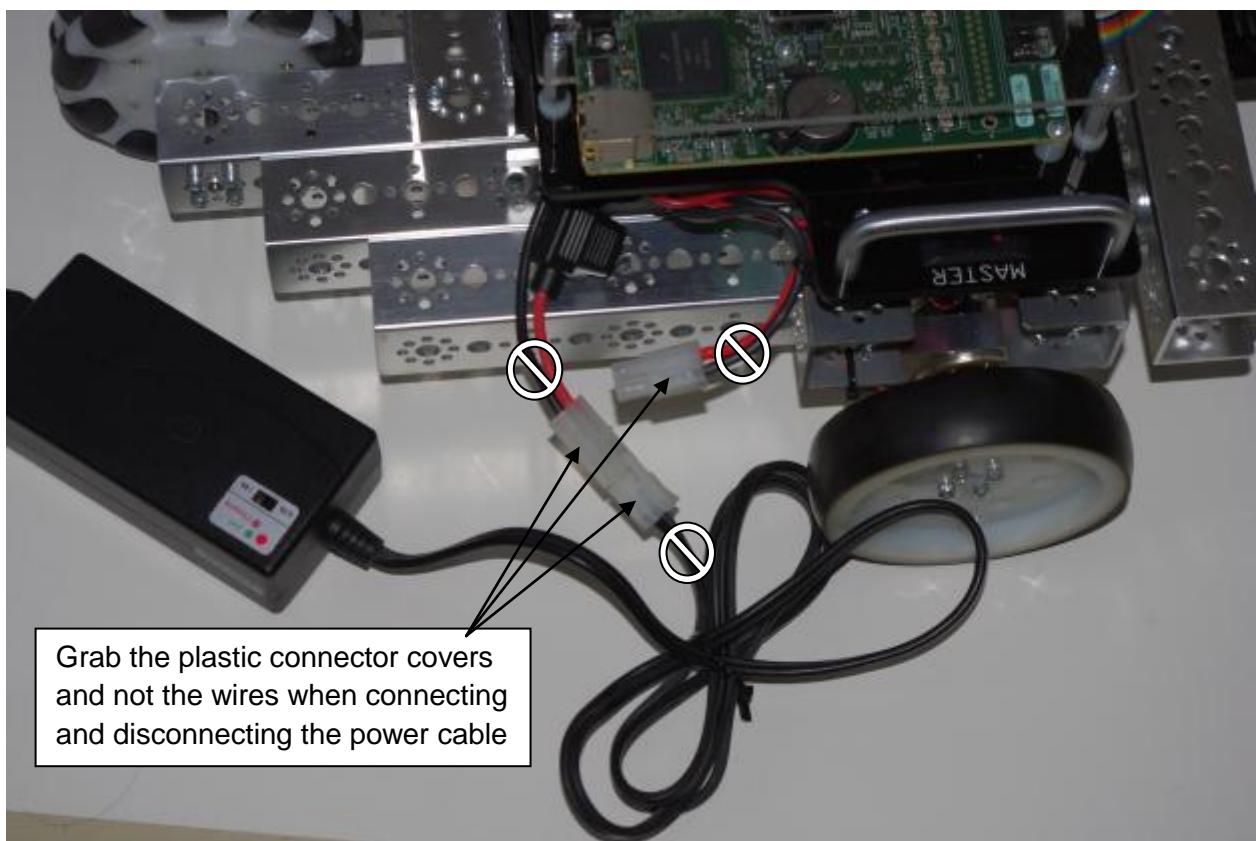
[StarterKit\\_Safety\\_Guide.pdf](#). You must have Adobe Reader 6.0.1 or later installed to view or search the PDF versions of the manual.



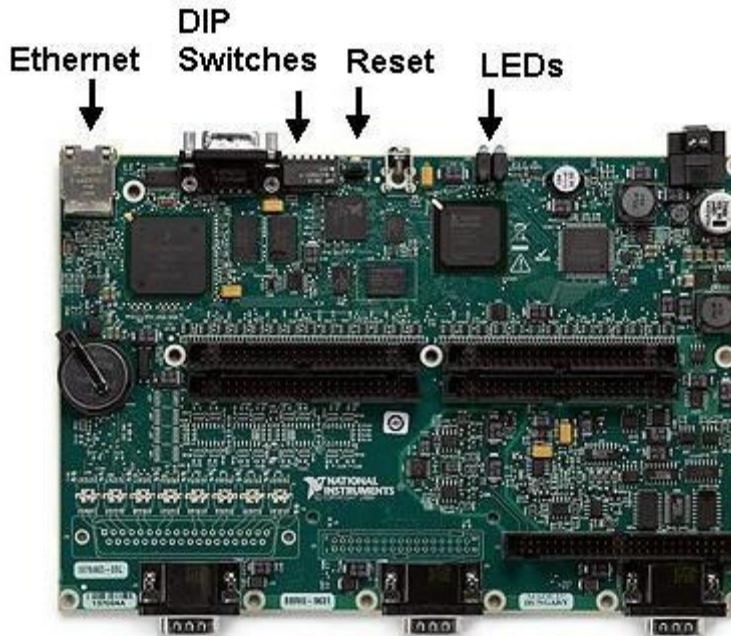
**Figure 0-1. DaNI 2.0 Hardware Component Block Diagram**

Turn off the master switch before connecting or disconnecting the charger. Plug in the power as shown in Figure 0-2 to charge the battery. The battery must be charged before the robot will operate. Note the location of the “connector” in Figure 0-1 relative to the other components. There is no connection available for external power to the motors so the battery is the only power source. The battery takes about 1.7 hours to charge. If the battery is low, the power LED (see Figure 0-3) might flash or not light up, the Ethernet link and activity lights will blink periodically in unison and the sensor motor might move unpredictably. Also, if the MOTORS switch is on, the drive motors might turn at slower than normal speeds. After the battery is charged, disconnect from the charger and connect the battery to the power input cable. The battery charge will last about 1 hour with the motors turned on and about 4 hours with the motors turned off. A switch on the charger can be set to 0.9 A and 1.9 A. Use the 1.9 A only when you need to charge quickly. Otherwise use the 0.9 A setting. Do not leave DaNI sitting with the master switch on. When not in use such as when developing software, turn off the master switch and place the robot on charge so the battery will be charged before software testing.

The above practice requires frequent charger connections. Always grab the plastic connector cover when disconnecting and connecting as shown in Figure 0-2. Do not pull on the power wires.



**Figure 0-2. DaNI Charger Connection**

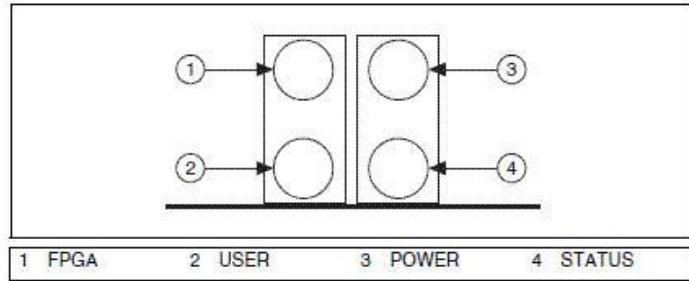


**Figure 0-3. Location of ethernet connection, DIP switches, Reset switch, and LEDs on the NI 9623 sbRIO**

The four-LED array shown in Figure 0-4 is helpful when troubleshooting. The POWER LED is lit while the NI sbRIO device is powered on. This LED indicates that the 5 V and 3.3 V rails are stable. The STATUS LED is off during normal operation. The NI sbRIO device indicates specific error conditions by flashing the STATUS LED a certain number of times:

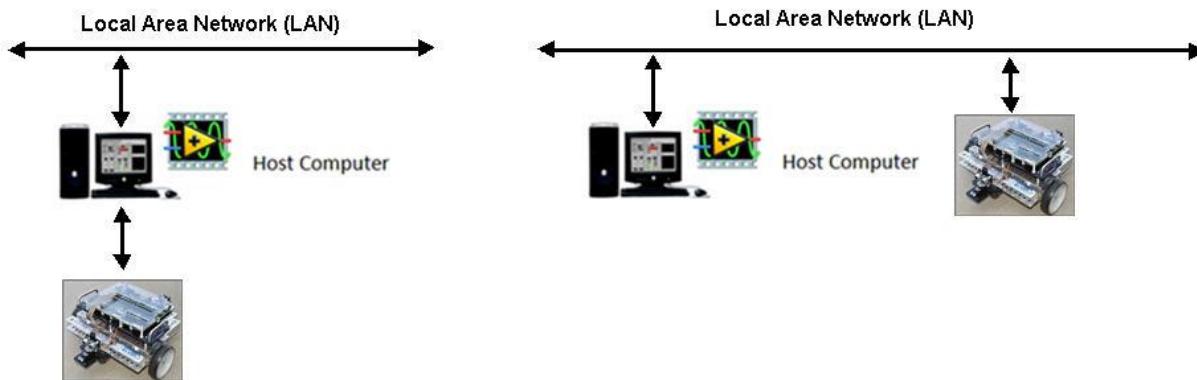
- One flash every couple seconds indicates that the sbRIO is unconfigured. The next section of the experiment will explain how to configure the device.
- Two flashes means that the device has detected an error in its software. This usually occurs when an attempt to upgrade the software is interrupted. Reinstall software on the device.
- Three flashes mean that the device is in safe mode because the SAFE MODE DIP switch is in the ON position. DIP switches will be explained later.
- Four flashes meant that the device software has crashed twice without rebooting or cycling power between crashes. This usually occurs when the device runs out of memory.
- Continuous flashing or solid indicates that the device has detected an unrecoverable error and the hard drive on the device should be reformatted.

You can control the User and FPGA LEDs from programs that you write.



**Figure 0-4. NI 9623 sbRIO LED Array**

A small mobile robot like DaNI has very limited on-board power, space, or payload ability. Consequently the on-board computer must be light weight, small, and low power. The sbRIO fills those requirements, but to do so, it is headless, meaning it doesn't have monitor, keyboard, or mouse peripherals. Also, it has limited capacity to store software. Therefore, it must communicate with a remote host computer where software is developed with a more capable operating system (OS) that supports peripherals and has storage capacity for the development software and OS. The robot can be connected directly to the host computer as shown on the left in Figure 0-5, or as shown on the right, connected directly to a local area network through a hub or switch. After robot control software is developed on the host computer, it is converted to a bitfile and downloaded to DaNI.



**Figure 0-5. Wired ethernet connections**

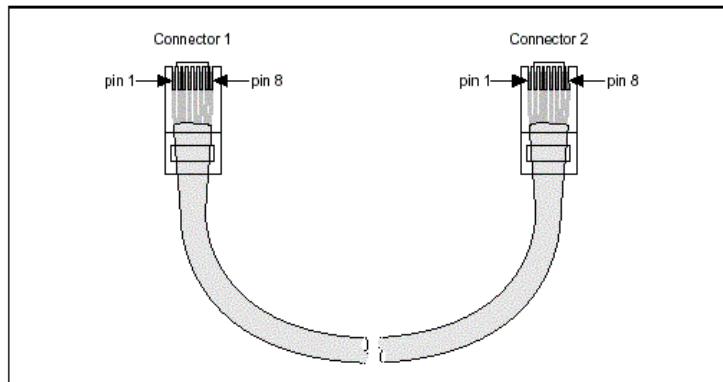
NI developed some software wizards to facilitate developing software and communicating between DaNI and a host computer. The wizards as well as 180-day evaluation of LabVIEW Robotics, LabVIEW Real-Time, and LabVIEW FPGA module software are on the DVD that is packaged with the kit. If it hasn't been preinstalled on your host computer, install the software from the DVD. Computer administrator privileges are required to install the software. The install may identify updates or patches. Copy the information so you can install these from the NI website after completing the DVD software installation.

Before you turn on the Master switch (shown in Figure 0-1), connect the robot to the computer with the cross over Ethernet cable provided in the kit or connect it to a network with a CAT 5 Ethernet cable. Then turn the motor switch off and the master power switch on (see “MOTOR” and “MASTER” in Figure 0-1). Note that the computer must use dynamic, or DHCP, instead of static IP addresses to download software with the Hardware Wizard.

Figure 0-5 shows the use of a crossover cable or a standard CAT 5 cable depending on the type of connection. As shown in the figure, a standard (straight through) cable without a crossover connector should be used when connecting hardware to a computer through an Ethernet hub or switch. If you don’t know if the Ethernet cables available to you are crossovers, a standard CAT 5 cable has a 1-to-1 mapping of pins from one connector to another, but a crossover cable connects the transmission pins from one end to the receive pins on the other end (crosses them over). Observe the color of the individual wires through the connectors. Figure 0-6 shows the color differences between a crossover cable and a standard Ethernet cable and the pinouts of a crossover cable. You can also use a multimeter to probe the connections. Connect the positive end of the multimeter to a pin on one end of the cable and probe the pins on the other end of the cable to determine if two ends have the same pin connections. If they are the same on both ends, it’s a straight through cable. Otherwise, it’s a crossover cable.

Ethernet cable conductor colors

Pin	Connector 1	Connector 2 (Standard)	Connector 2 (Crossover)
1	white/orange	white/orange	white/green
2	orange	orange	green
3	white/green	white/green	white/orange
4	blue	blue	blue
5	white/blue	white/blue	white/blue
6	green	green	orange
7	white/brown	white/brown	white/brown
8	brown	brown	brown



Ethernet cable pinouts

**Figure 0-6. Cat 5 Ethernet and crossover cable conductor colors and pinouts**

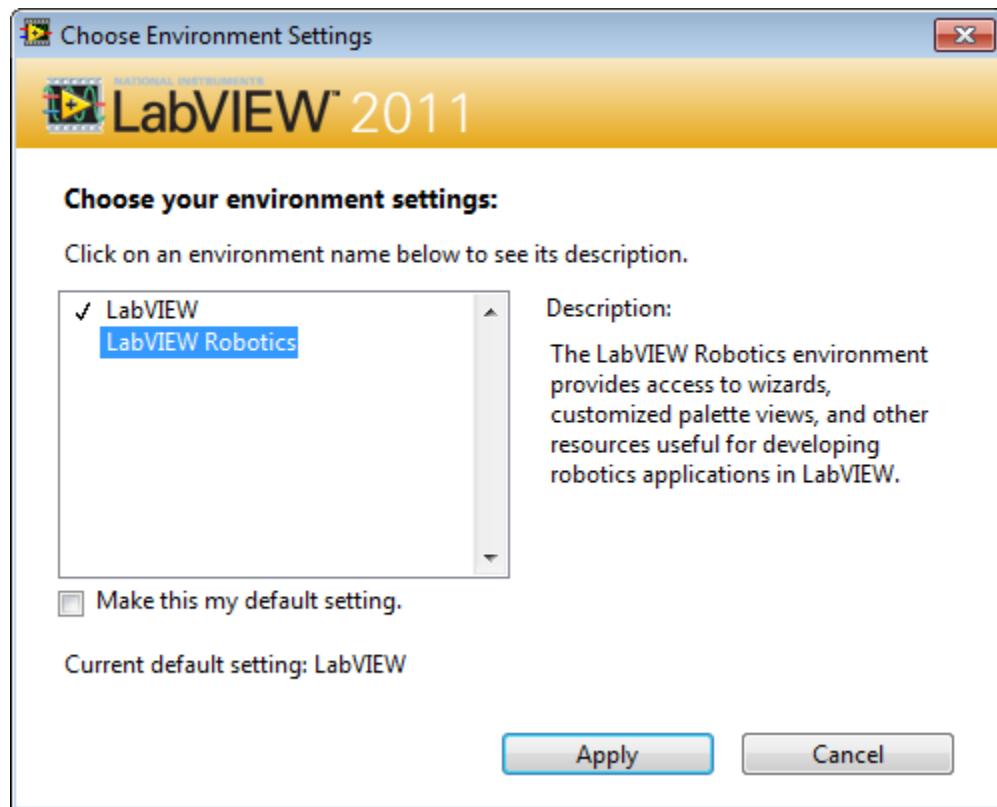
Before the host computer and DaNI can communicate over either type of network, the sbRIO must be configured. Configuration requires giving values for the IP (internet protocol) address, subnet mask, gateway and DNS (Domain Name System) server. The IP address is the unique address of a device on your network. Each IP address is a set of four one- to three-digit numbers in dotted decimal notation, for example 169.254.62.215. Each number is in the range from 0 through 255 and is separated by a period. The Subnet Mask determines whether DaNI is on the same network as the host. 255.255.255.0 is the most common subnet mask. Gateway is the IP address of a device that acts as a gateway server, which is a connection between two networks. The DNS Address is the IP address of a network device that stores host names and translates them into IP addresses. If you use the crossover-cable direct connection, the gateway and DNS can both be set to 0.0.0.0.

IP addresses can be assigned automatically with Dynamic Host Configuration Protocol (DHCP) or statically. A DHCP server on the network that is connected to the DaNI host assigns an IP address and other IP configuration parameters, such as the subnet mask, default gateway, and DNS server automatically to the host when it is booted so the host can communicate with other computers over the network. To determine whether your network is set to obtain an IP address automatically, click start and select Control Panel»Network Connections. Right-click Local Area Connection and select Properties. Select Internet Protocol (TCP/IP) and click Properties. If the Obtain an IP address automatically option is selected, your network uses DHCP. Otherwise, your network uses static IP addresses.

If DaNI is connected to a hub or switch on a network at an organization that controls network access, you will have to contact the people who control the access, such as the IT department, and request permission to connect to the network. The access controller may also assign IP addresses.

## **Establish Communications between DaNI and a host computer using the Hardware Wizard**

The hardware wizard is the easiest way to establish communication between the host computer and DaNI. It configures the communications automatically. Launch LabVIEW Robotics by clicking Start Menu » All Programs » National Instruments LabVIEW Robotics. If a window opens allowing you to choose the environment, choose LabVIEW Robotics as shown in Figure 0-7. If a Window doesn't open with that choice, choose Tools>>Choose Environment from the LabVIEW Menu Bar to open the window.



**Figure 0-7. LabVIEW Choose Environment Window**

Click on the Getting Started tab. Launch the Hardware Setup Wizard by clicking the Hardware Wizard icon shown in Figure 0-8. If you don't see the embedded video and the Hardware Setup Wizard, verify that you are on the "Getting Started" tab.

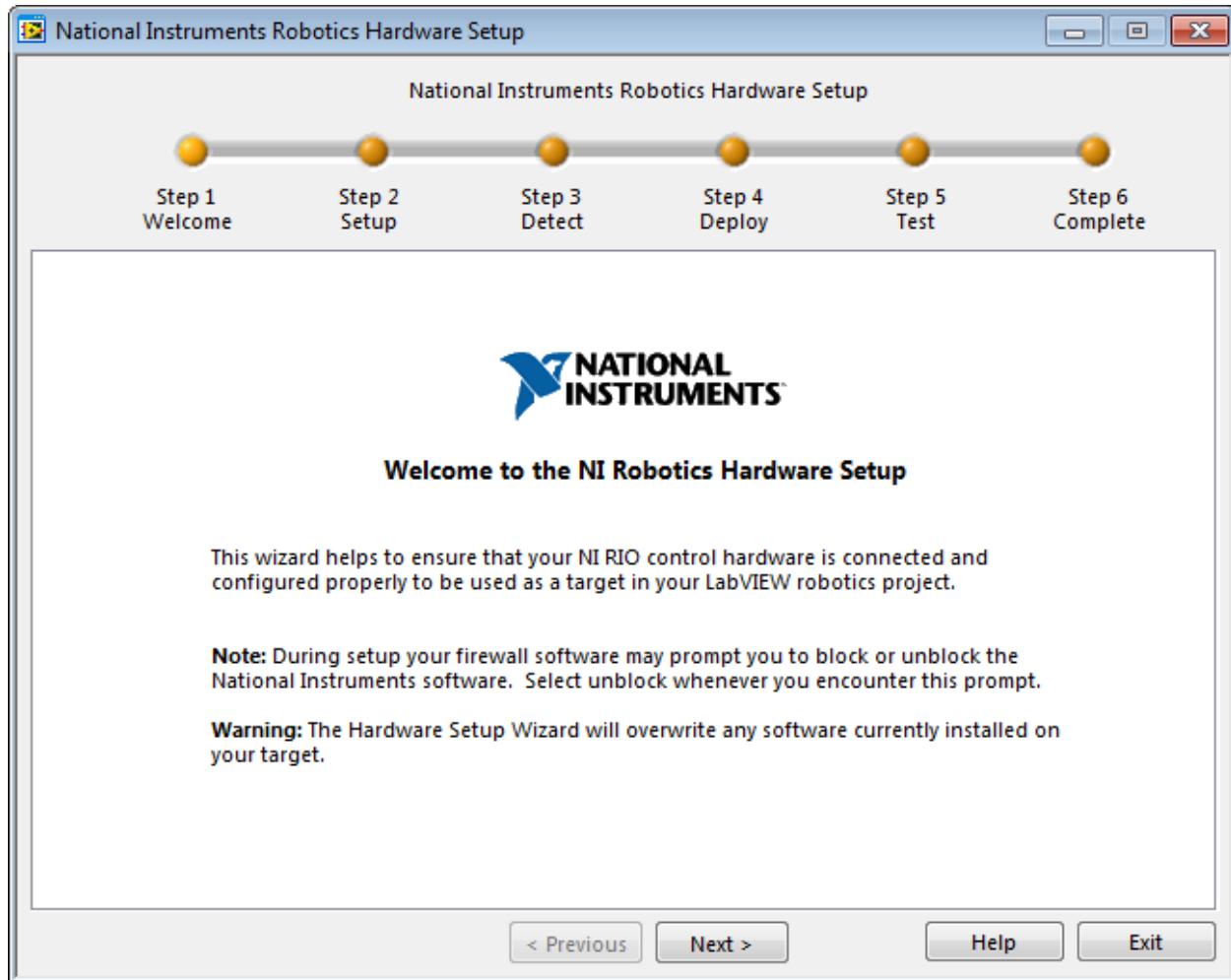
Help is available at: <http://zone.ni.com/devzone/cda/tut/p/id/10405>



**Figure 0-8. LabVIEW Robotics Getting Started Window**

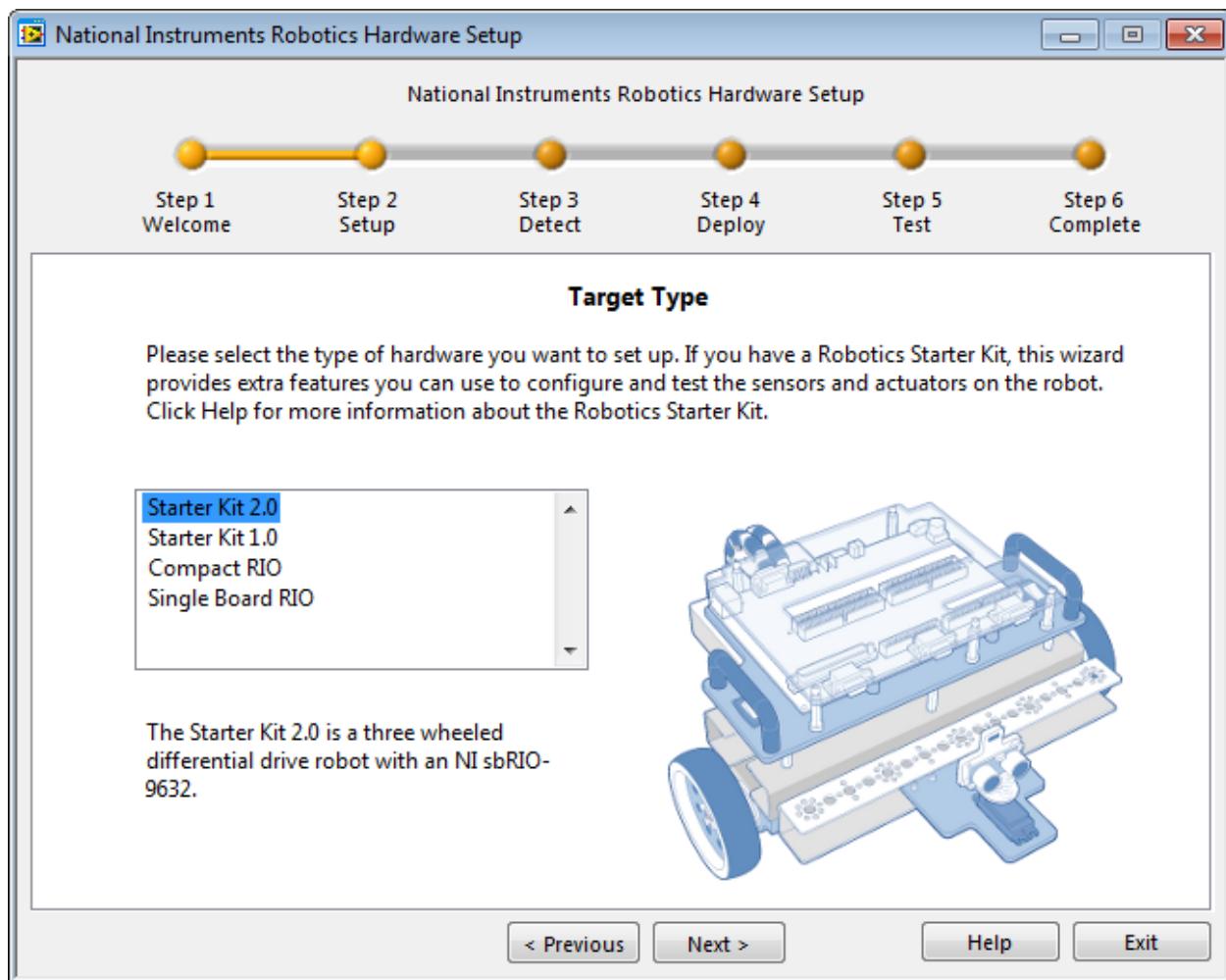
You can also open the Hardware Setup Wizard by selecting Start»All Programs»National Instruments Robotics Hardware Setup, or clicking the Hardware Setup Wizard link on the Getting Started page of the Getting Started window in LabVIEW.

The Hardware Setup Wizard window shown in Figure 0-9 opens and leads you through several steps to establish communication between the host computer and DaNI. Read the information in the window and click the Next> button.



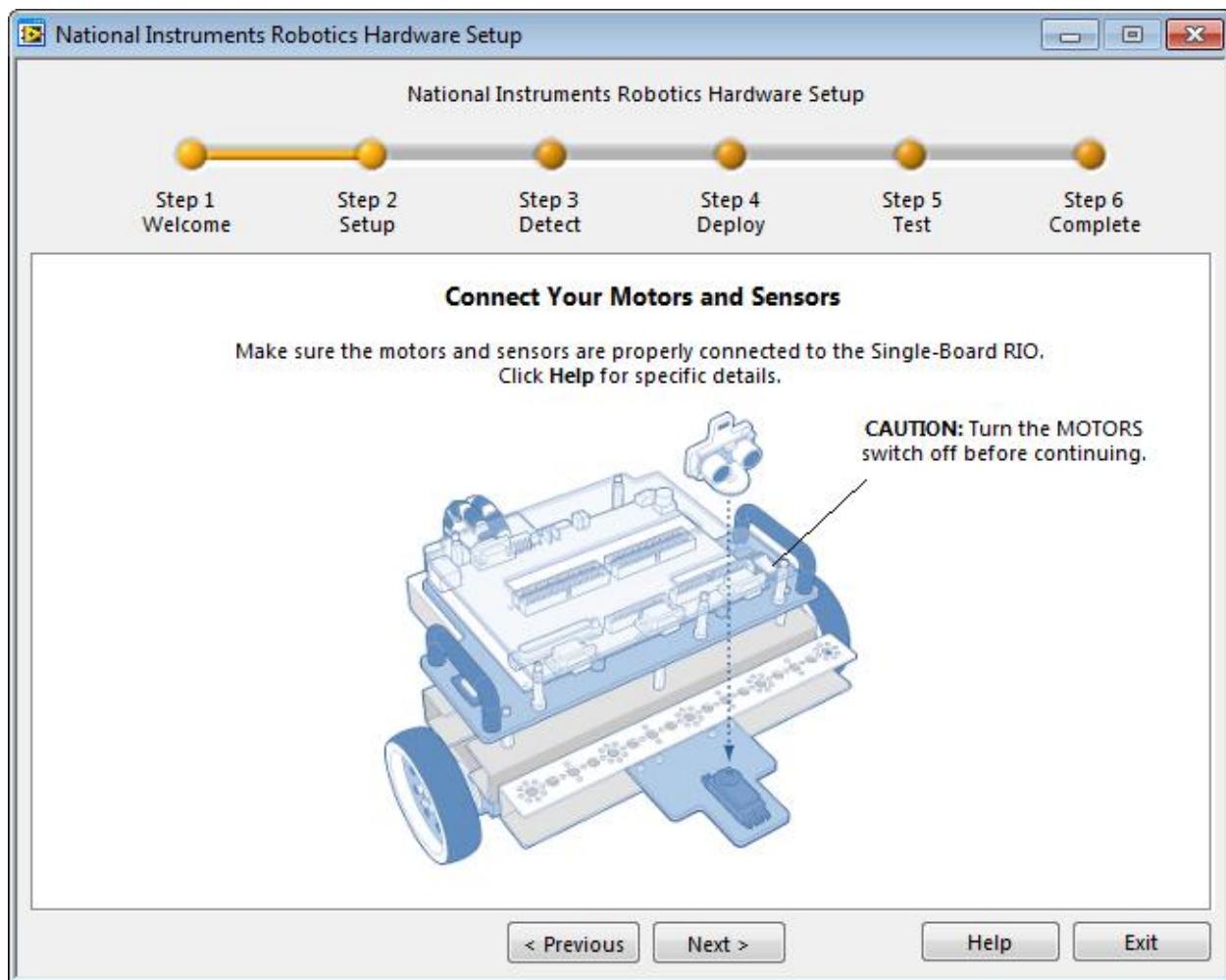
**Figure 0-9. Hardware Setup Wizard Window**

The window shown in Figure 0-10 opens giving you the option of setting up hardware for different types of targets (hardware used in robotic systems). Choose the Starter Kit 2.0 target type and click the Next > button.



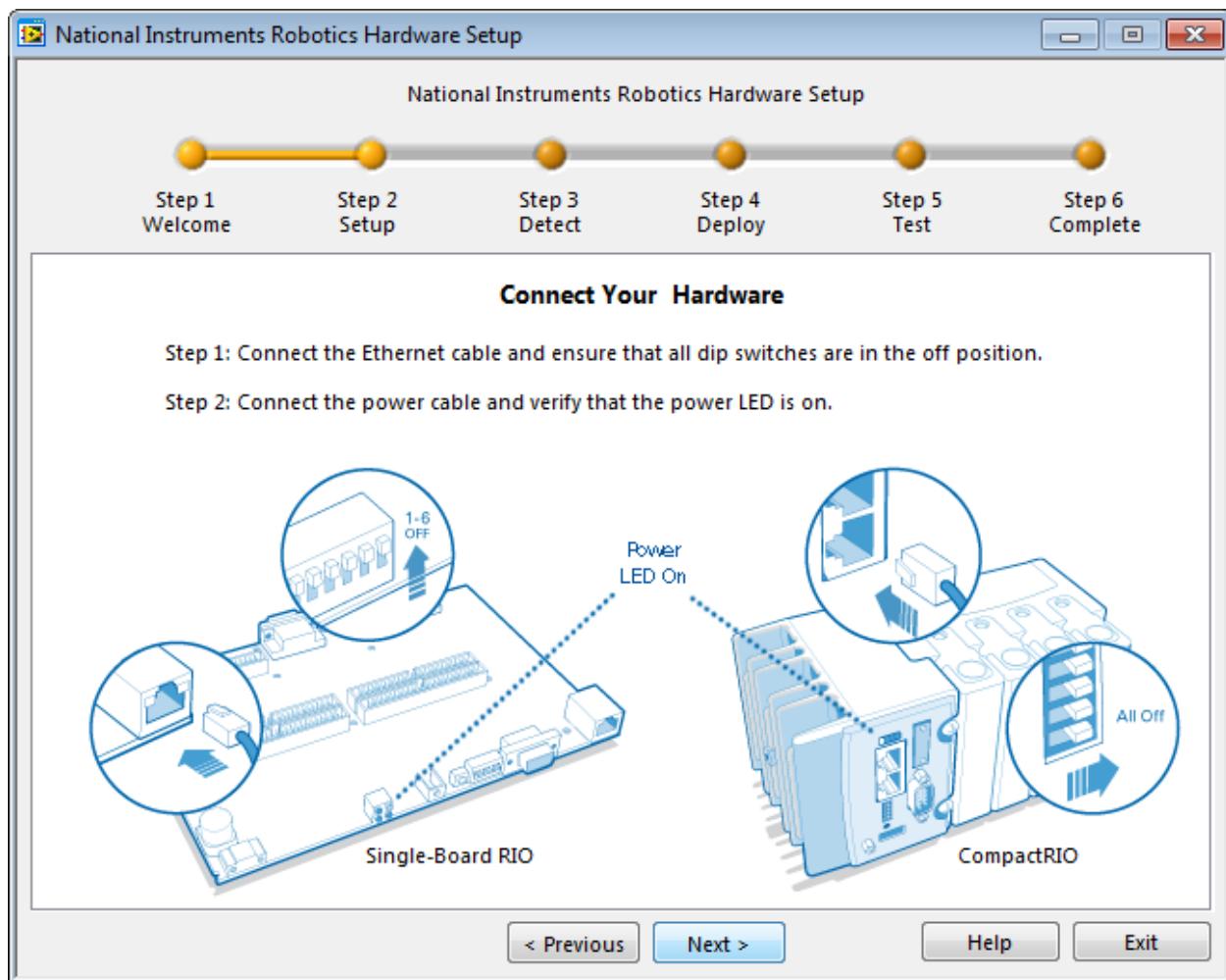
**Figure 0-10. Target type selection window**

The next, Figure 0-11, window reminds you to connect the motors and sensors to the robot and the robot computer before proceeding, and to make sure that the motors switch is in the off position. If the motors and sensors are connected to DaNI, click the Next> button.



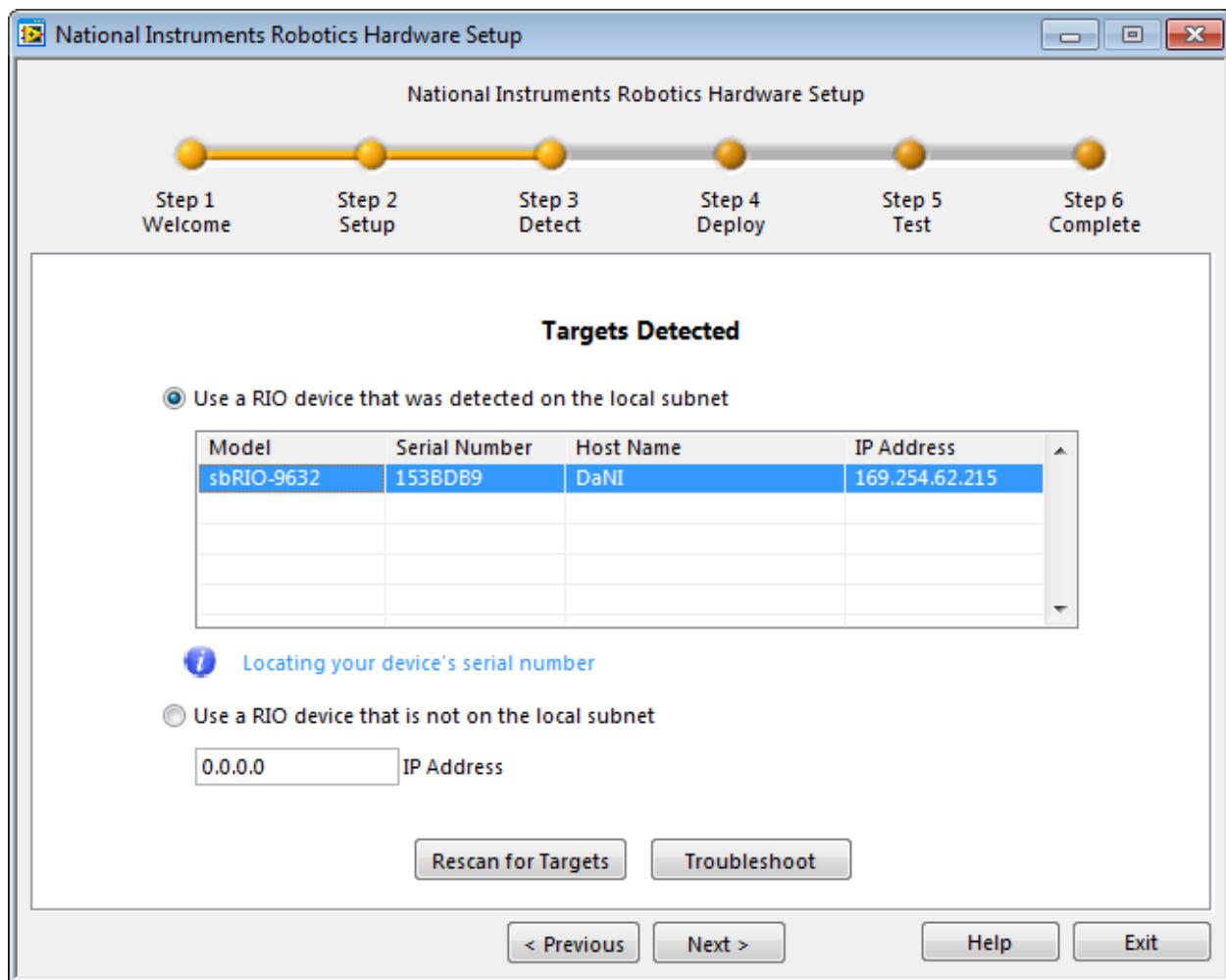
**Figure 0-11. Connect Motors and Sensors window**

The next window, Figure 0-12, gives you two steps, Ethernet and power connections. Connect the Ethernet crossover cable packaged with the kit between the sbRIO on DaNI and the host computer. The power is already connected, but make sure the battery has been charged and the Master power switch is in the ON position. The DIP switches and LEDs will be explained later, just check that the DIP switches are all in the OFF position and only one LED is lit. You can ignore the CompactRIO portion of the window.



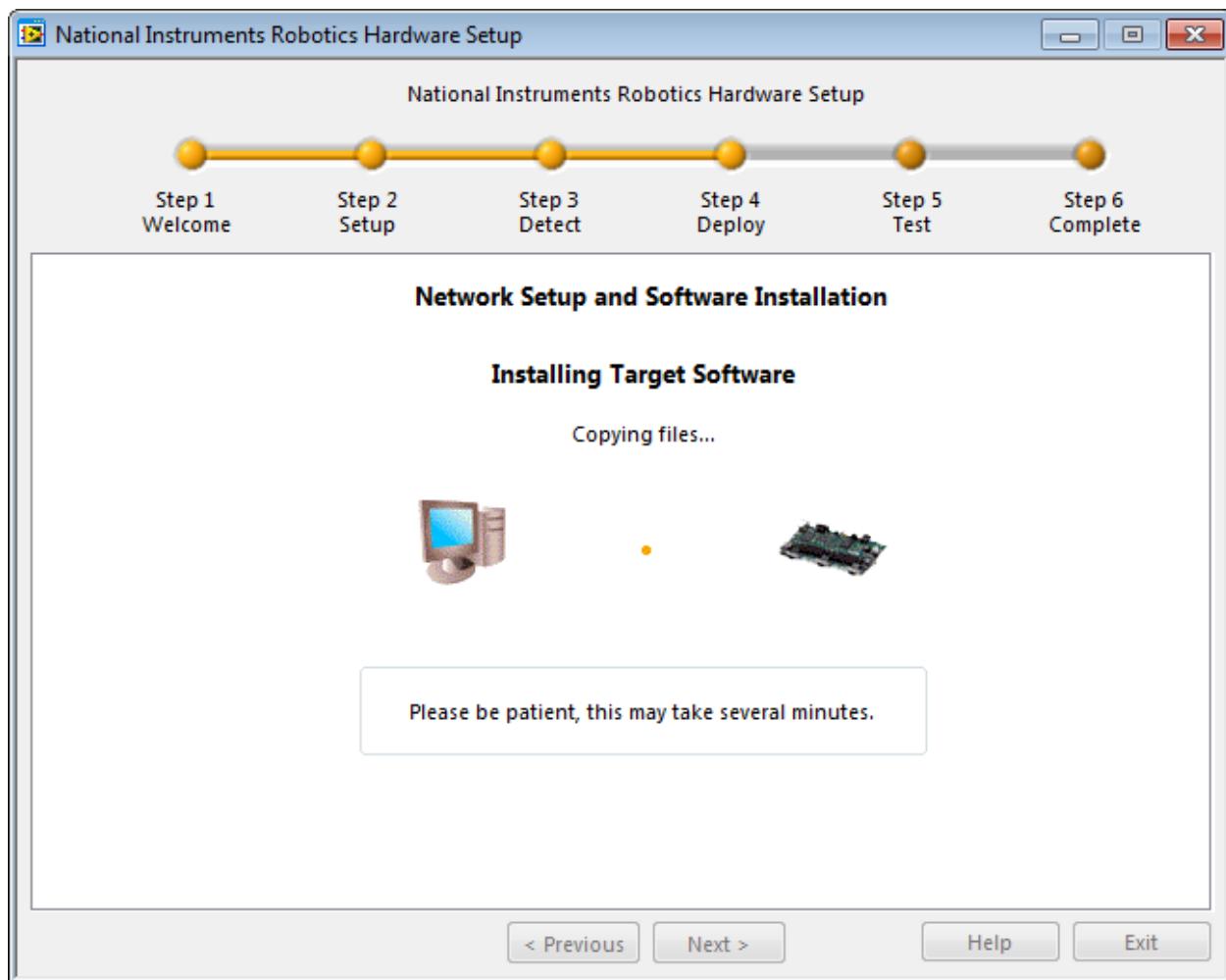
**Figure 0-12. Connect Ethernet and power**

After clicking the Next> button, it may take a few seconds for your target to be recognized. Once your hardware is detected and an IP address assigned as shown in Figure 0-13, verify that you have selected the device with the correct serial number. The serial is located on a small green sticker at the bottom right corner of the Single-Board RIO. Write down the IP address assigned to DaNI.



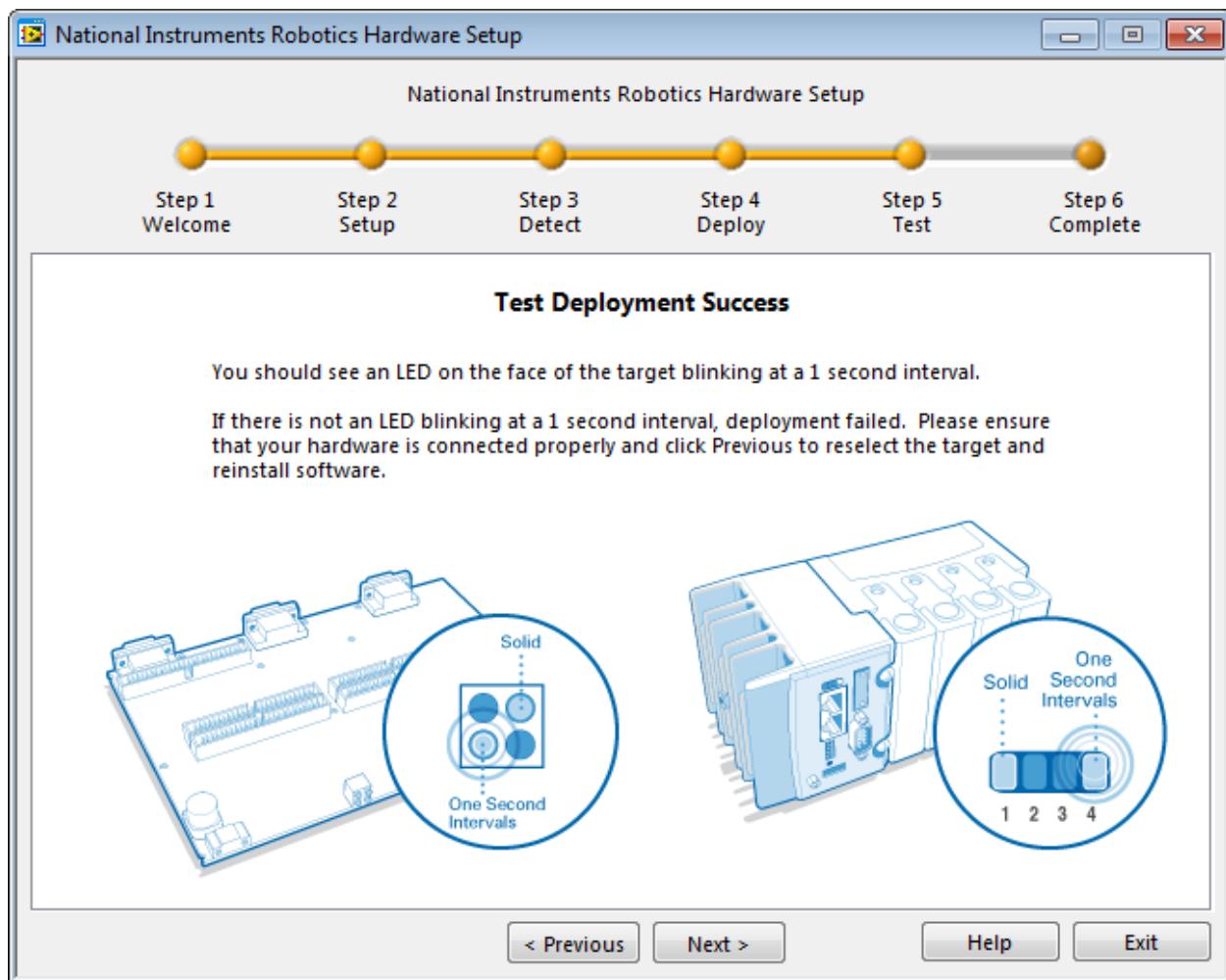
**Figure 0-13. Targets Detected window**

Since the system is able to support multiple CompactRIO or Single-Board RIO targets connected to the same subnet as the host computer, the Hardware Setup Wizard returns the model type and serial number of each available target on the Detecting Hardware page. This requires you to select the device with a mouse click so it is highlighted as shown in Figure 0-13. The Next> button will be disabled and greyed out until you click on the target. Click Next> to continue with the wizard, allowing it to install software and copy files over to the Single-Board RIO target. This will take a few minutes and the windows shown in Figure 0-14 will be displayed.



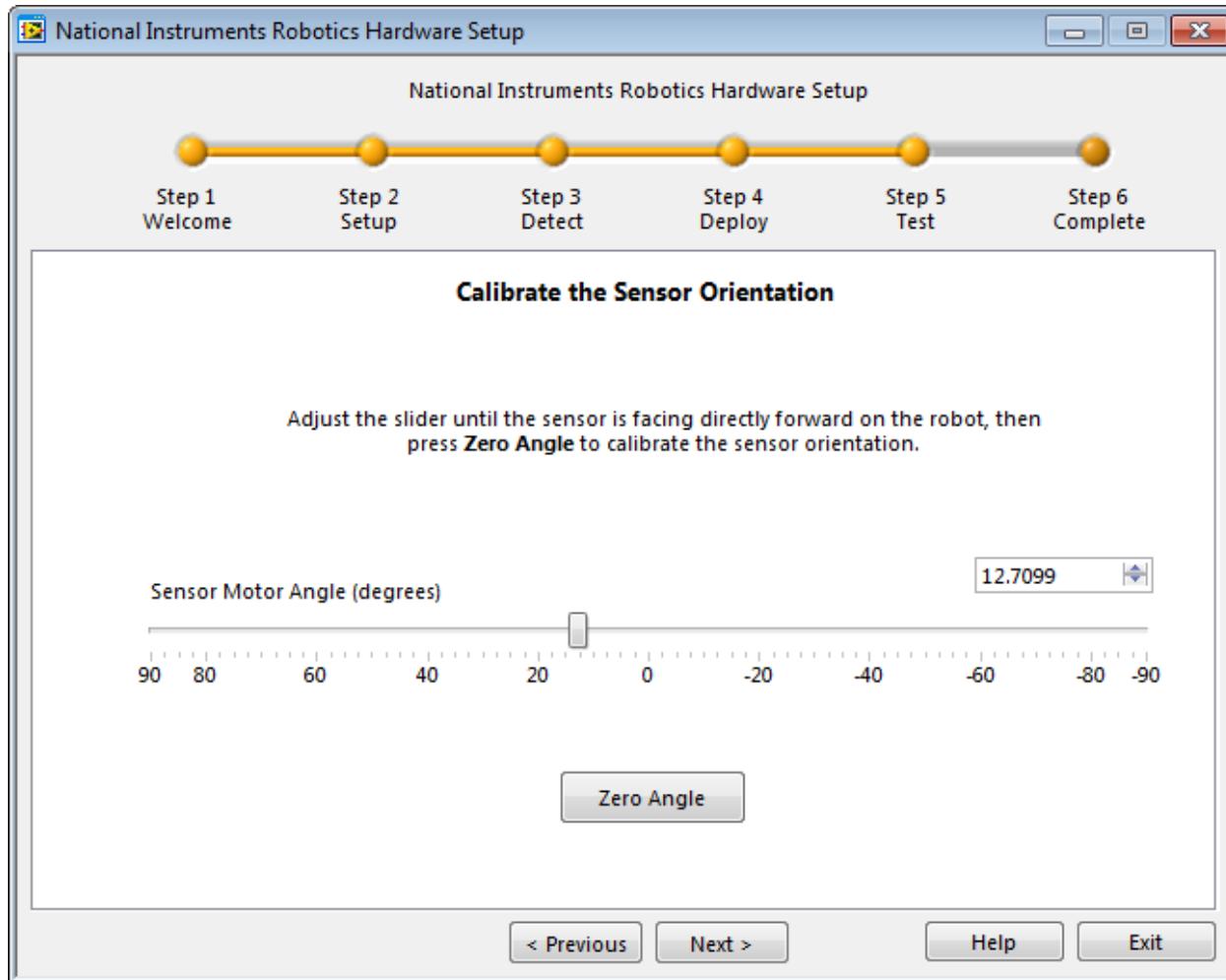
**Figure 0-14. Target Software Installation window**

After the software is installed, the Figure 0-15 window is displayed showing successful deployment. If the deployment was unsuccessful, there is some information for troubleshooting. Subsequent sections of this experiment contain more detailed information about what this Wizard does that will provide fundamental knowledge for troubleshooting and understanding the configuration process.



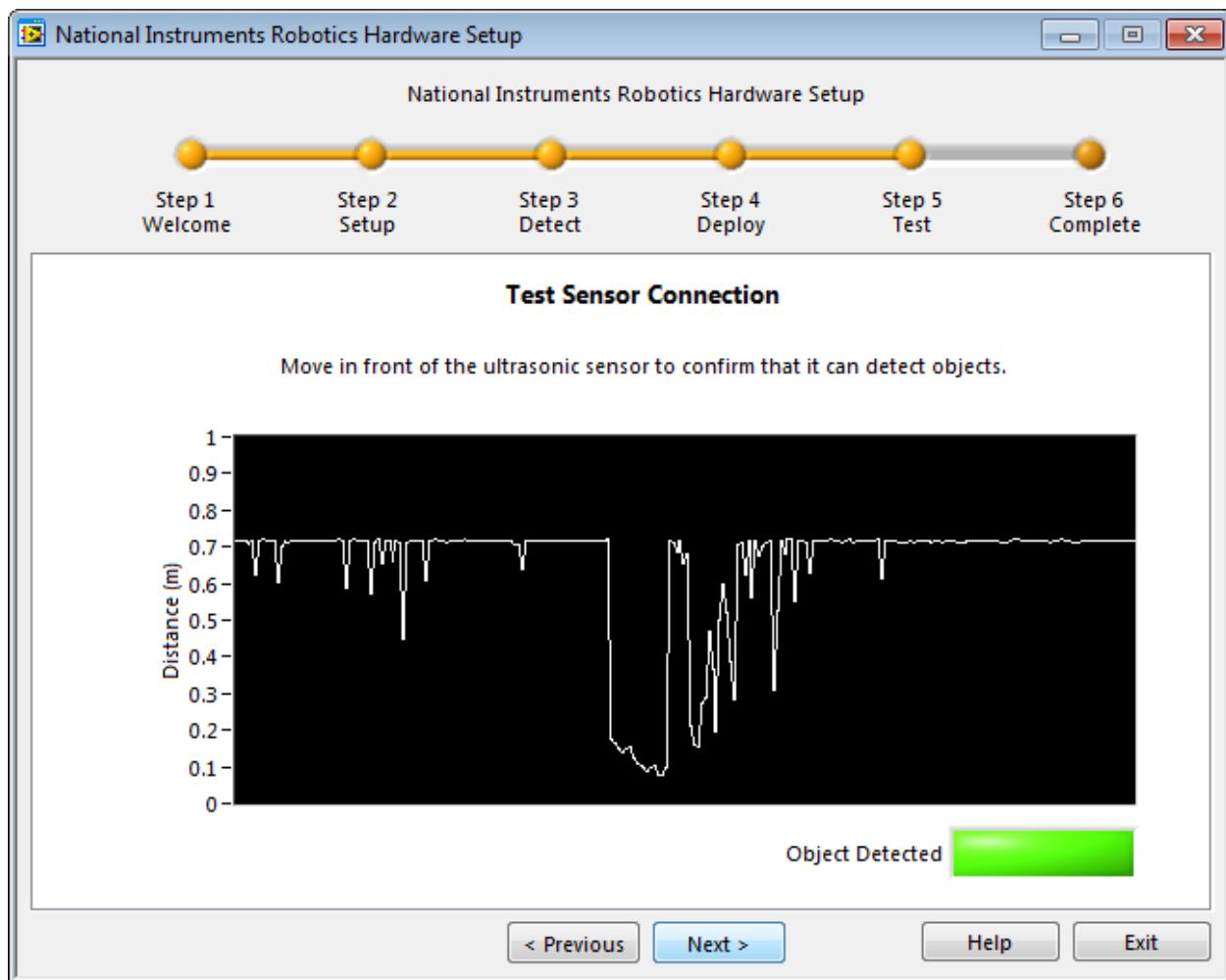
**Figure 0-15. Test Deployment Success window**

After the software is deployed to DaNI, you can calibrate the orientation and test the PING))) ultrasonic transducer. First, orient the transducer by adjusting the slider shown in Figure 0-16. There are no markings to make sure you have the sensor oriented directly forward, but try to get it as close as possible. Record the angle and click the Next> button. This angle will be written to a file on the sbRIO that initialization software will read. You will learn how to write your own orientation calibration program and write information to file on the sbRIO later in this set of experiments.



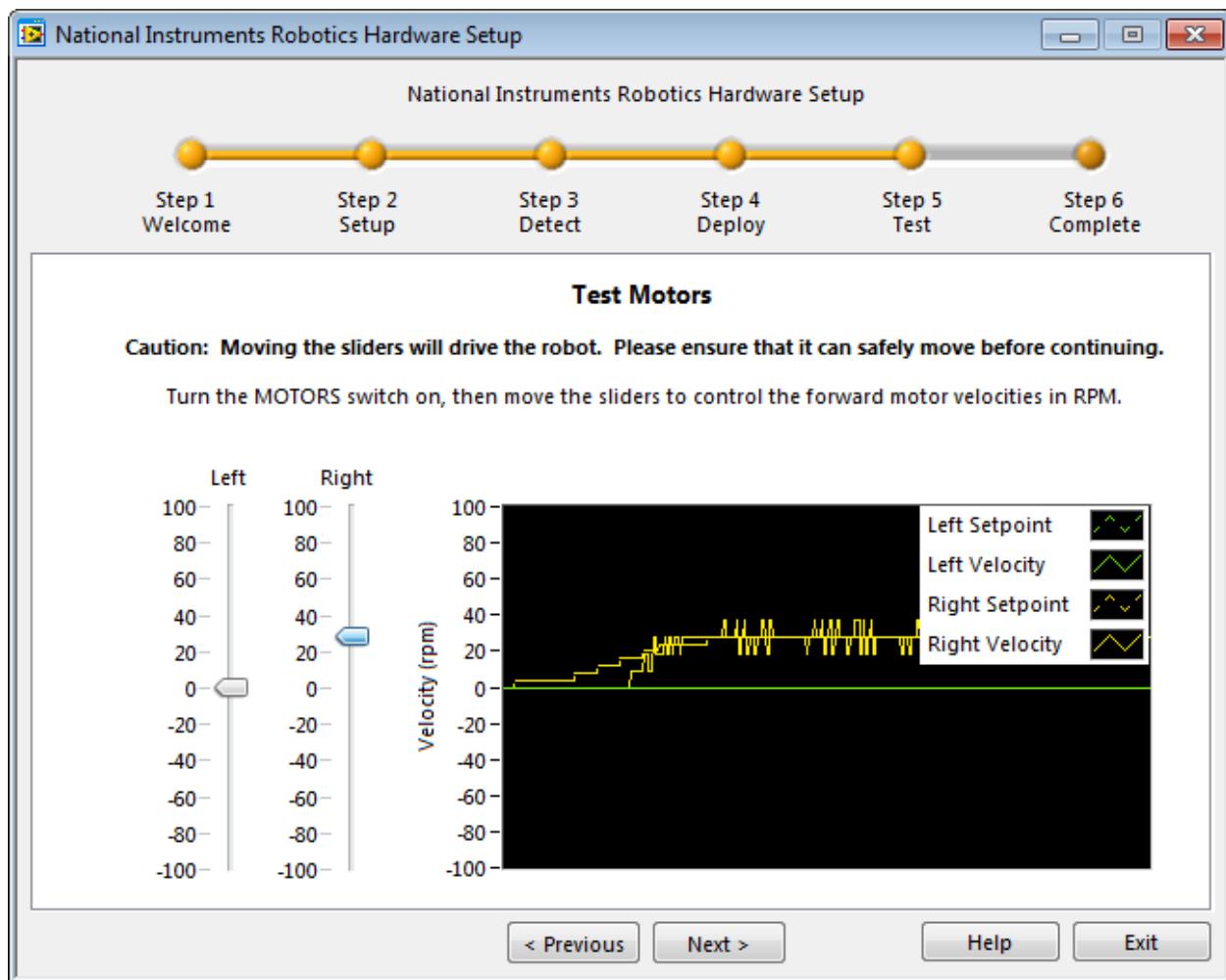
**Figure 0-16. Calibrate Sensor Orientation**

After you click **Next >**, a graph showing the distance (range) signal from PING))) is displayed as shown in Figure 0-17. Make sure that the sensor is functioning properly by locating DaNI orthogonal to a flat surface that will reflect sound, like a wall or the side of a box, at a known distance (in the 0.5 to 3m range) in front of the sensor. Wait for a few minutes after placing an object in the field of view of the ultrasonic transducer to observe the correct distance as previous distance signals may still be in memory.



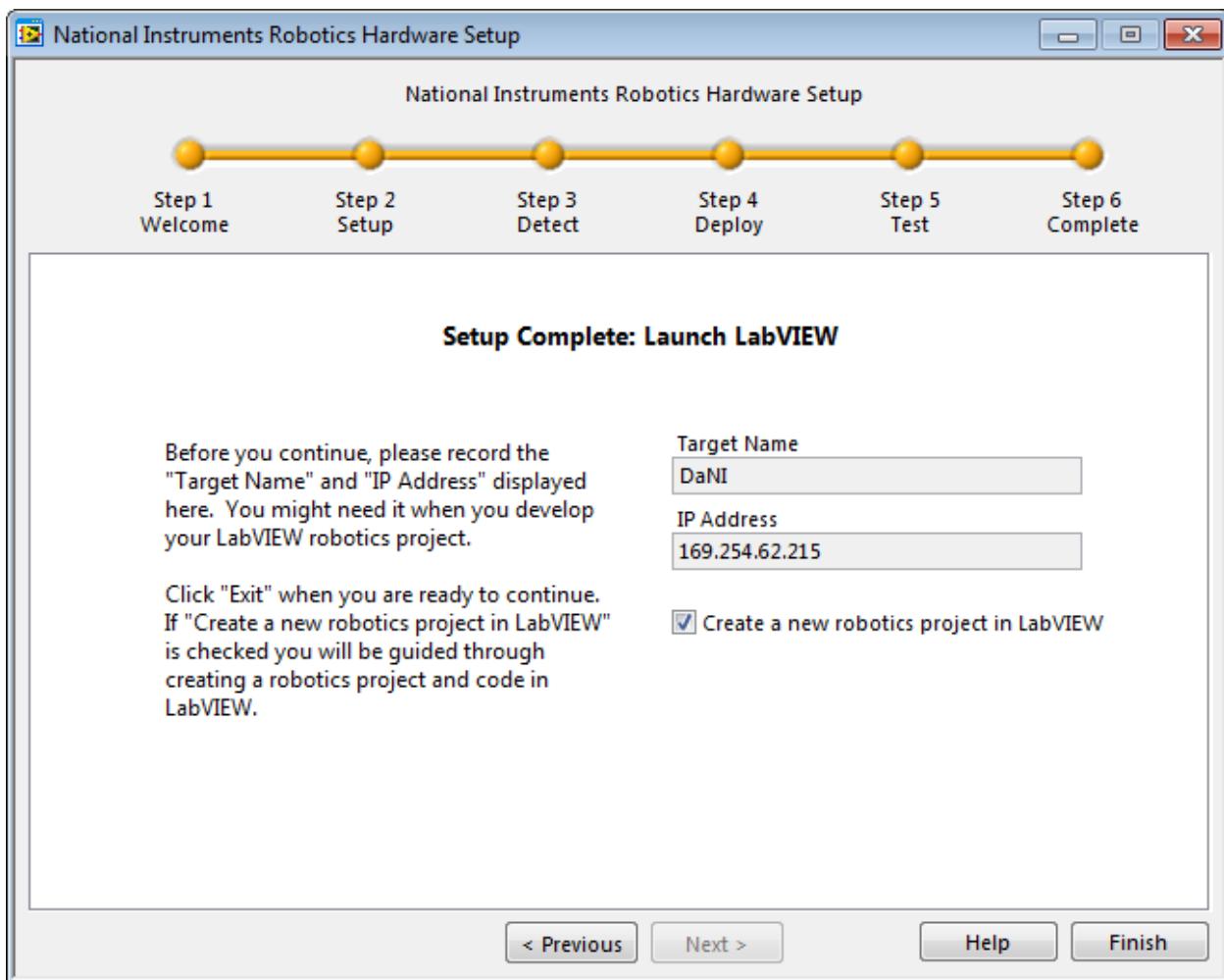
**Figure 0-17. Test Sensor Connection window**

After testing PING))) click Next> to test the motors and encoders. Place DaNI in an area where it can move safely, set the Motors switch to ON, and operate the sliders shown in Figure 0-18. After completing the motors test, click Next>.



**Figure 0-18. Test Motors window**

This completes the hardware setup and the window shown in Figure 0-19, is displayed. Check the “Create a new robotics project in LabVIEW” box and click the Finish button (there is no Exit button as stated in the instructions in the window).



**Figure 0-19. Step 6 window showing successful completion of the hardware setup**

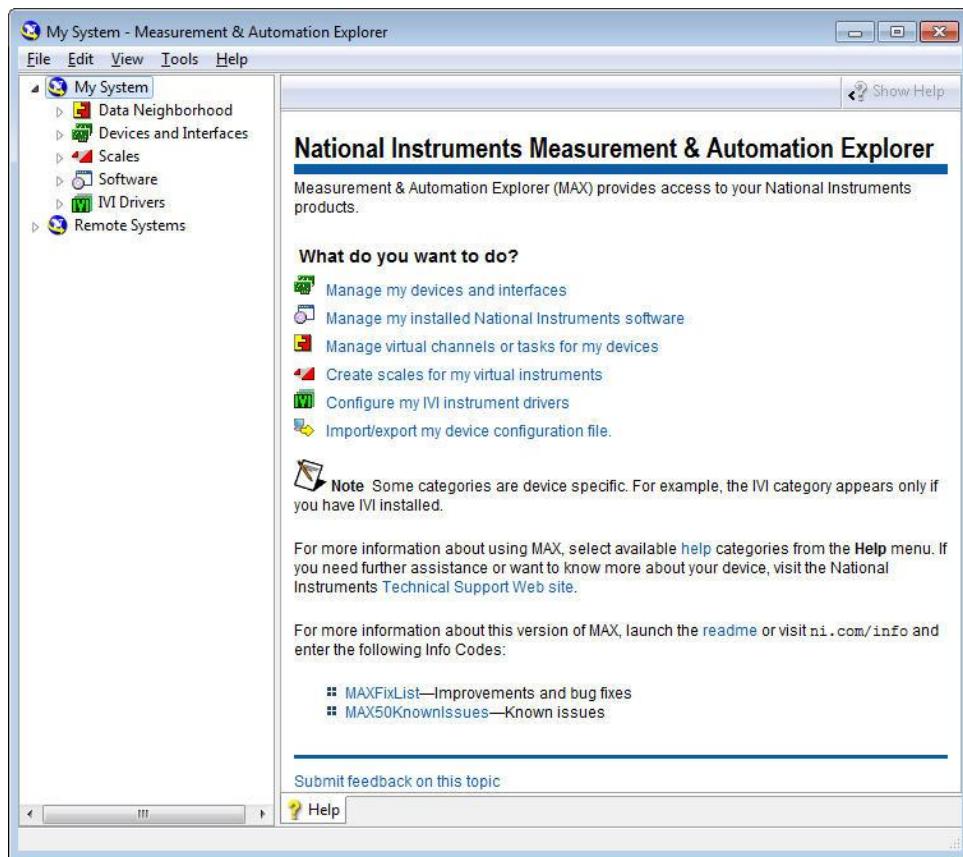
After you click Finish with a checkmark in the Create a new robotics project in LabVIEW checkbox, the Wizard will exit the Hardware Setup Wizard.

The following describes how to do the hardware setup without the Hardware Wizard. Even if you used the wizard, it is instructional to study the process so you understand what the wizard did. But if desired, you can go directly to the section on Creating a Project without the Hardware Wizard to create a LabVIEW Robotics Project.

## Establish Communications between DaNI and a host without the Hardware Wizard

If the Hardware Wizard isn't available or if your network uses static IP addresses, use the National Instruments Measurement and Automation Explorer (MAX) shown in Figure 0-20

(Start>>All Programs>>National Instruments>>Measurement and Automation) to configure the host to DaNI communications. If the host computer is already configured on a network, you must configure communications with DaNI on the same network. As shown in Figure 0-5 you can communicate either with at crossover directly to the host or a CAT5 cable to a hub or switch. If neither machine is connected to a network, you must connect the two machines directly using a CAT-5 crossover cable or hub. You can use the direct connection to configure DaNI from the host computer.



**Figure 0-20. Measurement and Automation Explorer (MAX)**

With MAX, you can:

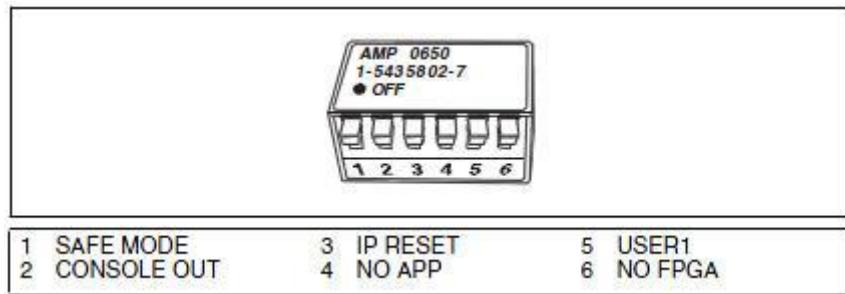
- Configure your National Instruments hardware and software
- Create and edit channels, tasks, interfaces, scales, and virtual instruments
- Execute system diagnostics
- View devices and instruments connected to your system
- Update your National Instruments software

In addition to the standard tools, MAX can expose item-specific tools you can use to configure, diagnose, or test your system, depending on which NI products you install. As you navigate

through MAX, the contents of the application menu and toolbar change to reflect these new tools.

To configure communications with the sbRIO, expand Remote Systems in the Measurement & Automation (MAX) configuration tree. Previously detected remote systems are already shown. MAX will add newly detected systems after a short delay. MAX automatically searches for new remote systems every time you launch MAX and expand Remote Systems.

If a previous user assigned an IP address to DaNI, you may need to change it so it is compatible with your network. You can reset the IP with the DIP switches identified in Figure 0-21. See Figure 0-3 to locate the switches on the sbRIO.



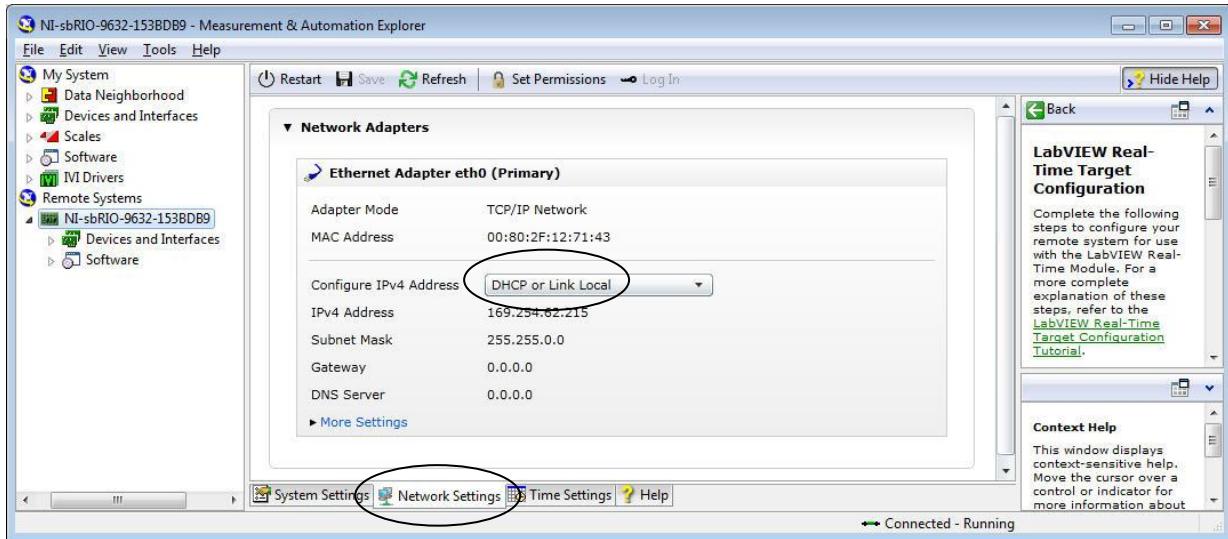
**Figure 0-21. sbRIO DIP switches**

If the safe mode switch is in the ON position at startup, the sbRIO launches only the essential services required for updating its configuration and installing software. The LabVIEW Real-Time engine does not launch. If the switch is in the OFF position, the LabVIEW Real-Time engine launches. Keep this switch in the OFF position during normal operation. The SAFE MODE switch must be in the ON position to reformat the drive on the device.

Set the Safe mode switch and the IP RESET switch to the ON position and push the sbRIO reset switch shown in Figure 0-3 to reset the IP address to 0.0.0.0. The status LED will blink 3 times in succession continuously to indicate that the sbRIO is in safe mode.

You can now configure a new IP address that matches your network configuration. You can set a DHCP or static IP. You can set either DHCP or static to communicate over the crossover cable or over an ethernet CAT 5 cable to a network hub or switch as was shown in Figure 0-5.

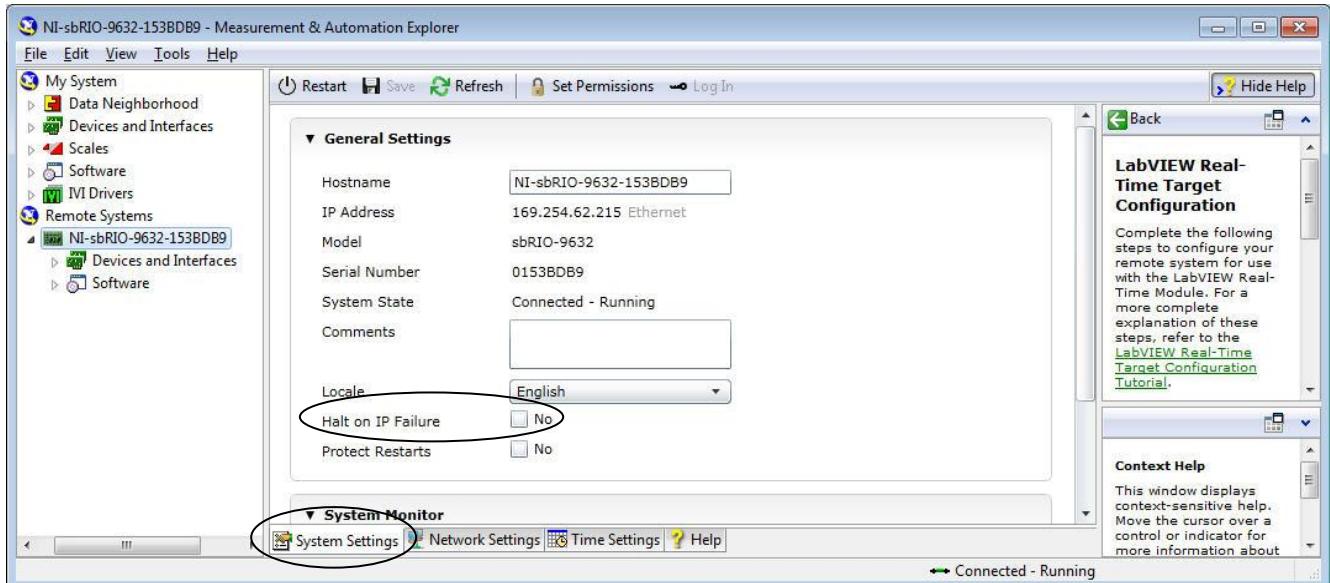
To verify that MAX detects the sbRIO target, expand the Remote Systems tree as shown in Figure 0-22. If no device is listed, refresh the list by clicking Refresh or pressing <F5>.



**Figure 0-22. DaNI sbRIO DHCP network settings information in MAX**

Configure the DaNI sbRIO target to use a Link Local IP address as shown in Figure 0-22. Select the sbRIO target and the Network Settings tab. Verify that Configure IPv4 Address is set to DHCP or Link Local. Set the Configure IPv4 Address to DHCP or Link Local.

Select the System Settings tab as shown in Figure 0-23. Uncheck the Halt on IP Failure checkbox is disabled. When the CompactRIO is configured to use a DHCP or link local IP address and the Halt on IP Failure checkbox is disabled, the sbRIO will use a link local address if it does not find a DHCP server, which it shouldn't since you are connected directly to the host using a cross-over cable.



**Figure 0-23. DaNI sbRIO system settings information in MAX**

Set the SAFE MODE and the IP RESET switch on the sbRIO target back to the OFF position. Click the **Save** button in MAX unless it is dimmed. If asked to reboot, choose yes. Press the sbRIO reset button on the CompactRIO target, and wait for the POST to complete. The status LED should turn off and not blink.

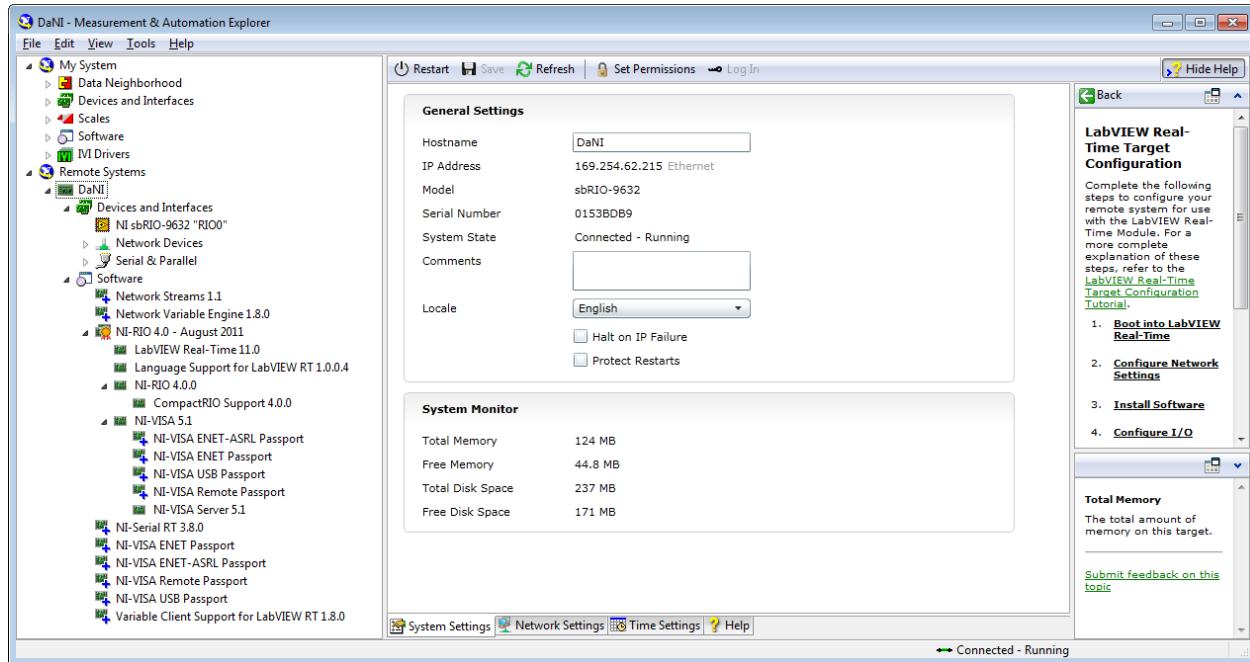
Record the IP address. When an RT target is configured to use a DHCP or link local IP address, the RT target may not always have the same IP address after rebooting. View the Network Settings tab in MAX to determine the current IP address.

You can use the System Settings tab in MAX to assign a different host name. Type in a new name, like DaNI, and click the MAX Save button.

If your system uses static IP addresses instead of DHCP, you can configure a new static IP address for the device in MAX. Similar to the process described above, select the sbRIO target and the Network Settings tab. Set the Configure IPv4 Address to Static instead of DHCP or Link Local. Enter new values for the IP, subnet, gateway and DNS. Base the values on the host computer or use values from your network administrator. To find out the network settings for your host computer, access the TCP/IP properties as described above, or run ipconfig. To run ipconfig, open a command prompt window (start>>search programs and files>>cmd), type ipconfig at the prompt, and press <Enter>. If you need more information, run ipconfig with the /all option by typing ipconfig/all to see all the settings for the computer. Use the first three dotted decimal values of the host for the IP address and set the fourth to a value different from the host if DaNI is connected to the host via a crossover cable and the host is not connected to a network. If DaNI is connected directly to a network via a hub or switch, contact the network administrator to obtain an IP address that isn't used by any other computer on the network. The subnet, DNS, and gateway values should be the same as the values reported for the network from the TCP/IP properties or ipconfig query. Make sure that the SAFE MODE and IP RESET DIP switches are set to OFF before rebooting the controller. Click **Save** and Click **Yes** when MAX prompts you to reboot the target. The sbRIO should now show up in MAX with the static IP that you configured. The status LED should turn off and not blink.

## Add, Remove, or Update Software on DaNI without the Hardware Wizard

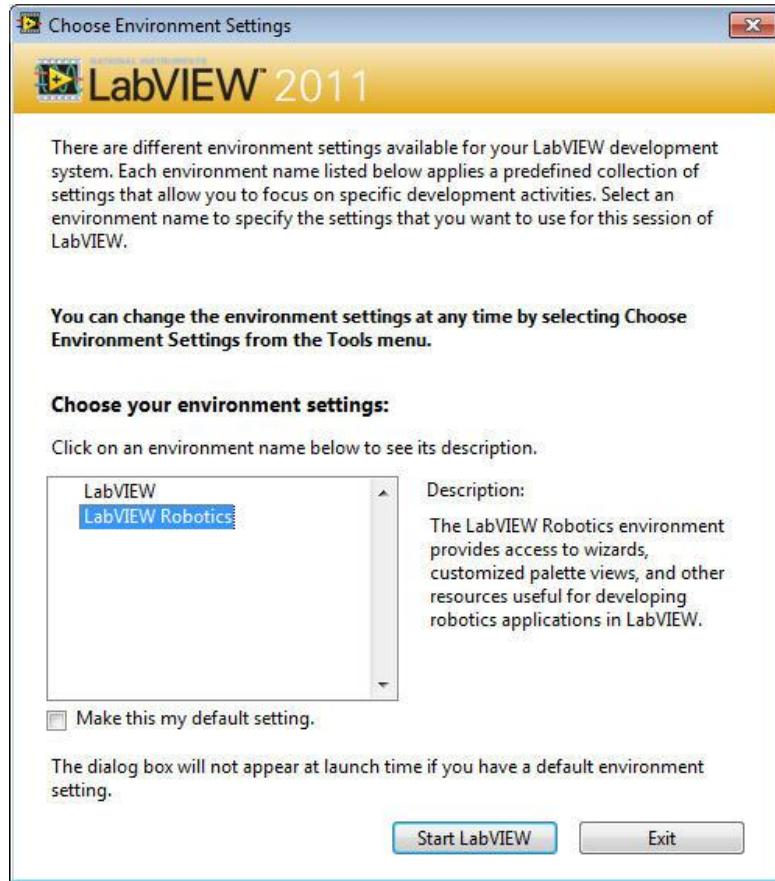
The Hardware Wizard automatically loads the necessary software for DaNI. If you didn't use the wizard, you can add, remove, or update software from MAX. If software has been loaded previously, you can view it by expanding the Software item in the MAX tree as shown in Figure 0-24. If it hasn't, right click the sbRIO item in the MAX tree and choose add software, or click on the sbRIO item and choose the Add/Remove Software button in MAX. This opens the Real Time Software Wizard window and you can choose which software components to download from the host to DaNI. All items were selected for the configuration shown in Figure 0-24.



**Figure 0-24. Software loaded on DaNI from MAX**

## Creating a Project without the Hardware Wizard

If you used the Hardware Wizard, a project was created and opens automatically. Projects are required to communicate between the host and sbRIO. Projects are used to group together both LabVIEW and non-LabVIEW files, create build specifications for executables, and deploy or download files to targets such as NI CompactRIO and NI Single-Board RIO. If you used the Hardware Setup Wizard and the Robotics Project Wizards In the previous segments of this experiment, they automatically create a project for you. If you used MAX without opening LabVIEW in the previous segments of this experiment, to develop a project, open LabVIEW 2011. The choose environment settings window shown in Figure 0-25 opens. Choose the robotics environment. You can make it your default environment if you like.



**Figure 0-25. Choose environment window**

Click the Start LabVIEW button at the bottom of the window and the Getting Started window shown in Figure 0-26 opens. Select Create New Robotics Project. Select the DaNI 2.0 project from the select project type window as shown in Figure 0-27 and click Next. If the host is connected to DaNI with the crossover cable and DaNI is powered on, the project wizard should inherit the IP address correctly from MAX as shown in Figure 0-28 and you can click Next. If not, enter the IP address and click Next. Enter a location on the host computer hard drive where you have write access as shown in Figure 0-29, enter a project name, and click Finish



Figure 0-26. LabVIEW Robotics 2011 Getting Started Window

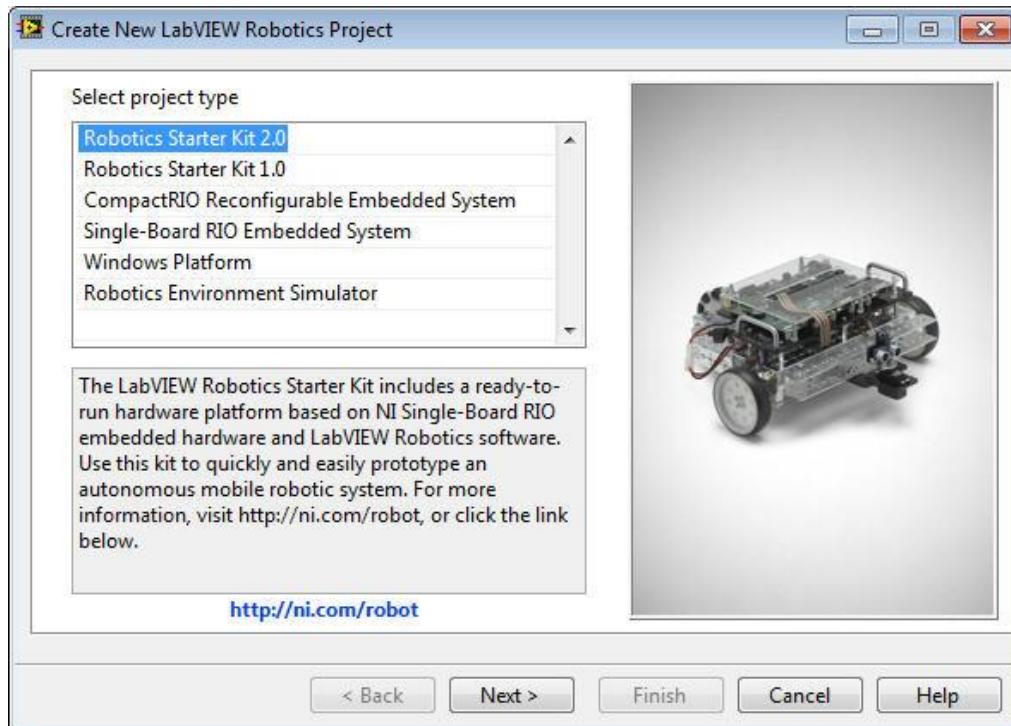
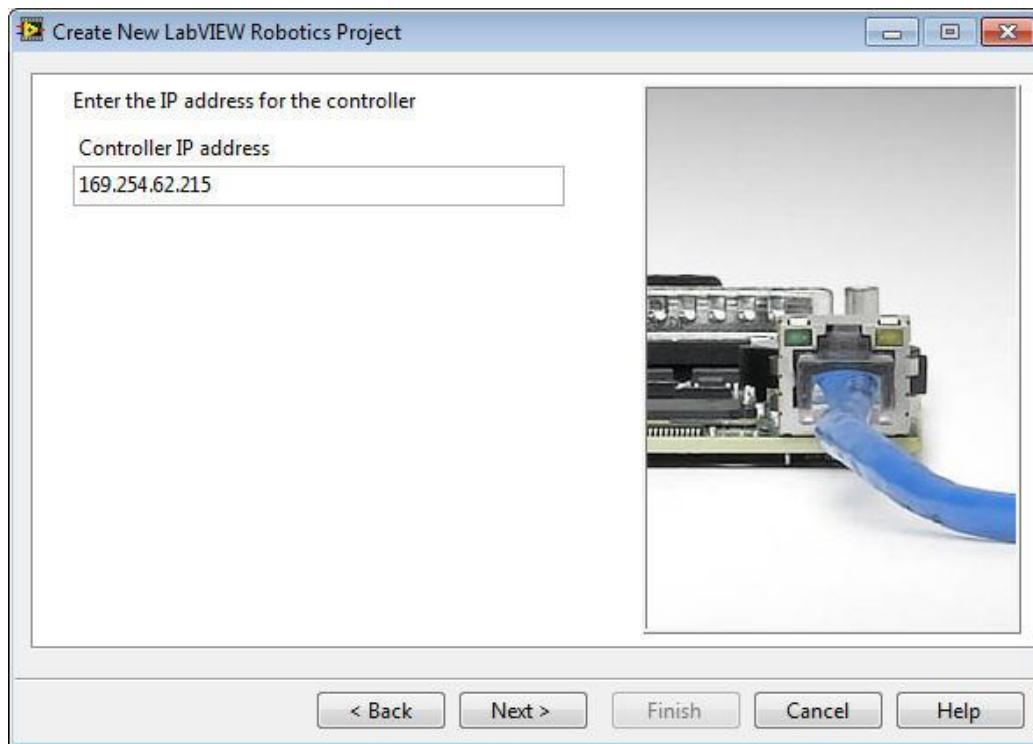
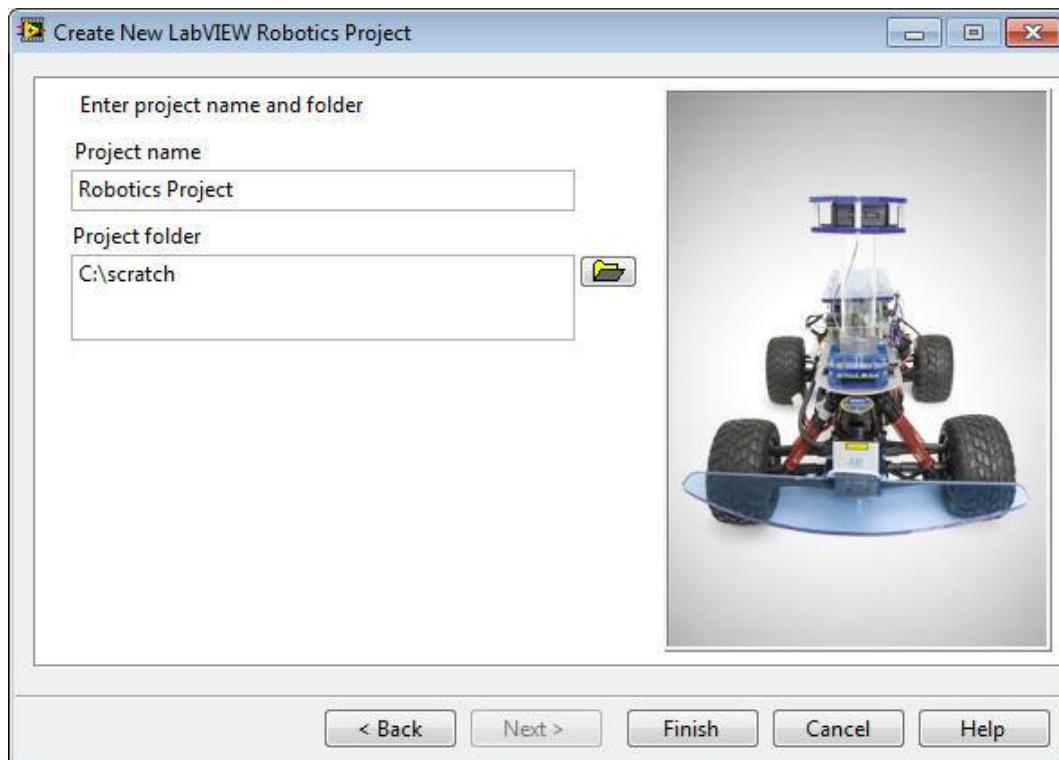


Figure 0-27. Select project type



**Figure 0-28. Enter the IP address**



**Figure 0-29. Project save location**

## Experiment 1-2 DaNI Test

If you used the Hardware Wizard, it will automatically build a project like the one shown in Figure 0-30 for you. If you didn't use the Hardware Wizard, you can use the Project Wizard (explained above) to build the project. To open a project that was closed when you exited LabVIEW, launch LabVIEW and click the project, as shown in Figure 0-31, if it is listed in the getting started window. If it isn't listed, use the Browse button to locate and open it.

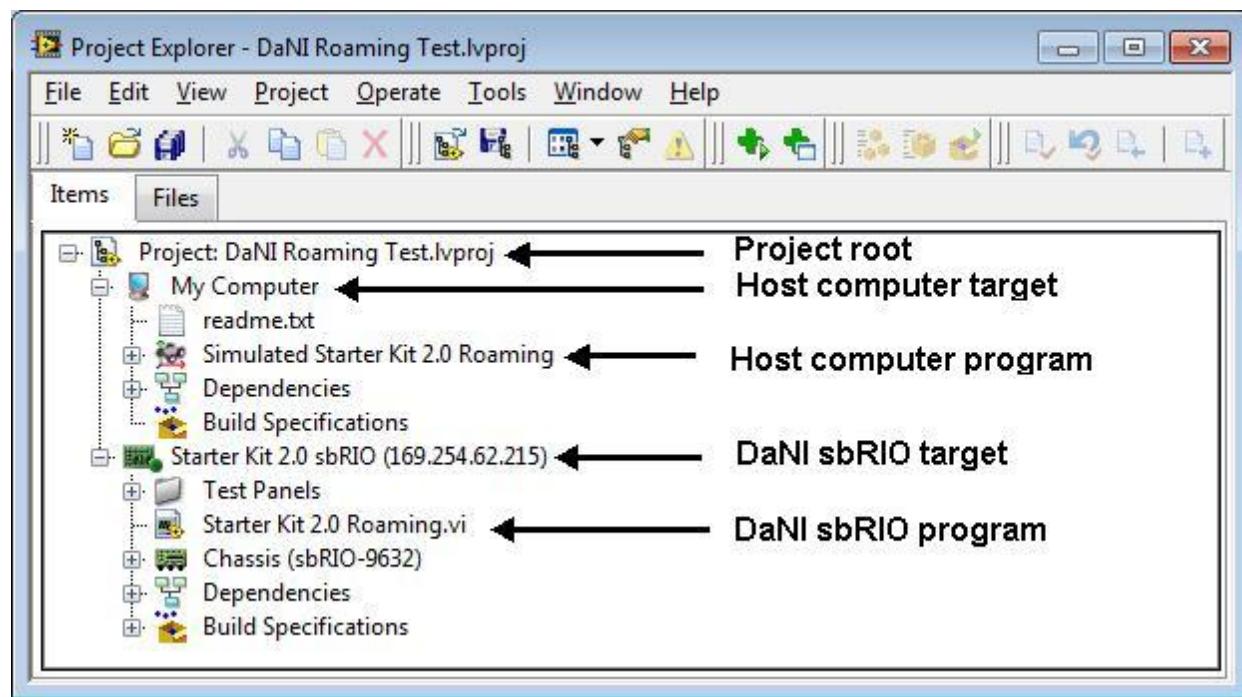


Figure 0-30. Project Explorer Window



**Figure 0-31. Open an existing project from the Getting Started window**

The project shown in the figure includes the host computer, the DaNI sbRIO target, sensor and actuator drivers, and software programs. Once you become more familiar with LabVIEW you will be able to develop projects without the wizard, and consequently will have more control over what is included in a project. National Instruments uses the LabVIEW Project Explorer to facilitate communication between a PC and a remote target (the sbRIO on DaNI). The Project Explorer window includes two pages, the Items page and the Files page. The Items page shown in the figure displays the project items as they exist in the project tree. The Files page displays the project items that have a corresponding file on disk. The Project Explorer window includes the following items by default:

Project root—Contains all the items for the project and displays the file name.

My Computer—Represents the local or host computer as a target in the project.

Dependencies—Includes items that software programs (VIs) under a target require.

Build Specifications—Includes build configurations for source distributions and other types of builds available in LabVIEW toolkits and modules. You can use Build Specifications to configure stand-alone applications, shared libraries, installers, and zip files.

The items in the project are arranged in a tree or hierarchical structure. The first item, "Project:..." is the root. This item shows the name of the file saved on disk with the file

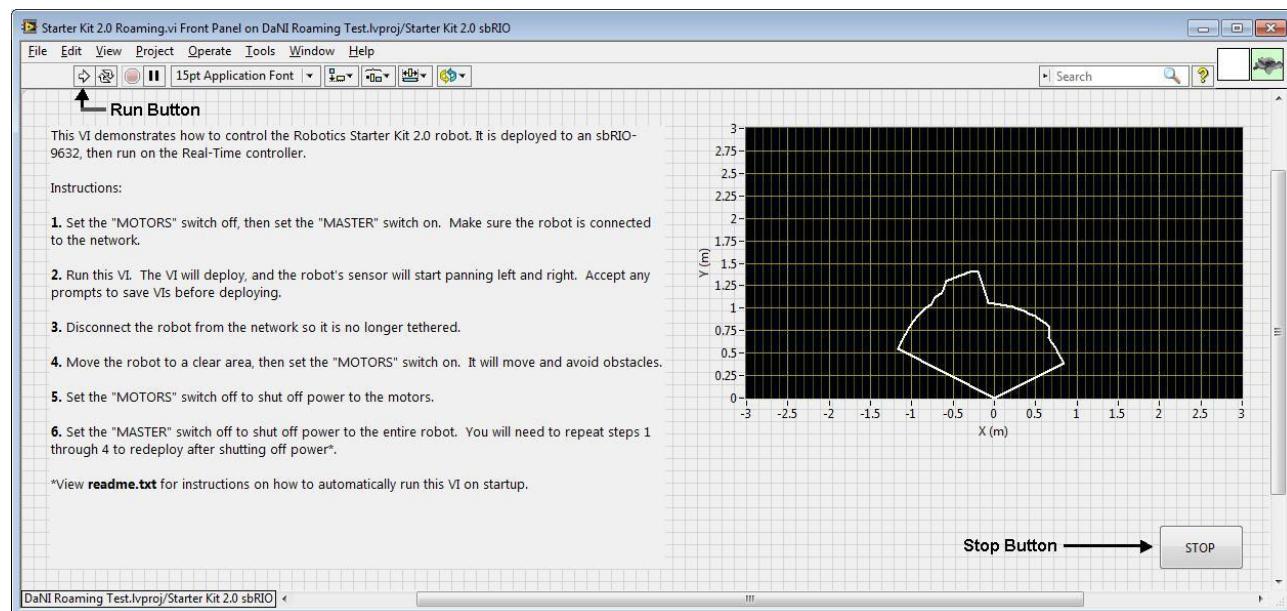
extension lvproj. The second item, My Computer, is indented to show it is lower in the hierarchy. It represents the host computer where programs are developed.

The third and fourth items, Dependencies and Build Specifications, are indented below My Computer indicating that they are lower in the hierarchy and belong to My Computer.

The next item moves up in the heirarchy so its level is equivalent to My Computer. It represents another computer in the project, the sbRIO on DaNI. In addition to the name of the computer, the IP address is displayed. The sbRIO item has dependencies and build specification items like My Computer and some additional items. The Chassis item is part of the sbRIO that connects to and communicates with transducers and actuators. The NI Robotics Starter Kit Utilities.lvlib item are some utility programs that have been written for DaNI to speed the development of programs by allowing users to focus on high-level robotics concepts.

To add a software program to a LabVIEW Project, right-click the hardware target which the program should run on, and select New » VI. If the program is placed under My Computer, it will execute on the host. If the program is placed under the sbRIO target, it will deploy to and execute on the sbRIO.

When you complete the Robotics Project Wizard, LabVIEW opens the Roaming.vi software program on the host computer as shown in Figure 0-32.



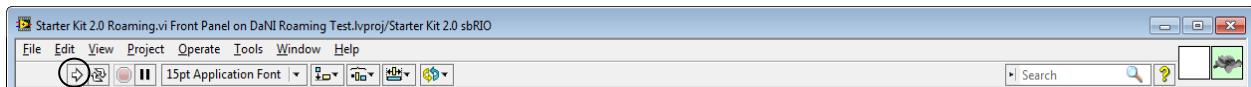
**Figure 0-32. Roaming program user interface**

This program was written for you in LabVIEW. LabVIEW programs are called virtual instruments, or VIs, because their appearance and operation imitate physical instruments, such as oscilloscopes and multimeters. A VI has two windows, a front panel window and a block

diagram window. The front panel shown in Figure 0-32 is the user interface for the VI. The block diagram will be discussed later. The front panel has a graph of distance to obstacles and a stop button. The distance to obstacles graph is from the PING))) ultrasonic transducer that pans +/- 65° while DaNI drives. The point direction angle is relative to the direction of travel, so the graph orientation or reference is as if you were riding on DaNI.

Follow the instructions on the left side of the front panel of the Roaming VI to test the configuration and software. The instructions ask you to run the VI, which you do by clicking the Run button.

Use the Run button on the Front Panel toolbar to execute the VI. The Front panel toolbar is shown in Figure 0-33.



**Figure 0-33. Front Panel toolbar**

The following explains the front panel and block diagram toolbar icons.

-  **Run** runs the VI. LabVIEW compiles the VI, if necessary. You can run a VI if the **Run** button appears as a solid white arrow, shown at left. The solid white arrow also indicates you can use the VI as a subVI if you [create a connector pane](#) for the VI.
-  While the VI runs, the **Run** button appears as shown at left if it is a top-level VI, meaning it has no callers and therefore is not a subVI.
-  If the VI that is running is a subVI, the **Run** button appears as shown at left.
-  The **Run** button appears broken, shown at left, when the VI you are creating or editing contains errors. Click the broken **Run** button to display the [Error list](#) window, which lists all the errors. If the **Run** button still appears broken after you finish wiring the block diagram, the [VI is broken](#) and cannot run.
-  **Run Continuously** runs the VI until you abort or pause execution.
-  **Abort Execution** aborts execution of the top-level VI. If more than one running top-level VI uses the VI, the button is dimmed. You also can use the [Abort VI](#) method to abort the execution of the VI programmatically.
-  **Note** The **Abort Execution** button stops the VI immediately, before the VI finishes the current iteration. Aborting a VI that uses external resources, such as external hardware, might leave the resources in an unknown state by not resetting or releasing them properly. Design VIs with a stop button to avoid this problem.
-  **Pause** pauses or resumes execution. When you click the **Pause** button, LabVIEW highlights on the block diagram the location where you paused execution. Click it again to continue running the VI. The **Pause** button appears red when execution is paused.
-  **Highlight Execution** displays an animation of the block diagram execution when you click the **Run** button. If the **Highlight Execution** button appears yellow, [execution highlighting](#) is enabled.
-  **Retain Wire Values** saves data values. When you click the [Retain Wire Values](#) button, LabVIEW saves the values at each point in the flow of execution so that when you place a probe on a wire, you can immediately obtain the most recent value of the data that passed through the wire. This debugging tool can impact the performance of your VI.
-  **Step Into** opens a node and pauses. When you click the **Step Into** button again, it executes the first action and pauses at the next action of the subVI or structure.
-  **Step Over** executes a node and pauses at the next node.
-  **Step Out** finishes executing the current node and pauses. When the VI finishes executing, the **Step Out** button becomes dimmed.

15pt Application Font |  **Text Settings** [changes the font settings](#) for the VI.

-  **Note** When a VI stops at a [breakpoint](#), the [Call list](#) pull-down menu appears in the location of **Text Settings** if other VIs call the stopped VI. You can select a VI from the [Call list](#) pull-down menu to go to the block diagram of that VI.
-  **Align Objects** [aligns objects](#) along axes.
-  **Distribute Objects** [spaces objects](#) evenly.
-  **Resize Objects** [resizes multiple front panel objects](#) to the same size.
-  **Reorder** [moves objects](#) relative to each other. Select the **Reorder** pull-down menu when you have objects that overlap each other and you want to define which one is in front or back of another.
-  **Clean Up Diagram** [reroutes all existing wires and rearrange all existing objects](#) on the block diagram automatically.
-  **Show Context Help Window** [displays the Context Help window](#).
-  **Enter** appears to remind you that a new value is available to replace an old value. The **Enter** button disappears when you click it, press the <Enter> key, or click the front panel or block diagram workspace.
-  **Warning** appears if a VI includes a warning and you placed a checkmark in the [Show Warnings](#) checkbox in the [Error list](#) window.



**Synchronize with Other Application Instances** applies changes to the VI in all application instances. You cannot undo changes made to the VI after you click this button. This button is available only if you [edit a VI that is open in multiple application instances](#).

Before clicking the Run button, review the LabVIEW Robotics Starter Kit Safety Guide by navigating to the LabVIEW\readme directory on the DVD that is packaged with the kit and opening StarterKit\_Safety\_Guide.pdf. Remember that DaNI is expensive so be very careful not to damage it.

When you click the Run button, the software that was automatically developed by the Wizards will be deployed to the DaNI sbRIO and the Deployment Progress Window shown in Figure 0-34 will be displayed.

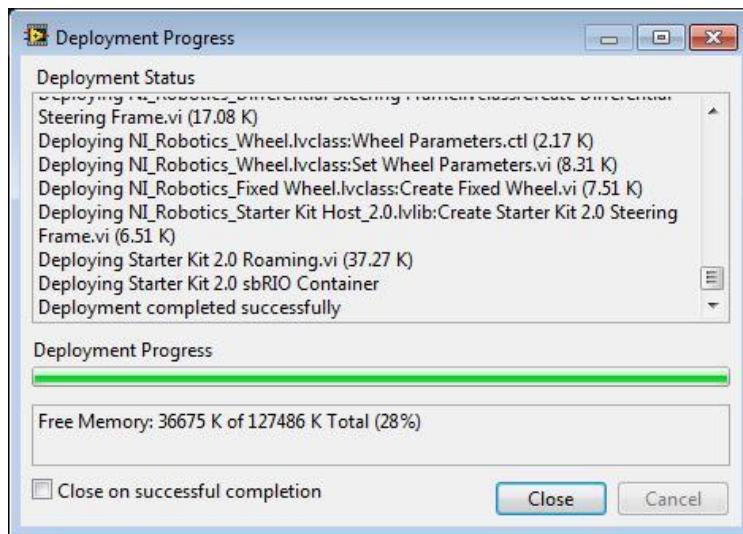


Figure 0-34. Deployment progress window

When you disconnect the network cable to allow DaNI to roam untethered, the program running on the host computer will display a series of messages like the one shown in Figure 0-35. When you click the OK button, the application on the host will terminate. That is okay since DaNI no longer needs this program. Whether the program on the host runs or not doesn't affect the operation of DaNI after the network cable has been disconnected.

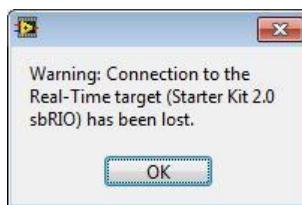


Figure 0-35. Lost connection message

If the software deploys and DaNI functions with and without the network tether, it has successfully tested and you can proceed to the next section of the experiment. Save the project in a folder on your computer where you have write access and you can open it in the future without using the robotics project wizard.

## Experiment 1-3 Evaluate the Operation of an Autonomous Mobile Robot

Read completely through this section of the experiment before you start. Study the material in a robotics text on mobility and autonomy, like Chapters 1 and 2 in Siegwart et al (2011) to integrate fundamental concepts with the results from this experiment. Set up an indoor area with a flat floor for the robot to operate in. Set up the area to gather information to answer the questions below. Avoid stairs and any other drop offs where DaNI might drive over the edge, fall, and be damaged. Draw a plan view or make a map of the environment to scale of the area. Figure 0-36 shows an example. If it is in a room, draw the walls, doors, furniture and other obstacles. Draw at the level of what the robot “sees.” That is, the robot will see the legs of a chair or table, not the entire piece of furniture. You can draw it in a CAD program or on paper.

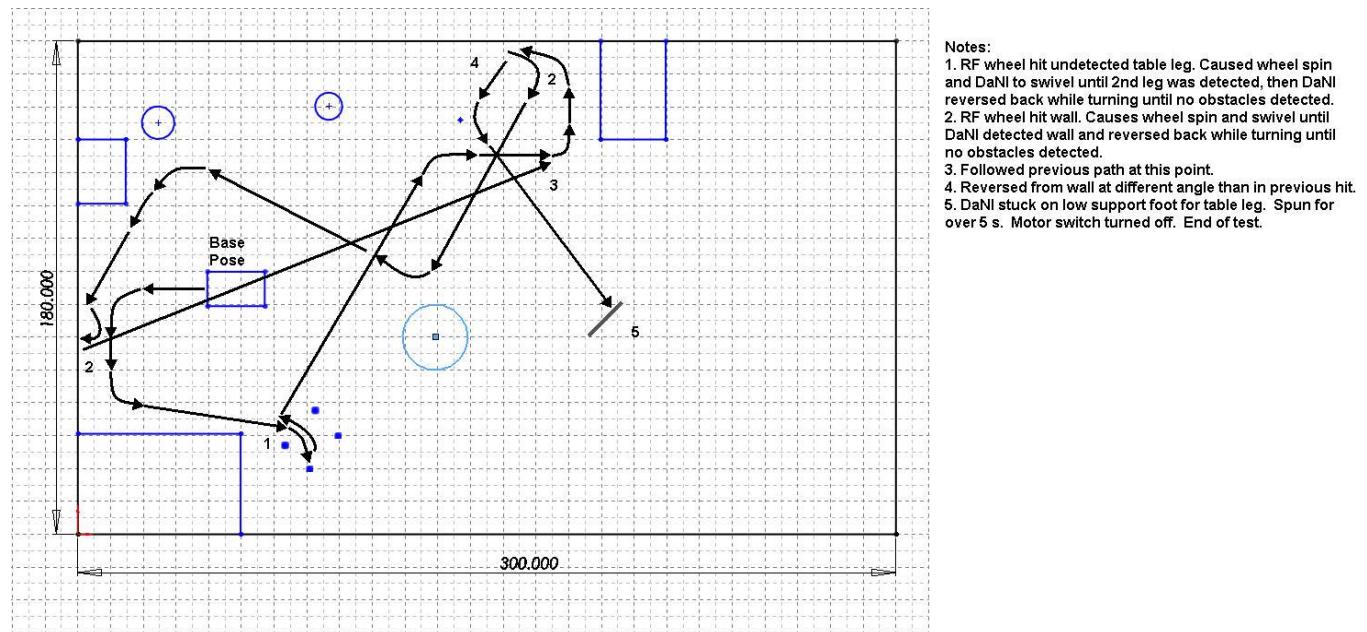


Figure 0-36. Example roaming path map

Open the robotics project that was built by the project wizard in the previous segment of the experiment, unless it is already open. You can open it by double clicking the file in Windows Explorer or by opening the LabVIEW 2011 program and clicking on the project that should be

listed in on the Getting Started window. If it isn't listed, choose File>>Open or choose Open Project and browse to the project.

Review the LabVIEW Robotics Starter Kit Safety Guide by navigating to the LabVIEW\readme directory on the DVD that is packaged with the kit and opening StarterKit\_Safety\_Guide.pdf

Place the robot in an orientation and location that you will identify as the base pose. Start the robot as in the test run above. Allow the robot to roam (navigate autonomously) for about 1 - 2 minutes. If the robot gets "stuck", i.e. with its wheels locked or spinning for over 5 seconds, push the motor stop button. Immediately pick DaNI up or press the motor stop button if DaNI approaches stairs and any other drop offs where DaNI might drive over the edge, fall, and be damaged.

Draw the path that the robot takes during the 1 - 2 minutes on the map. To draw the path, it is helpful if the area has a grid, like a tiled floor, or a floor with taped or string grid marks. The robot moves quickly so it is best, but not necessary, to record a 1- 2-minute video.

Start the robot again in the base pose and draw the path it takes twice more, so you have 3 path drawings on the map. Draw the 2nd and 3rd paths in different colors or line types to make them easily distinguishable from the first.

Navigate to the Pittsco web site and identify the wheel, motor, and drive train components used to construct DaNI. Create a drawing on paper or with CAD software that shows how the components are assembled. Report the specifications of each component.

Which of the four types of wheels described by Siegwart et (2011): standard, caster, Swedish, or spherical does DaNI use? Explain the effects of changing DaNI's rear wheel with one of the other three types. Explain the effects of changing DaNI's front wheels with one of the other three types.

What types of surfaces is DaNI limited to? Can it operate on tile, carpet, and wood floors? What size cracks or open spaces in a floor would limit DaNI's mobility?

Do you think DaNI could climb up and down a ramp (Don't do this as part of the experiment unless the ramp has sides to DaNI falling off.)? Would it be better to begin the climb forward or backward? Why?

Do you think DaNI was designed for operation outdoors (Don't operate it outdoors unless the weather is fine and the surface is clean and flat, like on a concrete or paved driveway)? What might happen if DaNI got dusty and gritty? What would happen if it got wet?

Could DaNI operate completely unsupervised i.e. (completely autonomous) for several hours (yes or no)? Why? Are there some obstacles that DaNI doesn't detect? Describe the obstacles

and explain why you think DaNI doesn't detect them? (This will be covered in more detail in the next experiment.)

Does DaNI have a goal to reach or is it just wandering around?

if DaNI can drive straight without having to avoid obstacles, how straight does it drive?

When DaNI detects an obstacle what action does it take? Does it slow down? Does it always turn the same direction to avoid an obstacle? What happens when DaNI bumps into something that makes it stop? Do its wheels spin on low-friction (slick) surfaces? On high-friction surfaces (carpet)? How does the construction of the robot keep it from being damaged when it bumps into something?

Place a small object that is below the servo motor on the floor and see if DaNI detects it as an obstacle?

What is the highest object that DaNI will drive over? How is that related to the wheel diameter? How is it related to the ground clearance?

Did DaNI turn in place anywhere while driving in the test area, i.e. with turning radius = 0? What commands do you think have to be given to the motors to make it turn in place?

## Experiment 1-4 Compare Autonomous and Remote Control

There isn't a wizard to create a project for remote control operation, so locate, copy to the host computer, and open the Teleop Starter Kit 2.0 project shown in Figure 0-37 by downloading from [ni.com](http://ni.com) or by copying from the starter kit DVD. Even though the name of the project is teleop (assuming teleoperation), the software can be used for remote control as well.

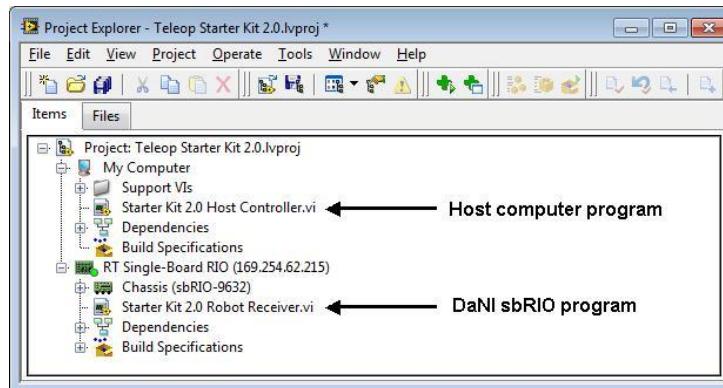
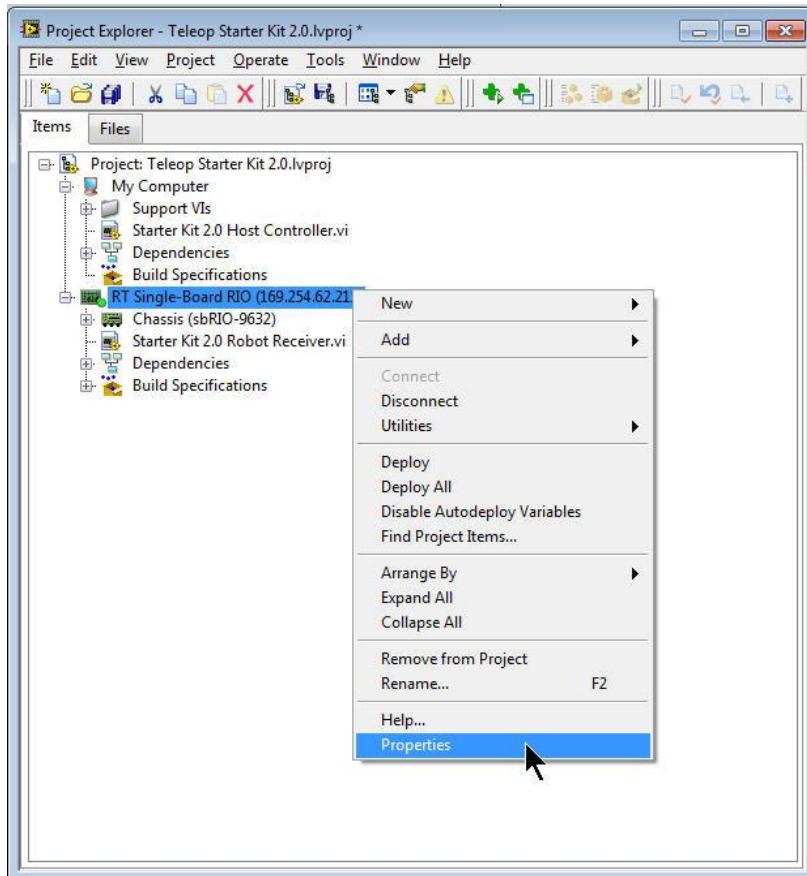


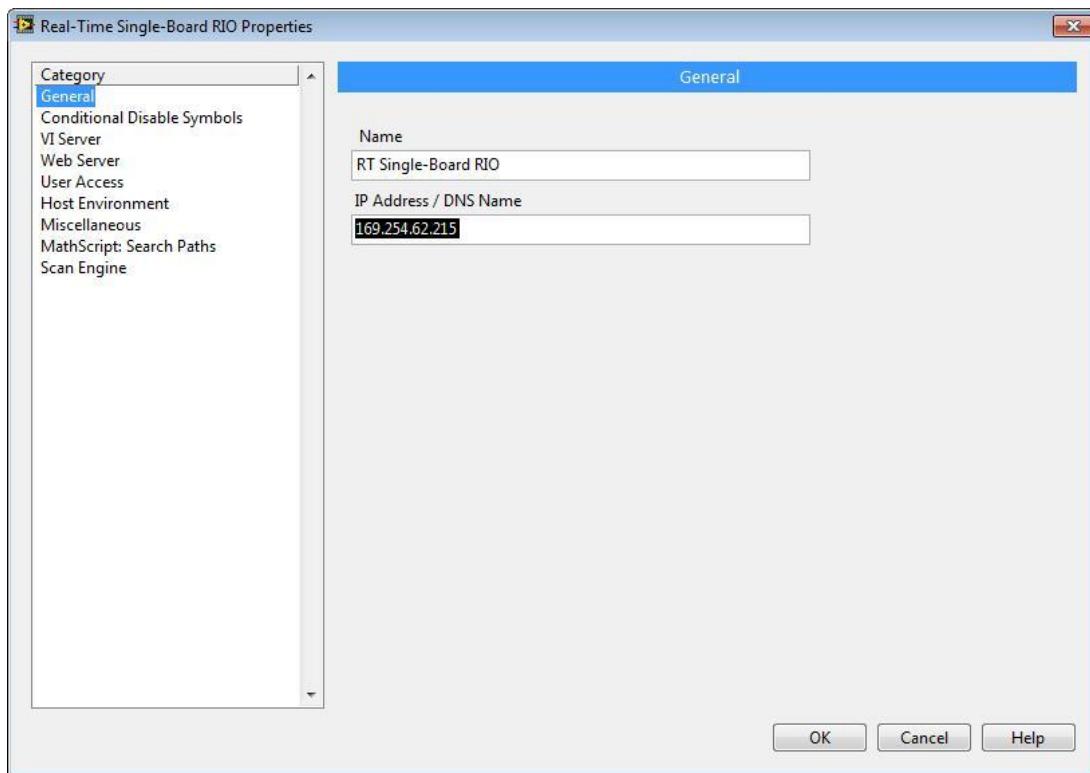
Figure 0-37. Teleop Starter Kit project

Because the project wasn't developed with a wizard, you need to enter DaNI's IP address. Remember that the IP Address might change in DHCP networks, so open MAX and check the IP address. Then, right click on the RT Single-Board RIO target in the project as shown in Figure 0-38 and choose properties.



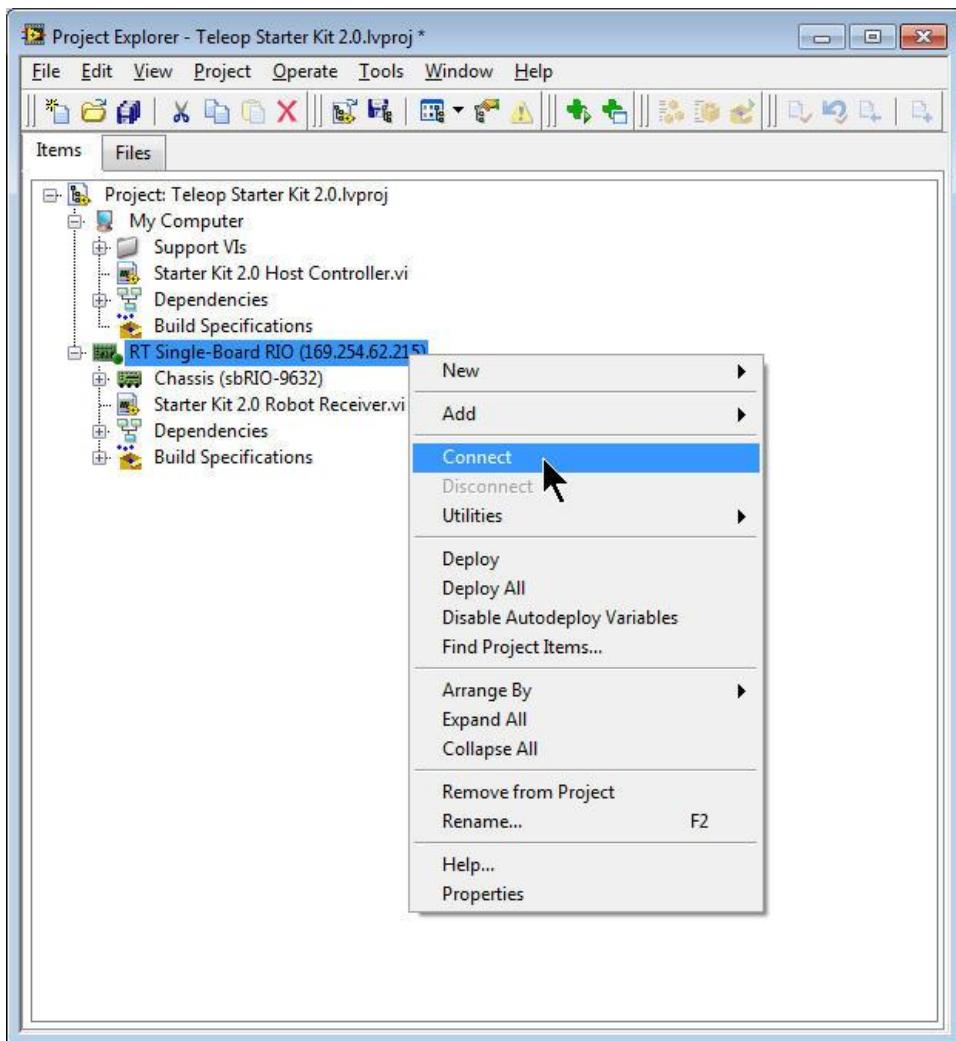
**Figure 0-38. Teleop project properties**

Enter the IP address of the DaNI sbRIO as shown in Figure 0-39 and click the OK button.



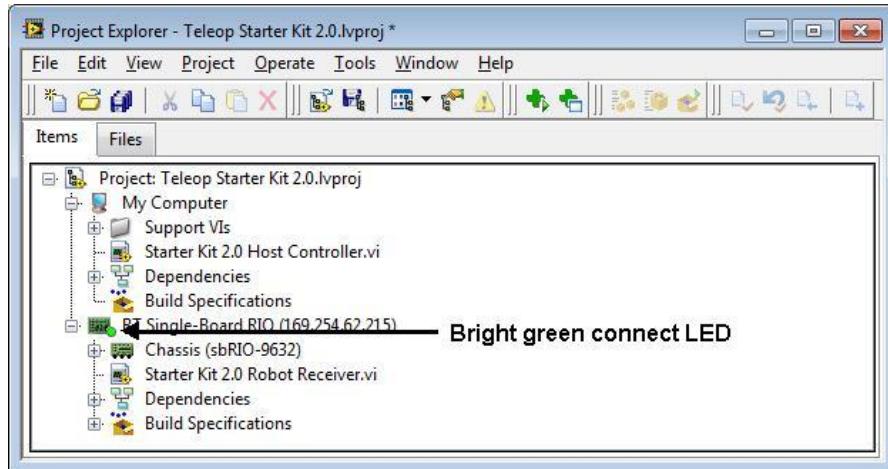
**Figure 0-39. IP configuration for teleop project**

Connect the cross over cable to the host computer or a CAT 5 cable to a hub or switch so DaNI can communicate to the host. It helps to have a long (~ 3 m) ethernet cable for this section of the experiment as you will drive DaNI while it is tethered to the host or hub/switch. Turn on the Master switch on, and the Motor switch off. Then, right click the sbRIO target in the project again and choose Connect as shown in Figure 0-40.



**Figure 0-40. Connect to DaNI in the project**

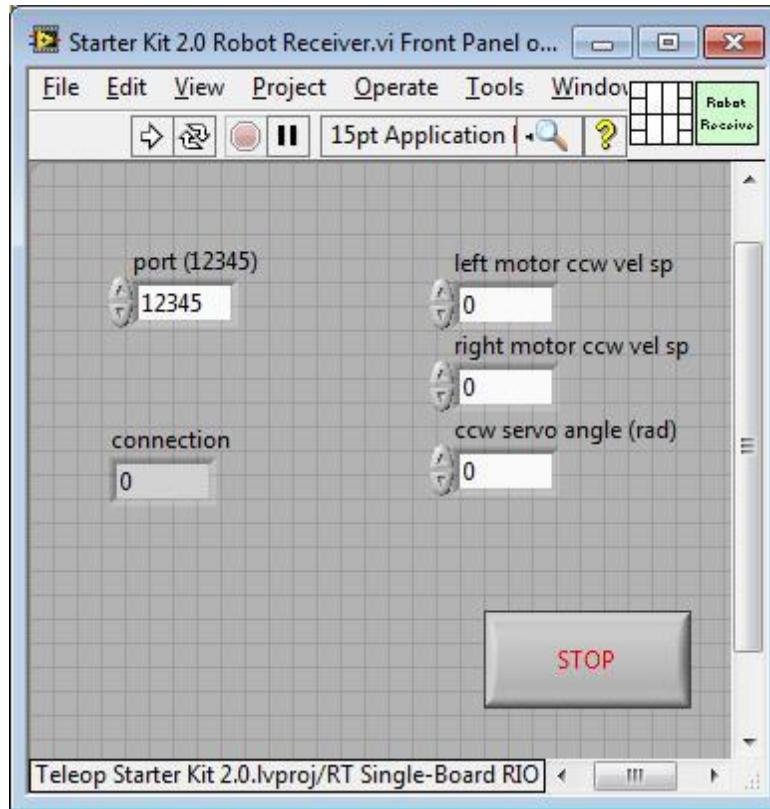
Software should deploy and the connect LED in the teleop project should turn bright green as shown in Figure 0-41.



**Figure 0-41. Bright green connect LED**

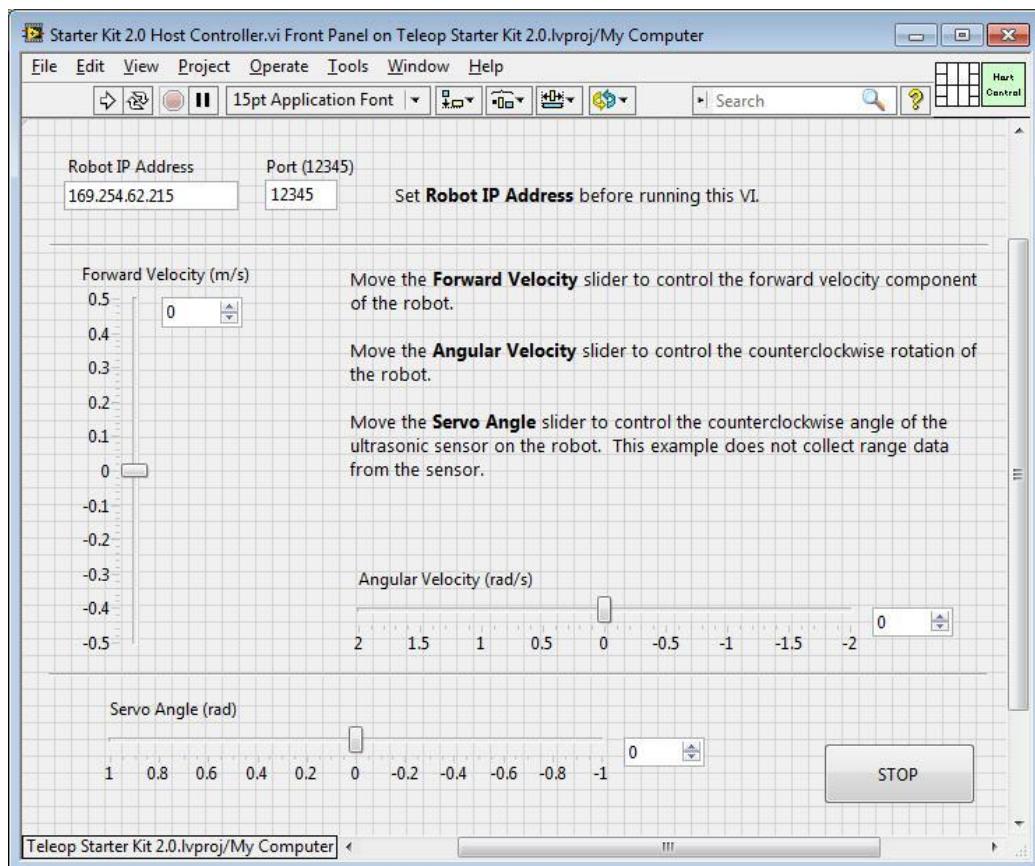
Note that there are two main programs, one for the host computer named Starter Kit 2.0 Host controller and one for the sbRIO named Starter Kit 2.0 Robot Receiver.

Open the Starter Kit 2.0 Robot Receiver program shown in Figure 0-42 that will run on the sbRIO target by double clicking it in the project explorer. Don't be concerned if you don't understand the port and other items on the front panel. Just click the run button to deploy and run the program on DaNI. Set the Master switch to on and the motor switch to off.



**Figure 0-42. sbRIO Robot Receiver VI front panel**

Open the Host Controller VI, shown in Figure 0-43, by double clicking it in the project explorer. Set the IP address and run the VI. It will communicate with the sbRIO program via the Ethernet cable, so DaNI will remain tethered in this part of the experiment. You will not disconnect the Ethernet cable as done in the previous section.



**Figure 0-43. Teleop project Host Controller VI**

Before test driving, move the servo angle slider to confirm that the host is communicating with the sbRIO program. You won't use the servo angle slider while driving, just use it to test communication before driving.

Turn the Motor switch on and move the Forward Velocity and Angular Velocity sliders to drive DaNI. Test drive DaNI until you have good control over DaNI's speed and direction. Then repeat the path from the previous section of the experiment with remote control to compare with autonomous operation and answer the following questions.

DaNI's only feedback is from the ultrasonic transducer and the wheel encoders. How does your perception compare with DaNI's when avoiding obstacles?

Which can react faster, your hands on the keyboard or the sbRIO computer program?

If the exercise required that DaNI operate for a longer period than 1 - 2 minutes, say for several hours, would remote control or autonomous operation be better?

# **Experiment 2 – Ultrasonic Transducer Characterization**

## **Instructor's Notes**

This experiment requires that the previous experiment be completed. Similar experimental area and tools used in the previous experiment are used here.

## **Goal**

Experiment with and characterize an ultrasonic transducer. Learn about the LabVIEW programming environment and learn some simple LabVIEW programming techniques.

## **Required Components**

Objects for ultrasonic transducer targets like a large cardboard box and a laptop bag.

Linear distance measuring tool like a ruler, meter stick, or tape measure.

Angle measuring tool like a protractor.

## **Background**

Students should study the Parallax website to obtain background on PING))), the ultrasonic transducer on DaNI.

Students should study the ultrasonic ranger sections of Chapter 4 in Siegwart et al (2011) or a similar text.

## **Experiment 2-1 Characterization with the Roaming VI Graph**

In the previous experiment DaNI roamed around an area that you designed. DaNI reacted when it saw an obstacle and it may have collided with some obstacles. This experiment will help you understand why DaNI detected some obstacles and not others. As you know from the

previous experiment, DaNI acquires data about obstacles from an ultrasonic transducer. Read a text like Siegwart et al (2011) to learn the fundamentals of the transducer.

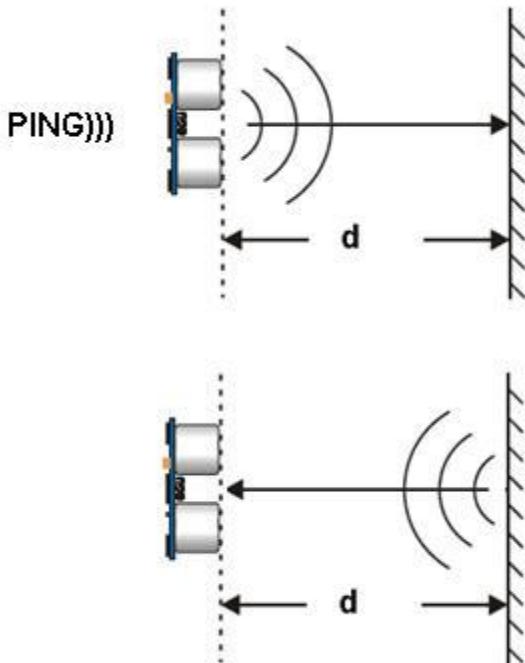
Study the specifications for PING))) on the Parallelex web site. The operational description and specifications report that PING transmits a short ( $200 \mu\text{s}$ ) burst of ultrasonic energy with 40 kHz frequency. Then it stops transmitting and “listens” for a reflected signal. The burst travels at 331.5 m/s to an obstacle and is reflected back to the transducer. The reflected signal could take up to 18.5 ms to return if the reflecting object is 3 m from the transducer. PING))) does not transmit any bursts while waiting for the receive signal. After receiving a reflection or timing out, because no reflection was received, PING))) waits  $200 \mu\text{s}$  before transmitting a new burst. Consequently, the period between bursts is about  $18.5 \text{ ms} + 2 * 0.2 \text{ ms} = 18.9 \text{ ms}$ . The transducer is connected to the sbRIO computer and the sbRIO acquires the transducer signal data.

The PING))) sensor provides an output pulse to the sbRIO that will terminate when the echo is detected, hence the width of this pulse corresponds to the distance to the target.

Use this information to calculate the bandwidth or frequency of the transducer. Explain how this might limit the speed that DaNI can roam.

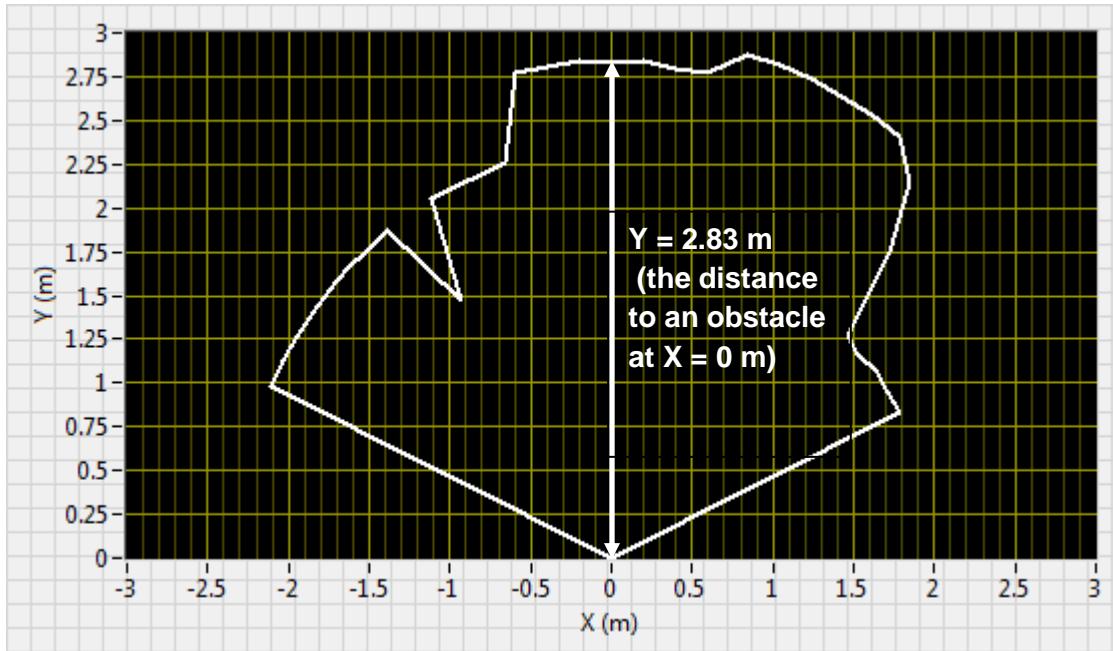
PING)))’s range is 0.2 to 3m. Explain why it can’t be 0 to 3 m.

Place DaNI in front of a good reflecting surface, like a wall, as shown in Figure 0-1 so that the surface of the target is parallel to the transducer backplane when the transducer is pointing at the wall in such a way that the center of the energy burst is perpendicular to the wall at the center of PING)))’s pan, i.e. when the servo angle is  $0^\circ$ . Clear a path about 3.5 m long. There can be some objects on the sides of the path, i.e. the path doesn’t have to be 3.5 m wide. Connect DaNI to the host via Ethernet or crossover cable. Turn on the Master switch, but not the motor switch. Open the roaming project and run the roaming VI.



**Figure 0-1. Plan view of the linear distance characterization set up**

Position DaNI so that PING))) is 3 m (the maximum range) from the reflecting surface, i.e. in Figure 3-1,  $d = 3\text{m}$ . Move away from the target so your body doesn't interfere with the results. The results should be similar to Figure 3-2. Since the servo motor pans (rotates) the PING))) mount pans  $65^\circ$  in either direction from the center, it will measure the distance to other objects in the area scanned by PING))) in Figure 0-2. For this measurement, consider only the distance to the obstacle at  $X(\text{m}) = 0$  and servo angle =  $0^\circ$ . For example, the distance the obstacle of interest, Y, is approximately 2.83 m (interpolating between 2.75 and 3m) at  $X = 0 \text{ m}$  in Figure 3-2.



**Figure 0-2. Distance to an object near the maximum PING))) range.**

Measure the distance from the obstacle to PING))) with a tape measure and compare it with the results of the Roaming VI graph.

Move DaNI toward the wall in 0.5 m increments and record the Y (m) value to the obstacle at X = 0 m each time. Make a graph of the Roaming VI versus tape measurement distance.

Instead of using a hard, reflective material like a wall, repeat the above experiment with something softer that would absorb the sound better. For example make a target from a box covered with a coat or a chair cushion. At what distance away does the signal drop below the threshold? What does this tell you about the ability to detect and avoid all obstacles of different materials with ultrasound?

Repeat the experiment with a small object and with a round object like a soccer or volley ball. What does this tell you about DaNIs ability to detect the size and shape of obstacles? Might PING))) report the distance to some type of objects and not others even though they are in the same location relative to the transducer?

Place a flat object, like the side of a cardboard box in front of PING))) and 2.75 m away. Orient it parallel with the wheel axle which will put it orthogonal to the transducer signal at angle 0. Run the Roaming VI with DaNI tethered and the motors off. Leave the box at this distance from PING))) and turn the box so it is no longer orthogonal to the signal while observing the graphed data. Stop turning the box when the obstacle no longer appears in the graph. Record the angle. Repeat this at 0.5 m intervals and graph the angle vs the Y (m) distance from PING))).

What does this tell you about DaNI collisions when approaching a wall at a shallow acute angle?

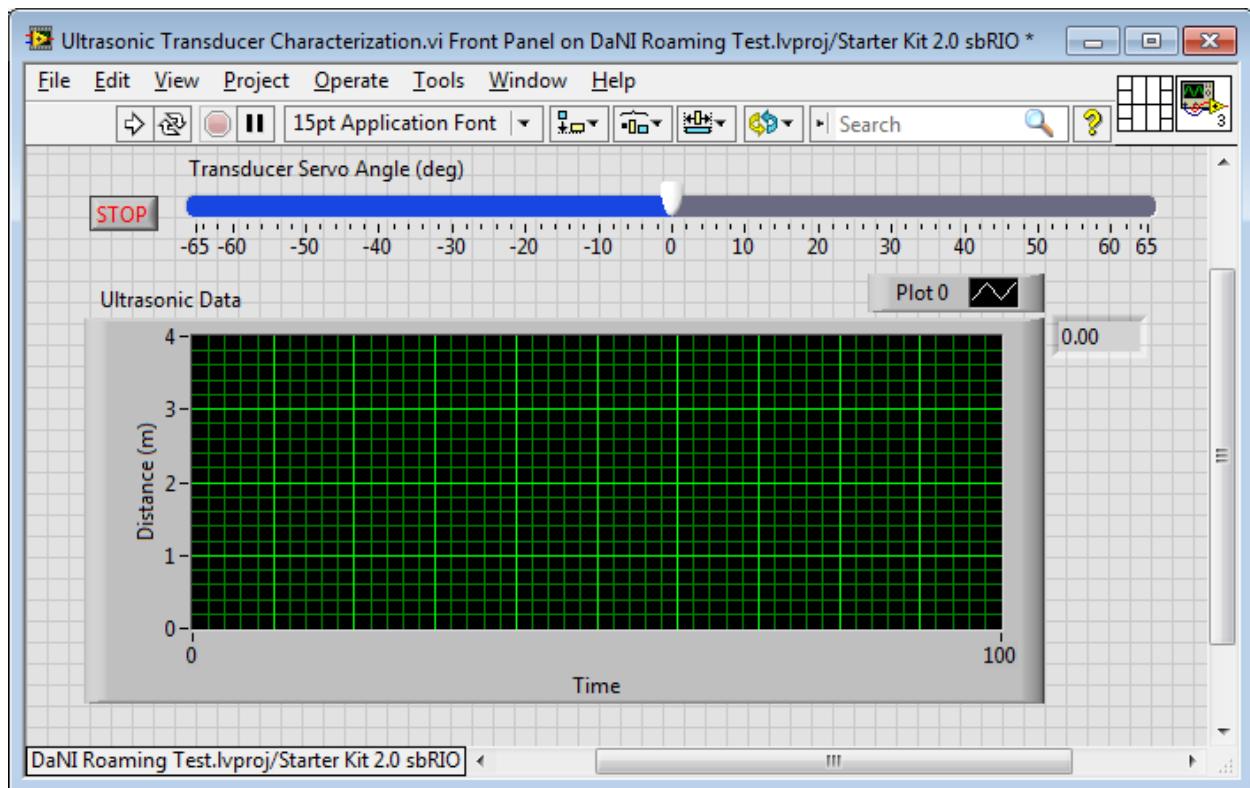
Place an object in front of the sensor while viewing the graph. How long does it take to obtain stable reading on the graph? Does the distance to the object affect the time to a stable reading?

Determine the height of an obstacle that DaNI will detect at various distances.

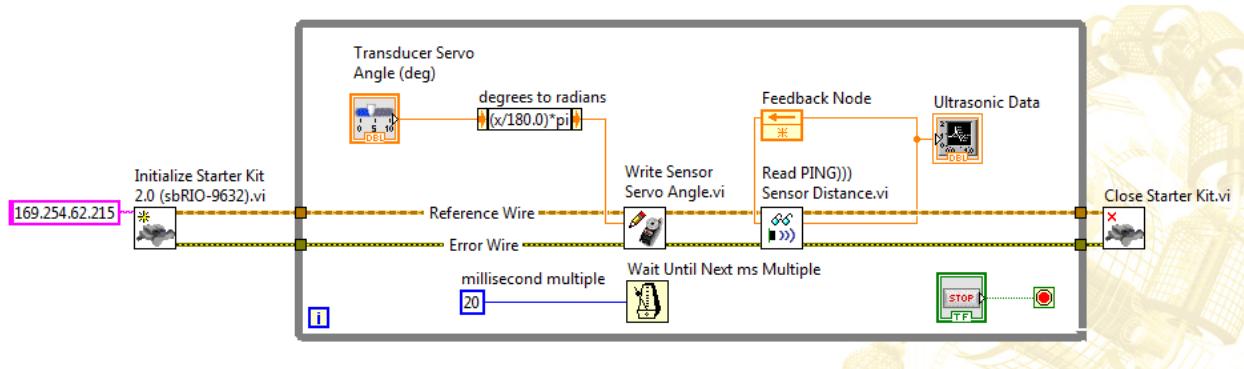
## Experiment 2-2 Introduction to LabVIEW

The results from the previous section of the experiment were approximate because you were interpolating graphed data. The experimental results would be more accurate if the exact distance was displayed digitally. It was also tedious to write down the different values from the interpolation. Saving to a file would be more useful. You could evaluate the FOV if DaNI wasn't panning during the experiment. You could also determine if there were several objects in the FOV, what distance was reported (average, closest, most far, most reflective, or?).

You need a different program instead of the Roaming VI to solve these issues. This segment of the experiment teaches you how to build the program shown in Figure 0-3 and in Figure 0-4 LabVIEW. If you already know LabVIEW, you can build this program in just a few minutes. If you don't, follow the steps below to learn LabVIEW and build the program. This will take longer, but as you progress through this set of experiments, concurrently learning LabVIEW, program development time will decrease. If you want to use a program that has already been built for you in the Roaming project instead of coding your own VI, navigate to the Test Ultrasonic Sensor VI in the Test Panels folder in the Roaming project. It is similar to the VI that will be created in this section of the experiment.

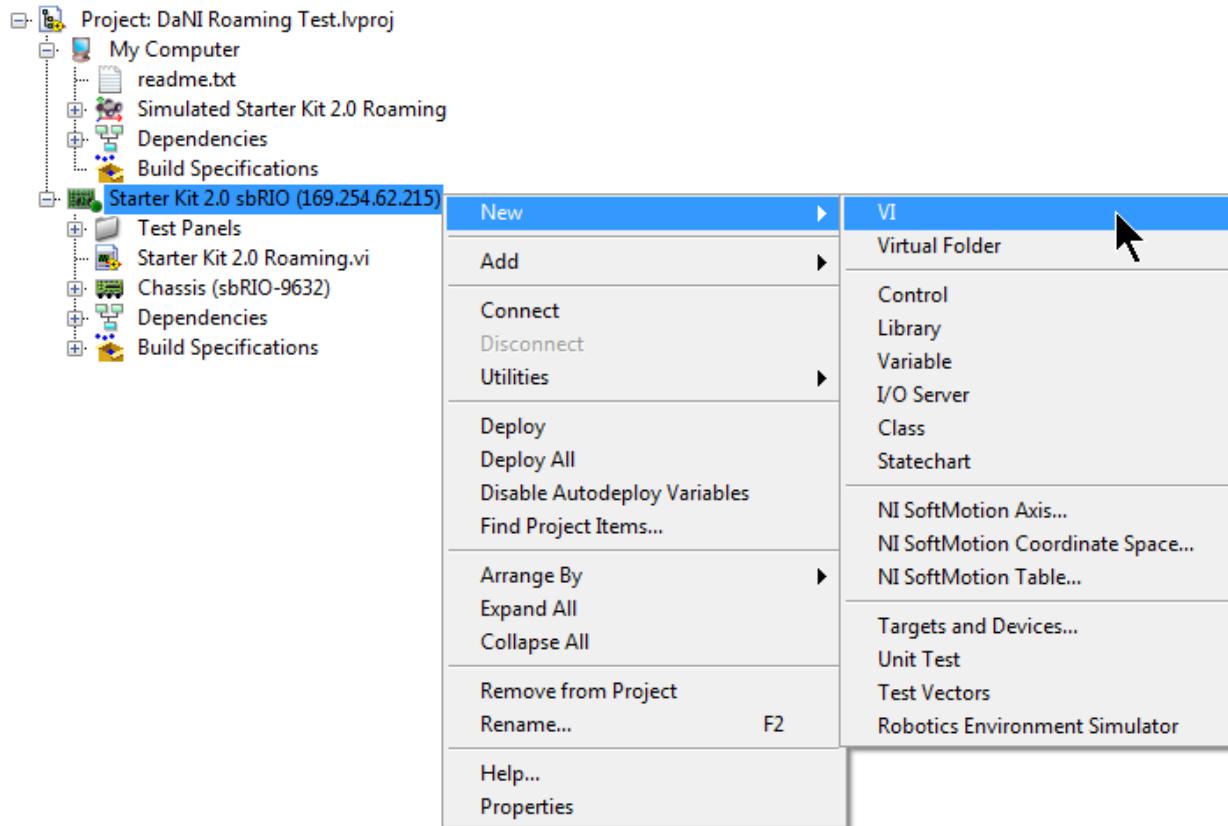


**Figure 0-3. Ultrasonic Transducer Characterization Program graphical user interface (front panel)**



**Figure 0-4 Ultrasonic Transducer Characterization Program graphical code (block diagram)**

You will create the program on a laptop or desktop PC, but you will send the program to the robot and it will run on the sbRIO. As discussed in Experiment 1, the LabVIEW Project Explorer facilitates communication between a PC and a remote target (the sbRIO) so you will create the program in the project. Open the Roaming project as shown in Figure 0-5, right click the sbRIO item, since the program will run on the sbRIO, and choose New>>VI.



**Figure 0-5 Add New VI in the DaNI Roaming Test LabVIEW Project**

To review Experiment 1, the items in the project are arranged in a tree or hierarchical structure. The first item, Project: Experiment 1Display Ultrasonic Data.lvproj, is the root. This item shows the name of the file saved on disk with the file extension lvproj. The second item, My Computer, is indented to show it is lower in the hierarchy. It represents the PC where programs are developed.

The third and fourth items Dependencies and Build Specifications are indented below My Computer indicating that they are lower in the hierarchy and belong to My Computer. Dependencies include items that programs require. Build specifications includes configurations for stand-alone applications, shared libraries, installers and zip files. You won't need to work with either dependencies or build specifications in this experiment.

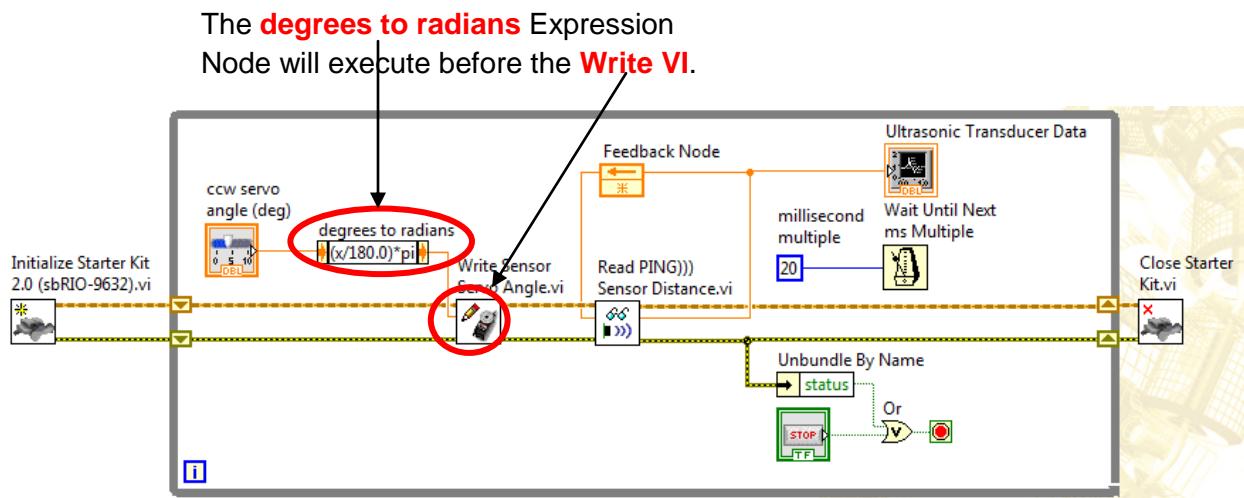
The next item moves up in the hierarchy so its level is equivalent to My Computer. It represents another computer in the project, the sbRIO on DaNI. In addition to the name of the computer, the IP address is displayed. The sbRIO item has dependencies and build specification items like My Computer and some additional items. The Chassis item is part of the sbRIO that connects to and communicates with transducers and actuators.

This section of the experiment will now focus on the concepts in LabVIEW necessary to develop the Ultrasonic Transducer Characterization.vi program. These concepts are essential to other experiments in this series as well as many other programming applications.

LabVIEW is different from most other general-purpose programming languages in two major ways. First, LabVIEW programming is performed by wiring together graphical icons on a diagram, as you could see in Figure 0-4. The graphical code is then compiled directly to machine code so the computer processors can execute it. This general-purpose programming language is known as G and includes an associated integrated compiler, a linker, and debugging tools. G contains the same programming concepts and all the standard constructs found in most traditional languages, such as data types, loops, hierarchical programming, event handling, variables, recursion, object-oriented programming, and others.

The second main differentiator is that G code developed with LabVIEW executes according to the rules of data flow instead of the more traditional procedural approach (in other words, a sequential series of commands to be carried out) found in most text-based programming languages like C and C++. Dataflow languages like G (as well as Agilent VEE, Microsoft Visual Programming Language, and Apple Quartz Composer) promote data as the main concept behind any program. Dataflow execution is data-driven, or data-dependent. The flow of data between nodes in the program, not sequential lines of text, determines the execution order.

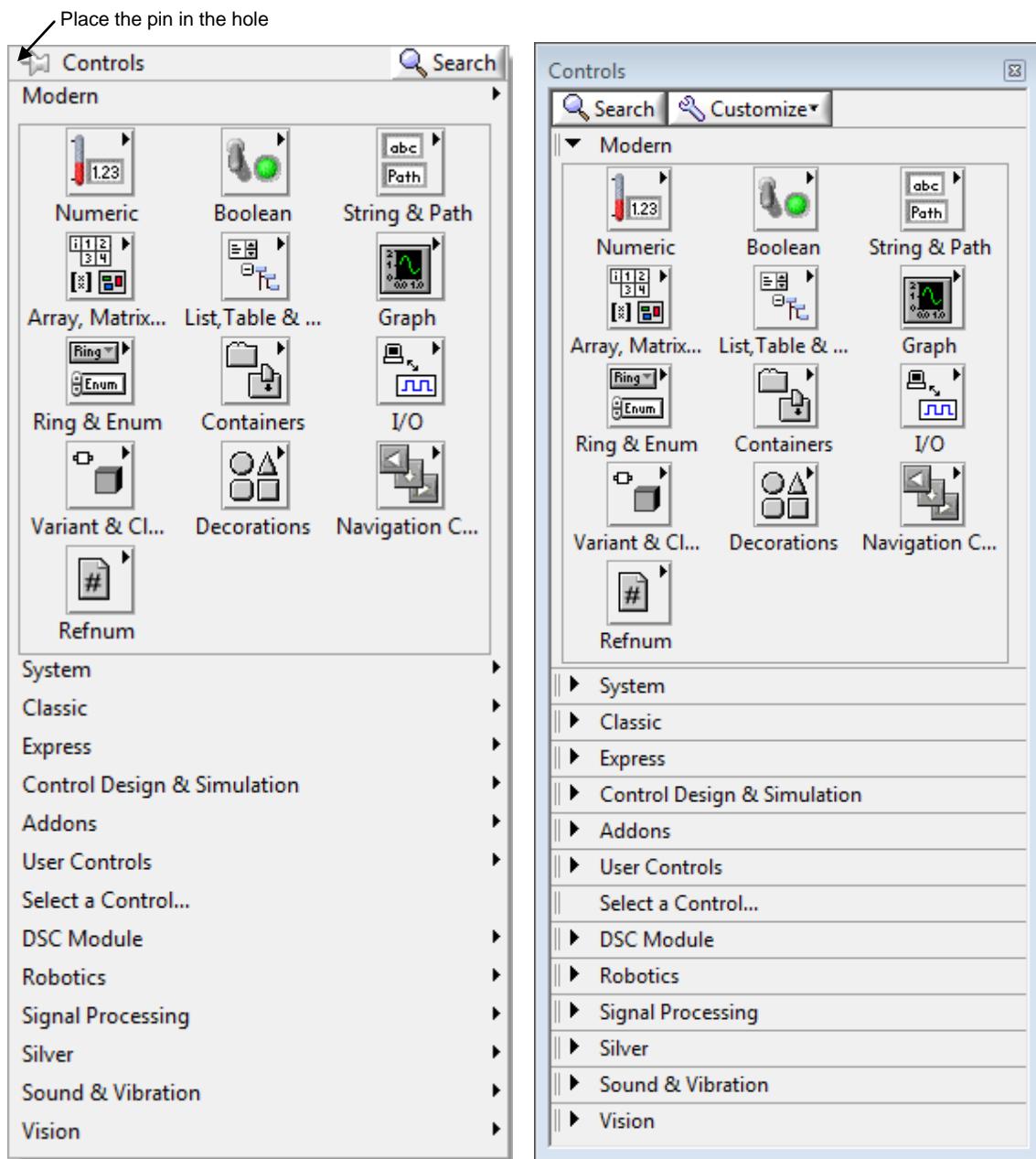
This distinction may seem minor at first, but the impact is extraordinary because it renders the data paths between parts of the program to be the developer's main focus. Nodes in a LabVIEW program (in other words, functions, structures such as loops, subroutines, and so on) have inputs, process data, and produce outputs. Once all of a given node's inputs contain valid data, that node executes its logic, produces output data, and passes that data to the next node in the dataflow path. A node that receives data from another node can execute only after the other node completes execution as shown in Figure 0-6.



**Figure 0-6. Data flow example**

As explained in Experiment 1, LabVIEW programs are called virtual instruments, or VIs, because their appearance and operation imitate physical instruments, such as an oscilloscope or a multimeter. LabVIEW contains a comprehensive set of tools for acquiring, analyzing, displaying, and storing data, as well as tools to help you troubleshoot code you write.

A VI has two windows, a front panel window and a block diagram window. Experiment 1 used the front panel window as the graphical user interface. You will build the front panel shown in Figure 0-3 in this section of Experiment 2. The front panel contains a Stop button, a Chart, and a Slider. These and other objects are created with a palette. The Controls palette contains the controls and indicators you use to create the front panel. You access the Controls palette from the front panel by selecting View»Controls Palette or by right clicking on any empty space in the front panel. The Controls palette is divided into various categories; you can expose some or all of these categories to suit your needs. The number and type of categories depends on which modules were loaded with LabVIEW. Figure 0-7 shows a Controls palette with all of the categories exposed and the Modern category expanded. If you maneuver the pin in the upper left of the palette, you can pin the palette and it will change as shown.



**Figure 0-7. Floating and pinned controls palettes**

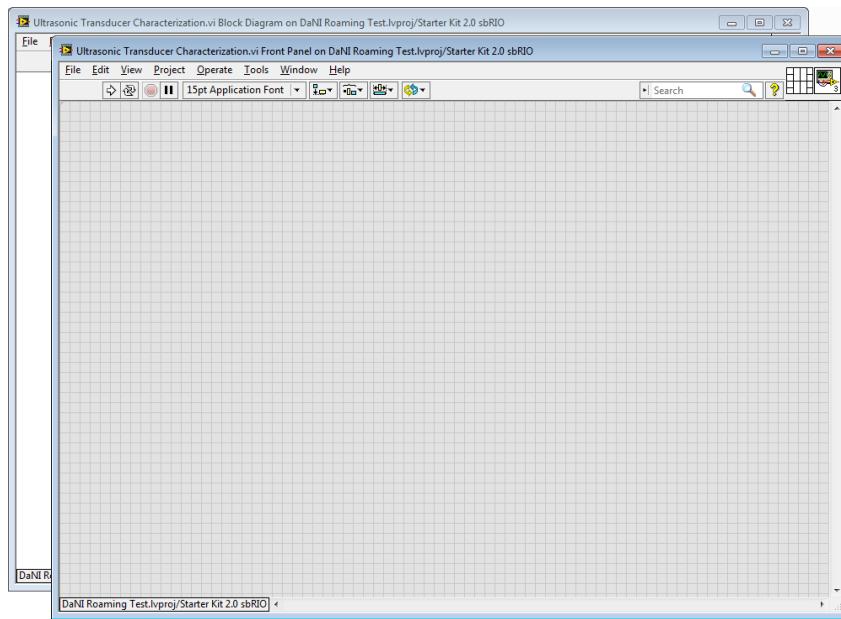
You create the front panel with controls and indicators, which are the interactive input and output terminals of the VI, respectively. Controls are knobs, push buttons, dials, and other input devices. Indicators are graphs, LEDs and other displays. Controls simulate instrument input devices and supply data to the block diagram of the VI. Indicators simulate instrument output devices and display data the block diagram acquires or generates. The Stop button and the Slider in Figure 0-3 are controls and the chart is an indicator.

The user can stop the program with the Stop button. The user can control the angular position of the PING))) mount with the slider. The user can see the values generated by the DaNI

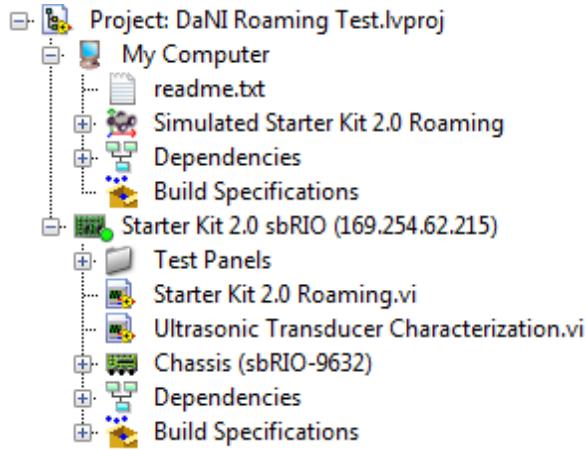
ultrasonic transducer on the chart indicator. The VI acquires the values for the indicators based on the code created on the block diagram.

Every control or indicator has a data type associated with it. For example, the Stop button control is a Boolean data type. The most commonly used data types are numeric, Boolean and string. The numeric data type can represent numbers of various types, such as integer or real. Objects such as meters and dials represent numeric data. The Boolean data type represents data that has only two possible states, such as TRUE and FALSE or ON and OFF. Boolean objects simulate switches, push buttons, and LEDs. The string data type is a sequence of ASCII characters. String controls receive text from the user such as a password or user name. String indicators display text to the user. The most common string objects are tables and text entry boxes.

When you right clicked the sbRIO item in the project explorer window and chose New>>VI a blank VI like the one shown in Figure 0-8 opens. Name it by saving it as Ultrasonic Transducer Characterization.vi with File>>Save As in the same folder as the project. It will appear under the sbRIO target in the project as shown in Figure 0-9.

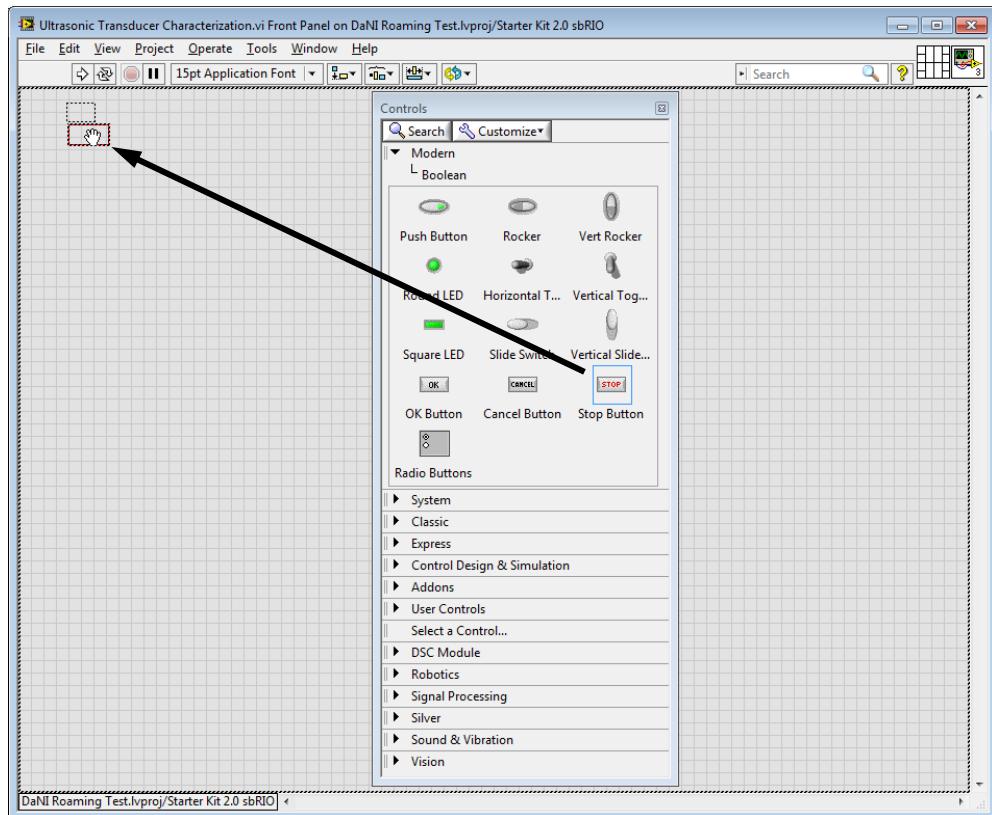


**Figure 0-8. Blank front panel and block diagram of the Characterization VI**



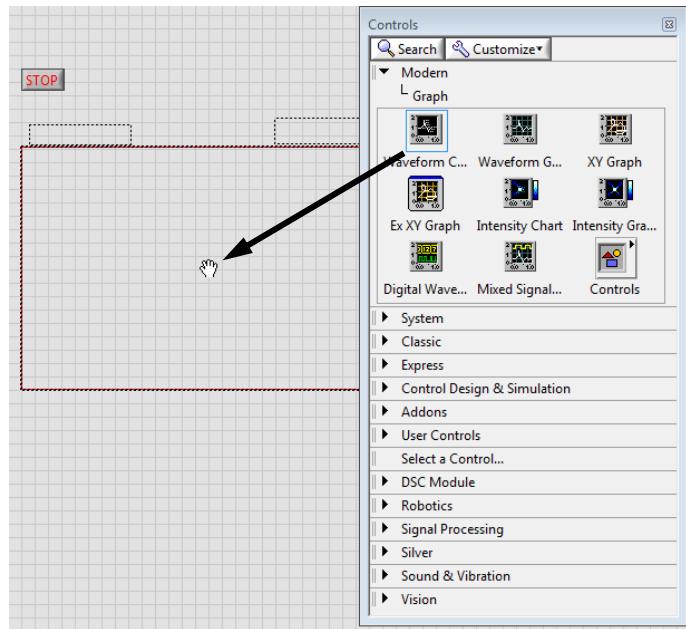
**Figure 0-9 Roaming project with the Characterization VI**

Add the Stop button by right clicking in the front panel window to open the controls palette. Click the Boolean palette and drag the Stop button control onto the front panel as shown in Figure 0-10.



**Figure 0-10. Place a Stop button on the front panel**

Since the Stop button has red stop text, you don't need the label. Right click the button, choose Visible Items and uncheck the label. Add the chart by clicking the Graph palette in the controls palette and drag the Waveform chart indicator onto the front panel as shown in Figure 0-11.



**Figure 0-11. Waveform chart on the front panel**

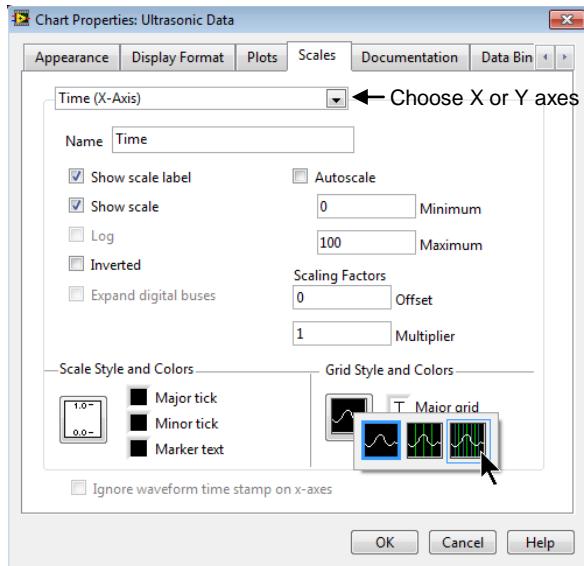
Change the chart name by typing Ultrasonic Data Chart while the name has a black background as shown in Figure 0-12. If the name doesn't have a black background, double click in the name area and edit the name. The chart name is called the label.



**Figure 0-12. Change the chart name (label)**

Change the vertical, dependant, or y axis title by double clicking on Amplitude and edit it to read Distance (m). Right click the chart and choose Y Scale in the short-cut menu. Then choose AutoScale Y to deselect it. (The checkmark will vanish.) Double click the upper range value of 10 and edit it to read 4. Double click the lower range value of -10 and edit it to read 0. Right

click the chart and choose Properties to open the window shown in Figure 0-13. Click the Grid Style Colors icon and choose the Major and Minor tick grids as shown. Then change to the Y-axis with the pull down and configure it with Major and Minor tick grid lines as well. Note the many additional properties you can change in this window and experiment with some of them.

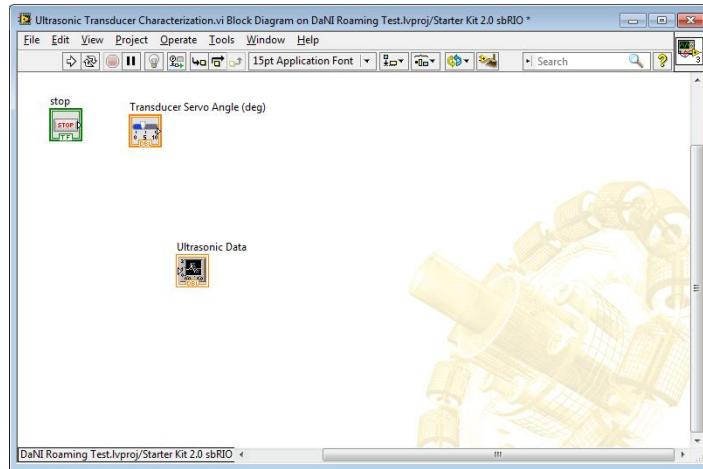


**Figure 0-13. Customize the chart**

Right click the chart, choose Visible Items >> Digital Display so you will know the exact value of the distance and you won't have to approximately interpolate values.

Add a Horizontal Pointer Slider control from the Modern Controls >> Numeric palette to the front panel using the same process as with the previous two objects, and resize and customize it as was shown in Figure 0-3.

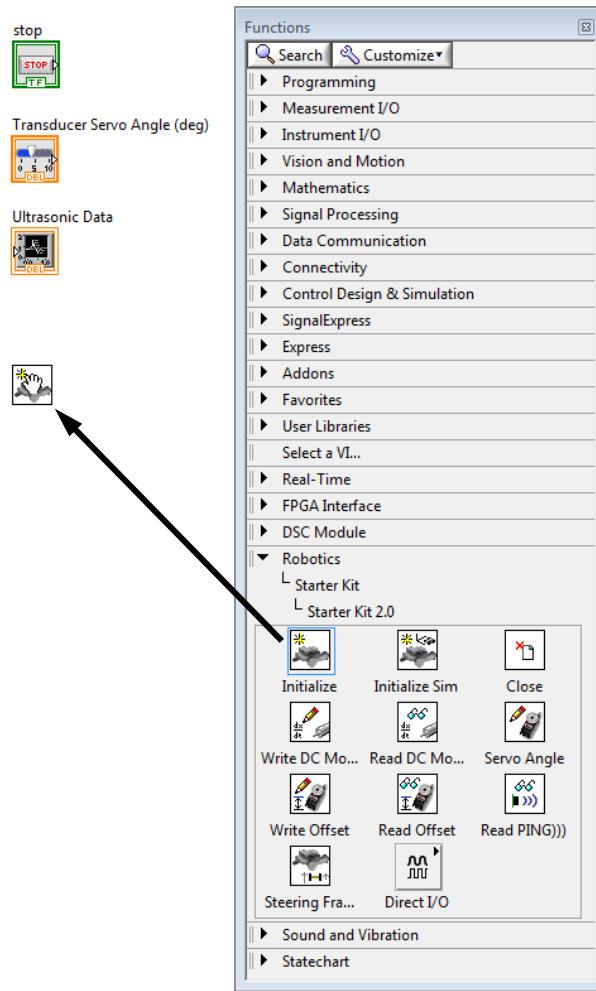
After you create the front panel, switch to the block diagram by either clicking on it, choosing Window>>Block Diagram, or by pressing the shortcut key: Ctrl E. In the block diagram, you create source code using graphical representations of functions that interact with the front panel objects. Block diagram objects include terminals, subVIs, functions, constants, structures, and wires, which transfer data among other block diagram objects as was shown in Figure 0-4. As seen in Figure 0-14 icons for the button, chart and slider were automatically placed on the block diagram as you created them.



**Figure 0-14. Front-panel object icons automatically added to the block diagram**

There are a set of icons in a palette that can be used to populate the block diagram window similar to the way the controls palette was used to populate the front panel window. To view the Functions palette, right click in the block diagram window. Build the block diagram shown in Figure 0-4 from left to right.

Add the Starter Kit 2.0 Initialize VI from the Functions>>Robotics>>Starter Kit>>2.0 palette as shown in Figure 0-15. This VI establishes a reference to the program that runs on the sbRIO FPGA (field programmable gate array). I.e. it establishes a link between the Characterization VI and the sbRIO FPGA VI so they can exchange data. Remember that the sbRIO includes a processor and an FPGA as shown in Figure 1-2.

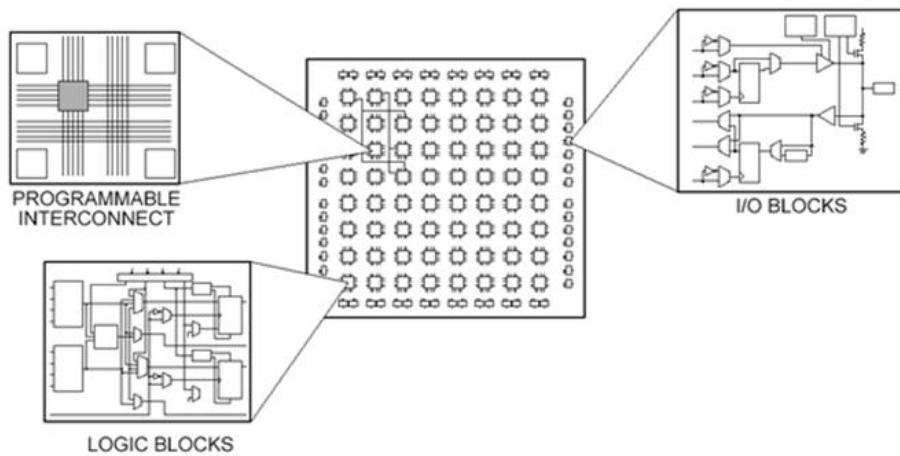


**Figure 0-15. Add the Initialize Starter Kit 2.0 VI icon to the block diagram.**

It can be difficult to locate the icon you want in the functions palette as there are a large number of icons available in a heirarchy. Click the search button shown at the top of the functions palette window in Figure 0-16 and type in part of the name.

As its name implies, an FPGA is hardware circuitry composed of an array of logic gates. However, unlike hard-wired printed circuit board (PCB) designs, which have fixed hardware resources, software written for FPGAs can literally rewire their internal. When an FPGA is configured, the internal circuitry is connected in a way that creates a hardware implementation of the software application. Unlike processors, FPGAs use dedicated hardware for processing logic and do not have an operating system. FPGAs are truly parallel in nature so different processing operations do not have to compete for the same resources. FPGA devices deliver the performance and reliability of dedicated hardware circuitry but are reconfigurable with software, in this case, LabVIEW VIs. In addition to offering high reliability, FPGA devices can perform deterministic closed-loop control at extremely fast loop rates. In most FPGA-based control applications, speed is limited by the sensors, actuators, and I/O modules rather than the

processing performance of the FPGA. For example, the proportional integral derivative (PID) control algorithm that is included with the LabVIEW FPGA Module executes in just 300 nanoseconds (0.000000300 seconds). PID control will be discussed in a later experiment. A single FPGA can replace thousands of discrete components by incorporating millions of logic gates in a single integrated circuit (IC) chip. The internal resources of an FPGA chip consist of a matrix of configurable logic blocks (CLBs) surrounded by a periphery of I/O blocks as depicted in Figure 0-17. Signals are routed within the FPGA matrix by programmable interconnect switches and wire routes.



**Figure 0-17. FPGA components**

One of the FPGA I/O blocks acquires data from the ultrasonic transducer. The sensor is physically wired to a terminal on the sbRIO that passes the data to the FPGA. Software configures the FPGA to communicate the data over the sbRIO bus to the real-time processor. The driver for the ultrasonic transducer has already been implemented for you, and it will be loaded onto the FPGA automatically. The FPGA VI that acquires the signal from the ultrasonic transducer has already been developed for you and compiled into a bitfile which is embedded in the project, as shown in Figure 0-18, and will be loaded automatically to the FPGA when the Characterization VI runs.

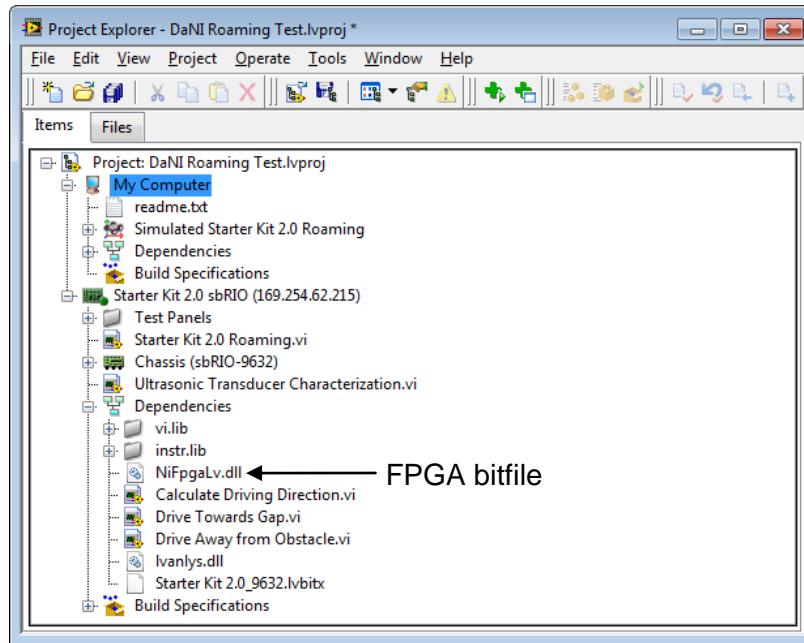


Figure 0-18. FPGA bitfile in the project

Virtual wires connect icons on the block diagram. To learn where wires can be connected to a particular icon, right click the icon and choose help or hover the cursor over the icon and use the Ctrl H shortcut. A window like the one shown in Figure 0-19 will open. Additional information, shown in Figure 0-20 is available if you click the Detailed help link or the ? button.

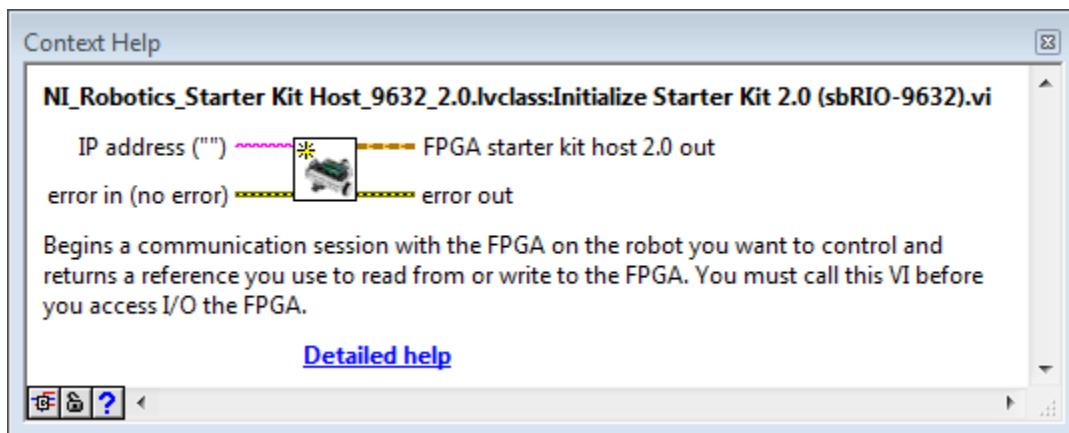


Figure 0-19. Help window for the Initialize Starter Kit 2.0 VI.

## Initialize Starter Kit 2.0 (sbRIO-9632) VI

Owning Palette: [Starter Kit 2.0 VIs](#)

Requires: [LabVIEW Robotics Starter Kit](#)

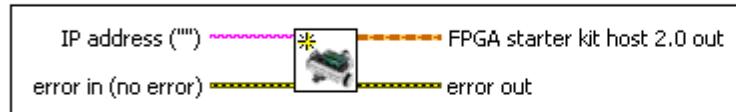
Begins a communication session with the FPGA on the robot you want to control and returns a reference you use to [read from or write to the FPGA](#). You must call this VI before you access I/O the FPGA.



**Note** You must configure the sbRIO-9632 chassis to [use the LabVIEW FPGA Interface programming mode](#) or the [Starter Kit](#) VIs do not run on the sbRIO.

Use the [Close Starter Kit](#) VI to end the communication session with the robot.

[Details](#) [Example](#)



Add to the block diagram



Find on the palette



**IP address** specifies the IP address of an sbRIO connected to the host computer. By default, this VI uses the IP address defined for the sbRIO target in the [Project Explorer](#) window.



**error in** describes error conditions that occur before this node runs. This input provides [standard error in](#) functionality.



**FPGA starter kit host 2.0 out** is a reference to the communication session between the host VI and the FPGA on the robot. You can wire this output to other [Starter Kit](#) VIs.



**error out** contains error information. This output provides [standard error out](#) functionality.

### Initialize Starter Kit 2.0 (sbRIO-9632) Details

To [simulate a Starter Kit application](#), use the [Initialize Simulated Starter Kit 2.0](#) VI instead of this VI.

### Example

Refer to the `Starter Kit 2.0.lvproj` in the `labview\examples\robotics\Starter Kit 2.0` directory for an example of using the Initialize Starter Kit 2.0 (sbRIO-9632) VI.



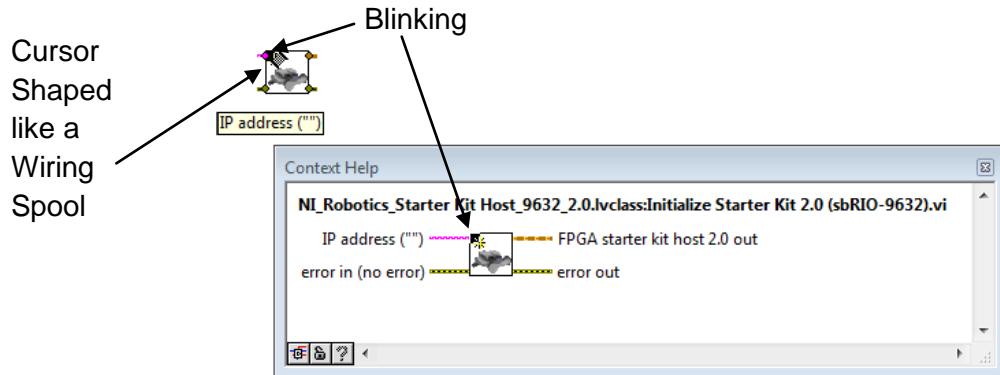
Open example



Find related examples

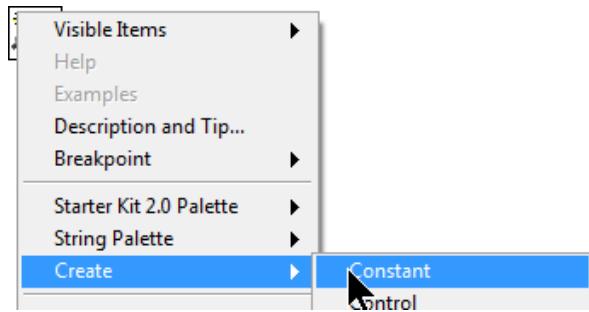
**Figure 0-20. Initialize Starter Kit 2.0 VI Detailed Help**

Figure 0-19 and Figure 0-20 show that there are four possible connections to this icon. Note the different colors of the wires. The colors indicate data types. Data is input to the icon on the left side and output on the right side. The upper left connection accepts the value of the IP address. If you are connected to a network via a cat 5 ethernet cable, this input can be useful, but it probably isn't necessary for a direct crossover connection to the host. To create this input, a constant and a wire are needed. You can create both at once. Hover the mouse cursor over the connection and the mouse shape will change to a wiring spool and both the connection terminal box and the connection in the help window will blink as shown in Figure 0-21.



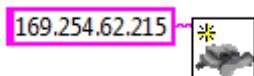
**Figure 0-21. Hover the mouse over the IP Address input connection**

Right click and choose Create>>Constant as shown in Figure 0-22.



**Figure 0-22. Create the IP address constant**

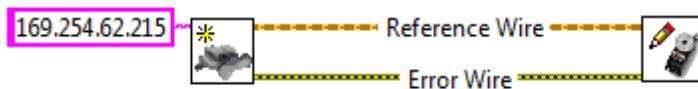
This will create a constant and will automatically wire it to the icon as shown in Figure 0-23. Enter the dotted decimal IP Address value into the constant. The data type of the constant is String which has a pink color.



**Figure 0-23. The IP Address constant**

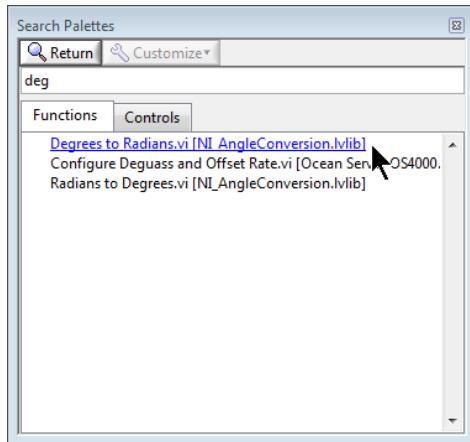
Next, place the Write Sensor Servo Angle VI on the block diagram from the Functions>>Robotics>>Starter Kit>>2.0 palette. This VI will allow you to point the transducer relative to the DaNI frame. Wire from the Initialize VI to this one as shown in Figure 0-24. The upper wire is a reference wire that identifies the initialized device in case the VI contains several such devices. An analogy for a reference data type is an address or a pointer. When the data is passed from the sbRIO FPGA to the processor, it is written to memory. The reference is analogous to the memory address. When the Initialize VI is executed, it automatically reserves

memory for the data communicated from the FPGA to the processor. The lower wire is error data so if an error occurred in the initialize VI, the write VI might not attempt execution.



**Figure 0-24. The data flow wire paths to the Write Servo Angle VI**

The help window for the Write VI indicates an input for the angle in radians. Most users are more familiar with degrees, so the user interface slider should be in degrees and the block diagram should convert to radians. There is a degrees to radians conversion available from the functions palette. Use the search button on the functions palette to locate it as shown in Figure 0-25.

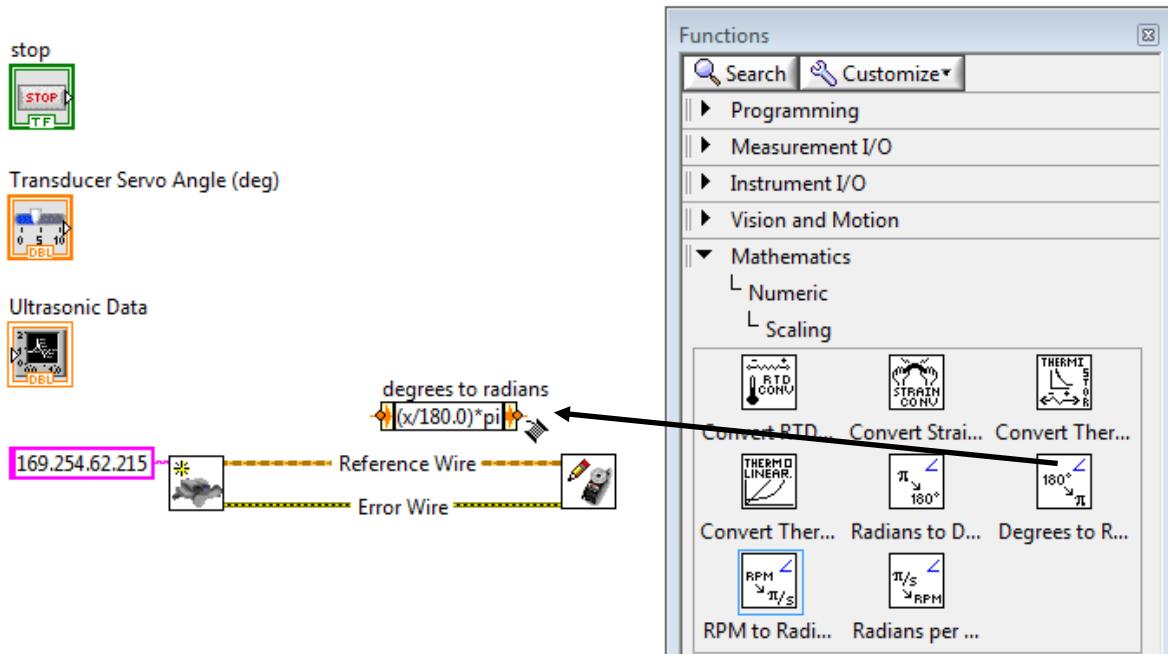


**Figure 0-25. Search results for Degrees to Radians conversion**

Double click the degrees to radians VI in the search list to see its location in the functions palette hierarchy. Drag the icon onto the block diagram to the position shown in Figure 0-26.

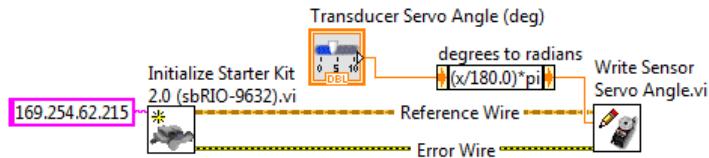
The icon represents an equation implemented in an Expression Node. The Expression node can evaluate expressions with a single variable. Several built in functions can be used in the node. Select Help from the LabVIEW menu bar and choose LabVIEW Help from the pull down, then search for Expression Node to view the online help that includes the list of functions.

Move the cursor over the output connection as shown. The cursor changes to the wiring spool shape and the connection blinks. Click connection to start the wire. Move the cursor to the ccw servo angle (rad) input connection on the Write VI. Note the orange color of the wire which represents a double precision floating point data type designated DBL.



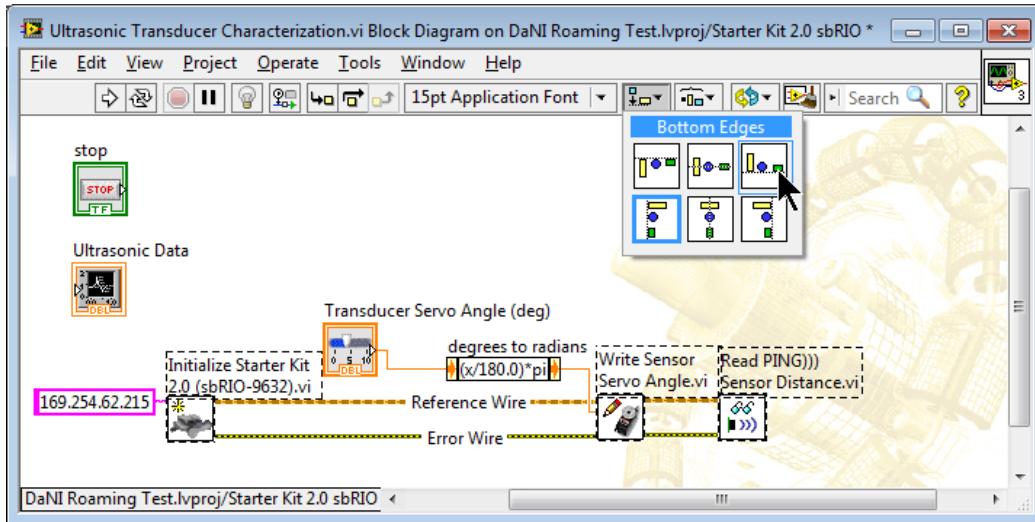
**Figure 0-26. degrees to radians conversion Expression Node on the block diagram**

Drag the slider icon into position and wire it to the input of the conversion node as shown in **Error! Reference source not found.**. Right click each icon, choose Visible Items and check label. Resize the label to compress the space as shown.



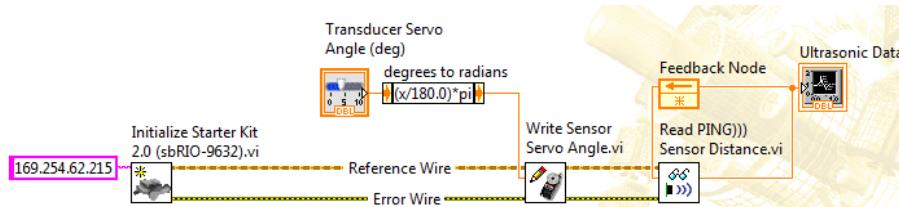
**Figure 0-27. Wire the slider to the input of the conversion node**

Next, place the Read PING))) Sensor Distance VI on the block diagram from the Functions>>Robotics>>Starter Kit>>2.0 palette. This VI communicates the ultrasonic transducer data from the FPGA to the processor. Wire the reference and error wires as shown in Figure 0-28. Using the Align button from the tool bar to align the bottom edges of the three icons as shown. You can select all three icons by pressing the shift key down while clicking each icon. This will facilitate creating a more readable diagram with straight wires. Show the Read VI's label.



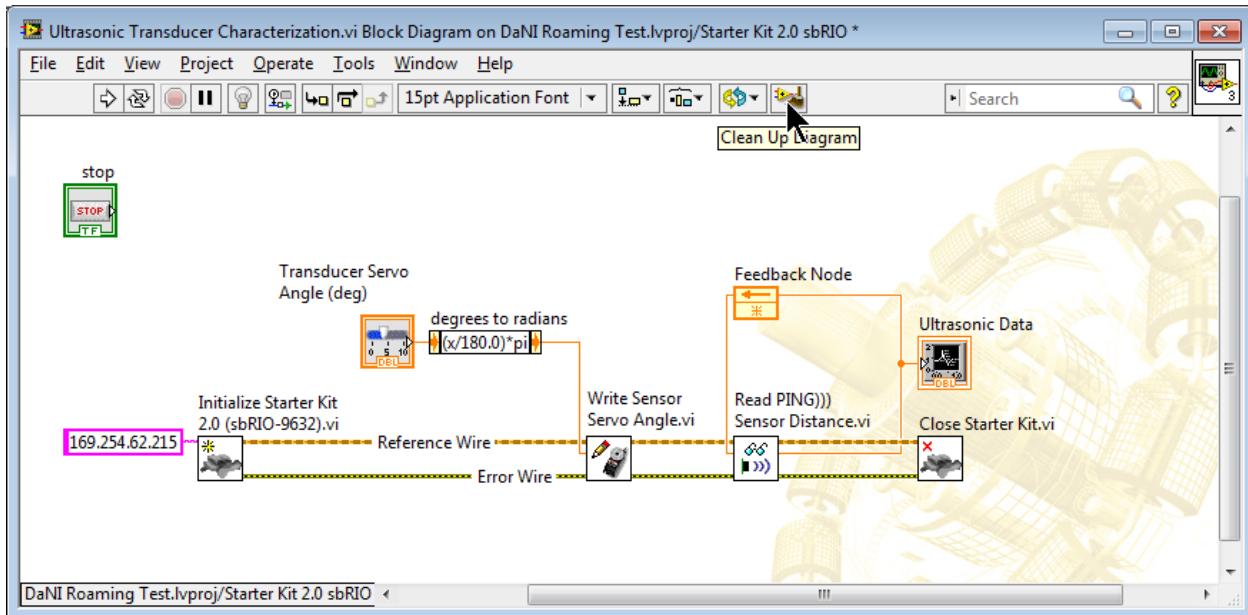
**Figure 0-28. Read PING))) Sensor Distance VI on the block diagram**

Show the help window for the Read VI and study the input and output connections. Reposition the Ultrasonic Data icon that represents the front panel chart. Wire the Distance (m) output of the Read VI to the chart as shown in Figure 0-29. Wire from the output of the Read VI to the input of the Read VI and a feedback node will automatically be inserted. Right click the feedback node and choose Change Direction and organize the wiring as shown. The purpose of the input that the feedback node is wired to is to provide a default value in case the FPGA did not provide a value.



**Figure 0-29. Wire from distance (m) output connection of the Read PING))) Sensor Distance VI to the chart and add a feedback wire to the default distand (NaN) input connection**

You should always close references to make the memory available to other programs and to prevent memory leaks that accumulate reserved memory. So the next icon to place on the block diagram is the Close Starter Kit VI, available from the Functions>>Robotics>>Starter Kit>>2.0 palette. Use show help to study the input and output connections. Wire the reference and error wires to the Close VI input connections as shown in Figure 0-30. This is also a good time to experience the block diagram clean up tool in the tool bar.



**Figure 0-30. Close the reference with the Close Starter Kit VI and use the Clean up Diagram tool**

The program will run on the sbRIO and communicate data from the FPGA to the processor on the sbRIO. Save it and save the project. The program was developed on the host computer and is saved in a folder on the host hard drive. When you run the program, it will automatically be deployed to the sbRIO. There is another set of code that automatically transfers the data over the Ethernet from the sbRIO to the host that is called Front Panel Communication. So while the sbRIO is connected to the host, you can view the front panel on the host monitor. You can think of it as running the block diagram on the sbRIO, running the front panel on the host computer, and communicating data between the two over the Ethernet cable.

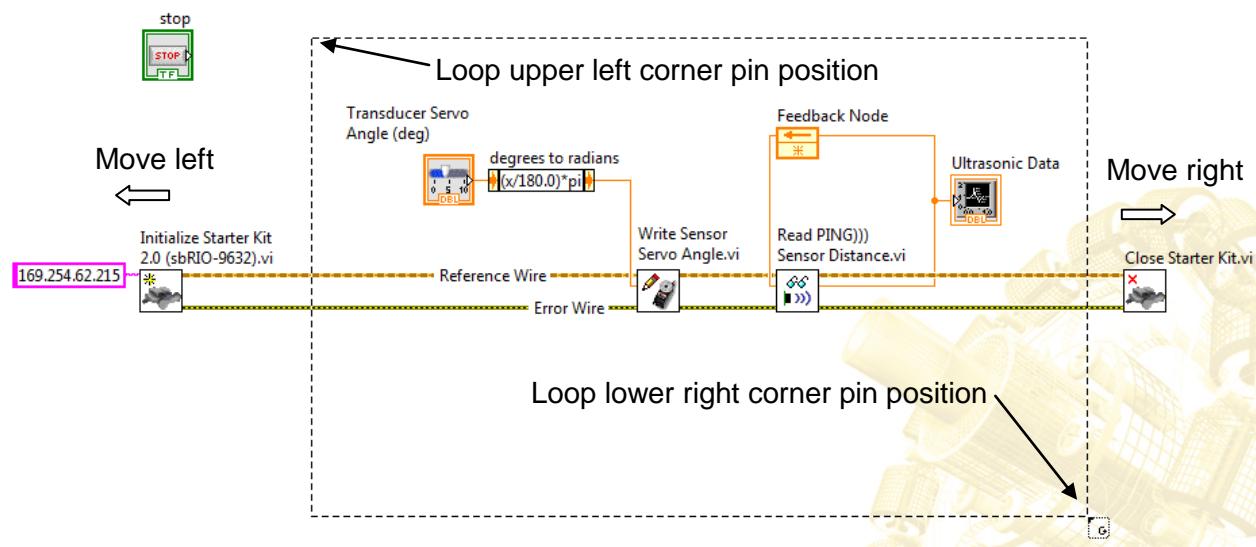
You have not finished the program yet, but you have code that will execute and can be tested. Test the program by running it and placing an object at a known distance from PING))). It is always a good idea to test code incrementally while you build programs. That way, you can more easily isolate and correct errors.

The program developed so far will only transfer one data point. You will add features in the following to iterate the code until you press the Stop button. So the Stop button is not used at this time.

It is much easier to characterize the transducer if you don't have to continually press the Run button to get a data point. You could use the run continuously and abort buttons, but as explained in Experiment 1 that is not a good option. The best way to repeat code execution is with one of LabVIEW's loop structures. You will use a structure called a While Loop. Place the code that is to be repeated inside the loop. Before adding the loop to the block diagram, make some space on the block diagram for the loop as shown in Figure 0-31 by moving the Initialize

VI to the left and the Close VI to the right. You can move them by dragging with the mouse or by selecting an icon and hitting an arrow key. Hold the shift key down while pressing the arrow key to move larger increments.

The position of the loop is shown by the dashed-line rectangle in the figure. It is not necessary to iterate initialize and close, in fact, it is reduces performance to do so. The program should initialize once and close once so those VIs should not be inside the loop. Select the While loop from the Functions>>Structures palette. The cursor changes to the dotted-line square shape with a darkened upper left corner as shown in Figure 0-31. Move the cursor to the pin position indicated in the figure. Left click to pin the upper left corner of the loop at the pin position. Then move the cursor to the location for the lower right corner of the loop.



**Figure 0-31. While loop location on the block diagram**

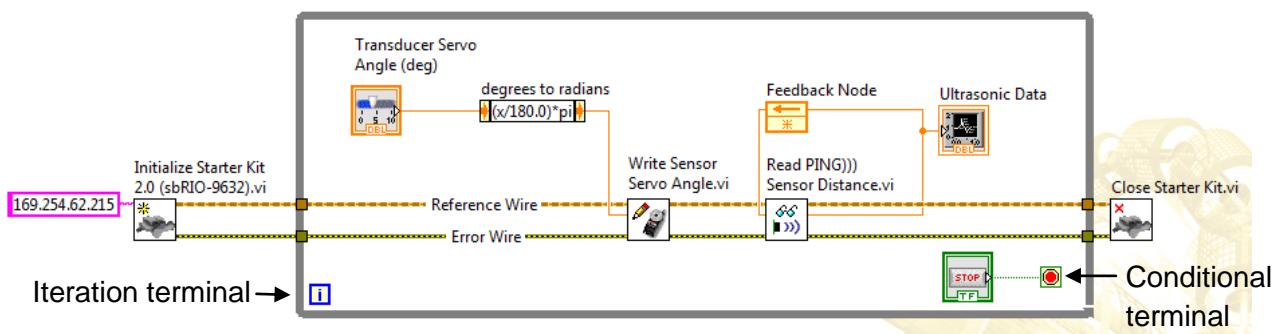
Figure 0-32 shows the block diagram with the While Loop border enclosing the code to be repeated. The loop executes the code inside its border repeatedly until stopped. The loop structure includes the iteration and Conditional terminals as shown. Study the LabVIEW Help for more information about the While loop.

You created a Stop button control on the front panel to stop execution of the Loop. The Stop control data should flow to the termination condition of the loop. The loop will continue to execute until the condition is true. Similar text pseudo code would look like this:

```
while (STOP is not pressed)
{
    //do what's in the loop
}
```

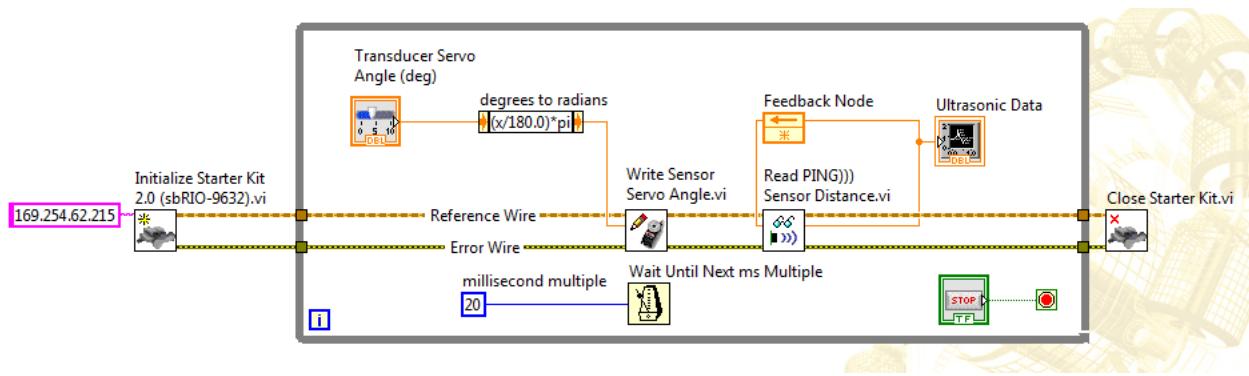
To implement the stop functionality, select the stop icon on the block diagram and drag it into the loop as shown in and wire it to the loop Conditional terminal.

Note that the Stop icon does not receive any inputs in the block diagram. Its value depends on the user. If the user presses the button, its value changes from False (the default) to True. When the Conditional terminal receives a true value, it terminates loop execution. Since the Stop icon doesn't have any inputs, its execution order is not obvious. It will execute after the code in the loop border executes. This assures that the code in the loop will always execute once. So the Conditional Terminal executes prior to the next iteration of the loop. If it receives a False value, the loop will iterate. If it receives a True value, execution terminates and the code following the loop, i.e. the Close VI will execute.



**Figure 0-32. The loop border encloses the code that will be repeated**

If you don't cause the loop to wait between iterations, it can monopolize all of the processor resources. LabVIEW has several wait functions you can use to solve this problem. Add the Wait Until Next function as shown in Figure 0-32 and create a constant of 20 ms so the loop will pause execution for a duration of 20 ms at every iteration. This is called Execution Control Timing.



**Figure 0-33. Characterization VI block diagram**

This completes the Characterization VI, so save it and save the project. Connect the Ethernet cable to DaNI. Turn the DaNI Master Switch on and the Motor Switch off. Right click the sbRIO

item in the project and connect. Click the Run arrow on the Characterization VI. The application will download and run, and the front panel will begin to update. To test the VI, move your hand back and forth in the ultrasonic transducer's field of view and watch the corresponding value change on the chart. The chart should display results similar to Figure 0-34. Press the Stop button when you finish, but do not close the VI, you will use it in the next section of the experiment to continue characterizing the transducer.

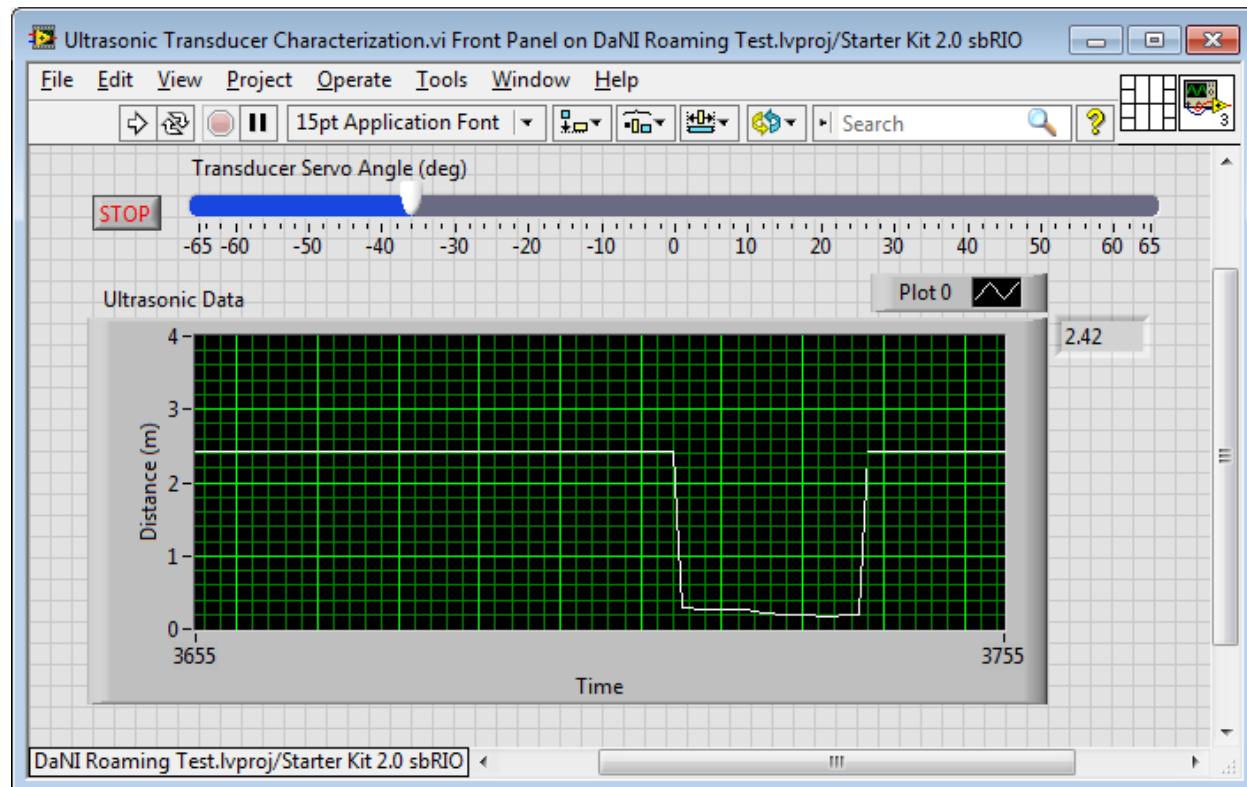


Figure 0-34. Results from moving a hand in the ultrasonic transducer field of view

## Experiment 2-3 Ultrasonic Transducer Characterization

To better understand the ultrasonic transducer, it is useful to understand what code executes on the FPGA. As you know from the investigations in this experiment, an ultrasonic transducer measures distance to an object by transmitting ultrasonic energy. The transmitted energy might contact an object that will reflect it back to the transducer. So after transmitting, the transducer becomes a receiver. Many surfaces can reflect ultrasonic energy at different reflection strengths. Some surfaces absorb most of the energy and reflect little. The receiver needs to incorporate a threshold to distinguish the reflection from an object from noise. So a received signal must be strong enough to exceed the threshold.

The transducer measures the time between transmission and receiving a valid signal. Time-of-flight is the time that elapsed between emitting a packet of (ultrasonic) pressure waves, and receiving the reflection. The time is measured at the FPGA in ticks of the FPGA's 40 MHz clock, meaning each 1 tick is equal to 25 nanoseconds. The FPGA code converts the time to distance at standard room temperature. The FPGA converts ticks to time-of-flight data and then to distance in meters. The FPGA implements equation below:

$$d = c * t / 2$$

where

$d$  = distance of object causing reflection of pressure waves

$c$  = speed of sound in air = 343 m/s at standard pressure and 20°C

$t$  = round trip time-of-flight

The FPGA receives a digital signal on DIO6 (refer to Figure 2-1). A digital signal has two discrete levels—a high and a low level, using the normal standard for TTL or transistor to transistor logic, the low level is 0 – 0.8 V and the high level is 2 – 5 V.

The signal from PING))) goes high during the transmit burst as shown in Figure 0-35. It stays high until a valid reflected signal is received. The FPGA records the total number of ticks between when the signal went high and when it drops. The number of ticks is converted to time which is converted to round trip distance with the above information. The FPGA divides the round trip distance by 2 and reports the distance to the object to the processor via the Read PING))) Sensor Distance VI.

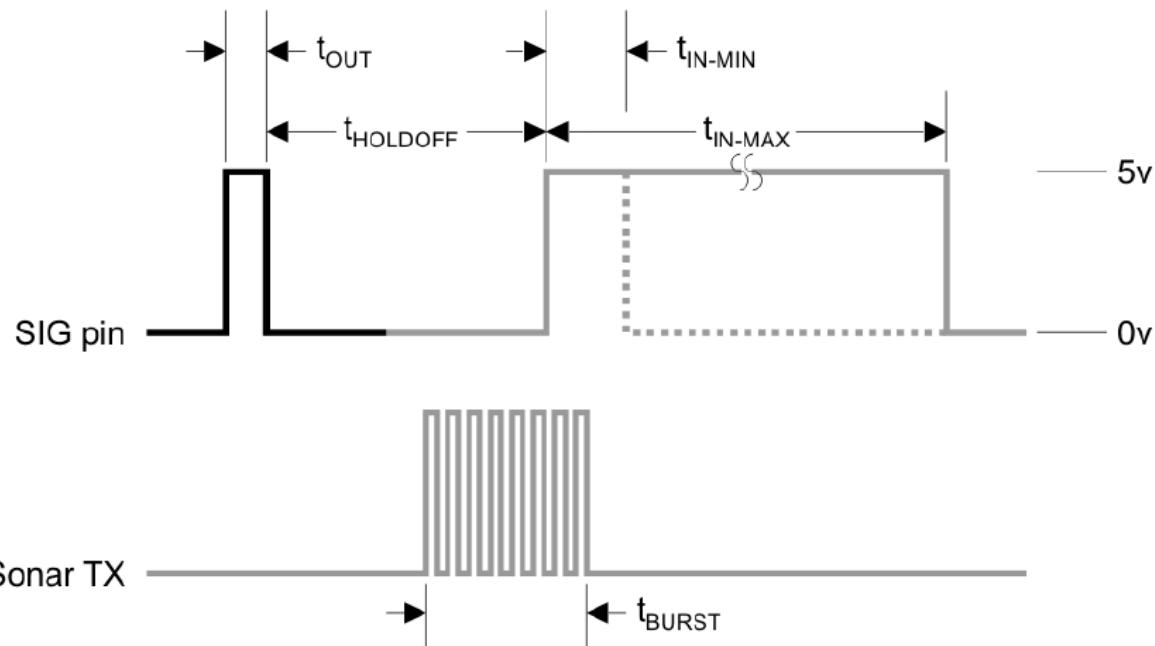


Figure 0-35. Transducer timing graphic

The Read VI communicates the distance from the FPGA to the processor. The ultrasonic characterization VI that you created calls the Read VI and requires you to interpolate the graph data, so repeat the characterization done previously with your ultrasonic characterization VI.

Since the ultrasonic characterization VI can measure distance without panning, you can evaluate the field of view (FOV) easier. Study the information about FOV in the specs on the Parallax web site. What is maximum field of view (FOV) of the transducer? Does the FOV vary with distance from the transducer?

Measure the FOV and compare with the website information. Place an object like a cardboard box 3m away from PING))). Orient it perpendicular to the transmission with the box side centered at the transmission. Carefully slide the box to one side, while maintaining the 3-m distance, until PING))) no longer reports an echo. The box has moved out of the FOV at this point. Record the distance from the transmission center line to the vanishing point. Repeat in the other direction and report the width of the FOV at 3m. Repeat at 0.5 m intervals from 3m to 0.5 m. Draw a plan view of the x distance from the transducer by the y distance signal vanishing point to the edge of the box to report how the FOV changes with distance from PING))).

The FOV is really 3D. It is conical. What does this mean in terms of effects from the surface on which DaNI travels? Consider driving to the edge of the floor surface, the height of the transducer, and the vertical angle of the transducer in your response.

Place several objects in the FOV. What distance was reported (average, closest, most far, most reflective, or?).



# **Experiment 3 – Motor Control**

## **Instructor's Notes**

This experiment requires that the previous experiments be completed. Similar experimental area and tools used in the previous experiment are used here.

## **Goal**

Discover how to control motors. Learn about motor drivers, PWM, and PID.

Experiment with and characterize an encoder.

Learn about VIs to control motors and how to build simple motor control programs. Learn how to automatically terminate programs. Learn about conversions between motor command units and user preferred units.

## **Required Components**

Test area similar to previous experiments.

Linear distance measuring tool like a ruler, meter stick, or tape measure.

Angle measuring tool like a protractor.

## **Background**

Students should study the Pittsco website material that describes the motors and encoders and the Dimension Engineering website material that describes the motor driver.

Students should study the section on optical encoders in Siegwart et al (2011) chapter 4

## **Experiment 3-1 Open Loop Motor Control**

You discovered how the ultrasonic transducer signal can be used to locate obstacles in the previous experiment. This experiment discovers how to control DaNI's motors.

To become more familiar with the two drive motors, study the information on the Pittsco web site for the TETRIX DC Drive motor, part number 39083. Look at the relationship between the

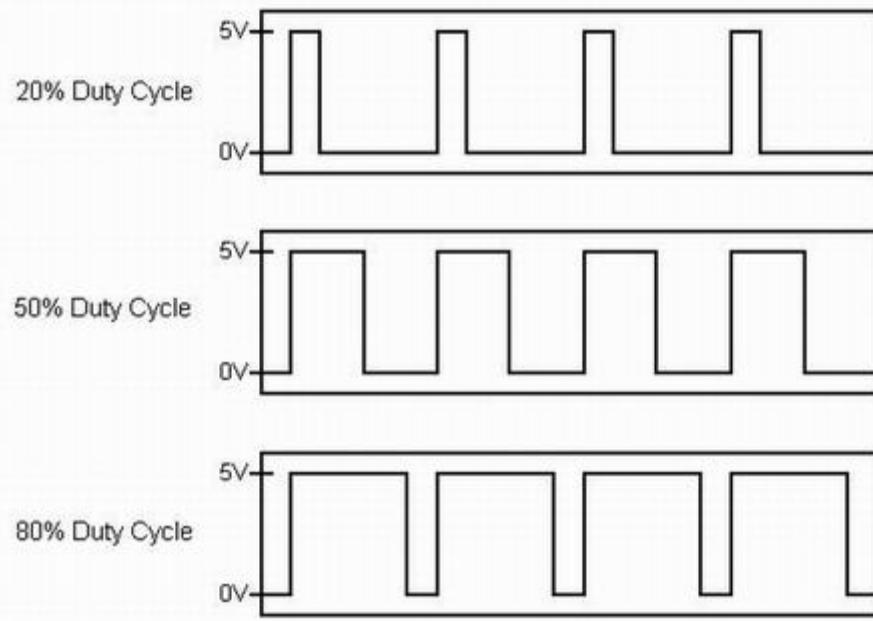
current, torque, efficiency vs. speed on the chart in the specifications. Note that the current range required is 0 - 4.66 A and stall current is 4.6 A. Then, study the sbRIO specifications noting that the current output is only 3 mA.

Refer to Figure 2-1 and note that the Sabertooth 2x10 RC motor driver is included in the DaNI components to provide the power needed to drive the motors. The motor driver connects to both the batteries and to the sbRIO. The program you write tells the motor driver how much power to provide from the batteries to the motors. Study the motor driver specifications on the Dimension Engineering web site. It can supply 10A each to two DC motors.

Note that Figure 2-1 shows signals from DIO4 go to the left motor and DIO5 go to the right motor. Digital input and output signals are either high (on) or low (off) as described in the previous experiment, but you will want to have more than two choices for velocity. You would like to drive slowly sometimes and quickly at others, so you would like to continuously vary the speed of the motors within a safe range. How can you write a program that will continuously control the speed of the motors with a digital output?

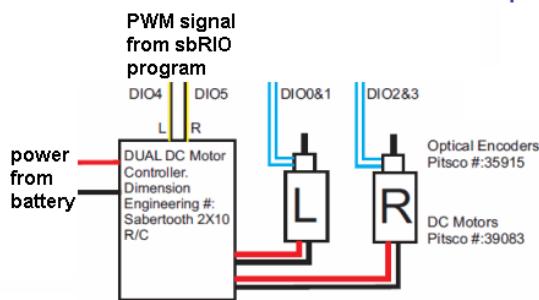
Pulse width modulation (PWM) is a technique in which a series of digital pulses is used to control an analog circuit. The length and frequency of these pulses determines the total power delivered to the circuit. PWM signals are most commonly used to control DC motors, but have many other applications ranging from controlling valves or pumps to adjusting the brightness of an LED.

The digital pulse train that makes up a PWM signal has a fixed frequency and varies the pulse width to alter the average power of the signal. The ratio of the pulse width to the period is referred to as the duty cycle of the signal. For example, if a PWM signal has a 10 ms period and its pulses are 2 ms long, that signal is said to have a 20 percent duty cycle as shown in Figure 0-1, which shows three PWM signals with different duty cycles.



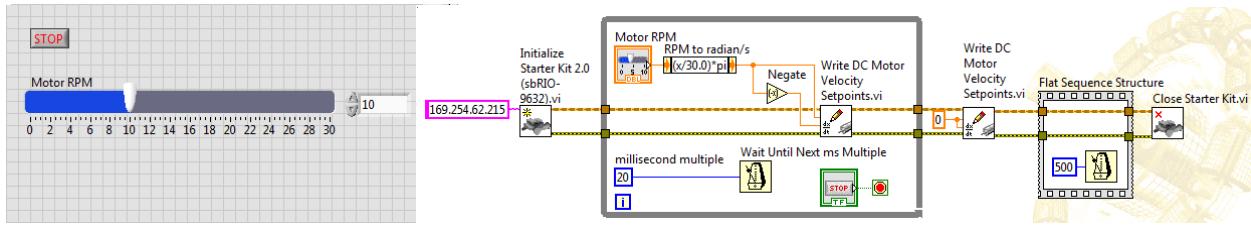
**Figure 0-1. PWM Signals with Different Duty Cycles**

Figure 0-2 shows the block diagram of the motor control circuit where the motor driver controls the power from the battery to the motors using commands from a software program running on the sbRIO.

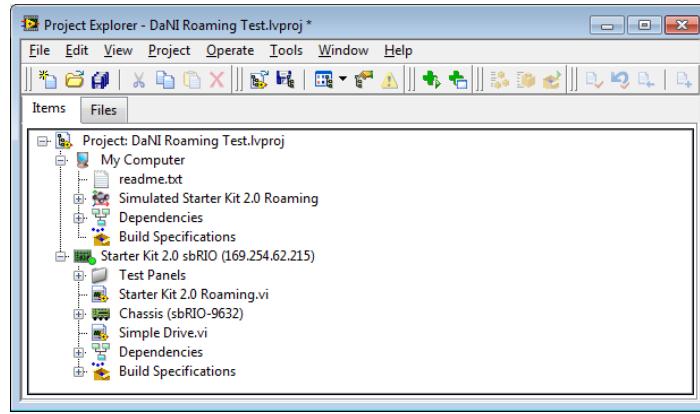


**Figure 0-2 motor control circuit block diagram**

The motor control software has already been written for you and resides in the FPGA bitfile that is loaded from the project. You just have to build a program similar to the ultrasonic characterization VI developed in the previous experiment to drive the robot. Use the programming skills from the previous experiment and the instructions in the following to build the Simple Drive VI shown in Figure 0-3.



**Figure 0-3 Simple Drive VI front panel and block diagram**



**Figure 0-4 Simple Drive VI in the DaNI Roaming Test project**

You can open the Ultrasonic Characterization VI in the project, modify it, and save it with a different name, or you can copy it and modify the copy, or you can add a new VI to the project. If you choose to copy and modify the characterization VI, open the VI and choose **File>>Save As**. The Save As dialog box opens. Click the **Copy** radio button. This creates and saves a copy of the file in memory to disk with a name you choose. If you enter a new file path or name for the file, the original file on disk is not overwritten or deleted. You can choose between:

**Substitute copy for original**—Both the original file and the new file exist on disk, but the original VI closes and the new VI opens. Use this option if you want to create a copy of the original file and immediately edit the copy. If the original file is in a project library or project, this option substitutes the new file for the original in the project library or project.

**Create unopened disk copy**—Both the original file and the new file exist on disk, but only the original file remains open in memory. Use this option if you want to create a copy of the original file but continue editing the original file, for example, if you want to create a backup copy. If the original file is in a library or project, this option does not add the new file to the same library or project.

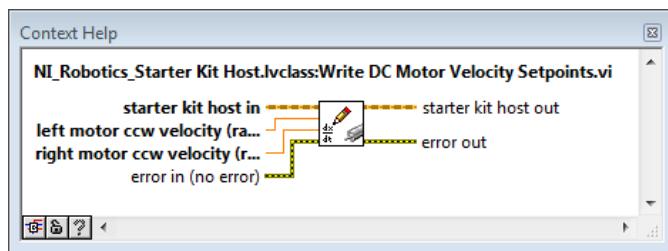
**Open additional copy**—Both the original file and the new file exist on disk and are open in memory. You must give the copy a new name, because two files of the same name cannot exist in the same application instance at the same time. Use this option if you want to create a copy of the original file, continue editing the original file, and also immediately edit the new file. If the

original file is in a library or project, you have the option of adding the new file to the same library or project by placing checkmarks in the appropriate checkboxes.

The main difference between the Simple Drive VI and the Ultrasonic Transducer Characterization VI is the Write DC Motor Velocity Setpoints VI. If you are editing a copy of the characterization VI, delete the servo control and ultrasonic read VIs and add two copies of the Write Setpoints VI, one in the loop and one after the loop as shown.

Show the context help for the Write Setpoints VI as shown in Figure 0-5. This VI has the same reference and error in and out connections as the Vis used in the previous experiment, the difference is that there are two velocity inputs and no additional outputs. Both velocity inputs have the same units, radians/second (even though it isn't visible in the context help window) indicating rotational velocities. Wire a constant of 0 to the inputs of the Write Setpoints VI after the loop to set the velocity to 0 before terminating the program. Use a front panel control so users can choose the velocity for the VI in the loop.

Users are more familiar with rotational velocity in RPM, so search the functions palette for an expression node that will convert RPM to radians/second and add it to the block diagram. Then create a slider control customized for rotational velocity in RPM on the front panel and connect it to the conversion node as shown in Figure 0-3. The range of the slider control should be 0 to 30 RPM. Wire the conversion node to the input of the left motor ccw velocity input. Before wiring the right motor ccw velocity, search the functions palette for and add a negate function as shown in Figure 0-3. The reason for the negate function is that the ccw acronym in the connector name means counter clockwise. It is counter clockwise looking at the outer portion of the wheel, so if the right wheel turns counter clockwise, it will reverse the robot. Consequently, the velocity of one wheel is in the opposite direction of the other, so to drive the robot straight ahead, you need to send a negative value to the left motor.



**Figure 0-5 Write DC Motor Velocity Setpoints VI context help**

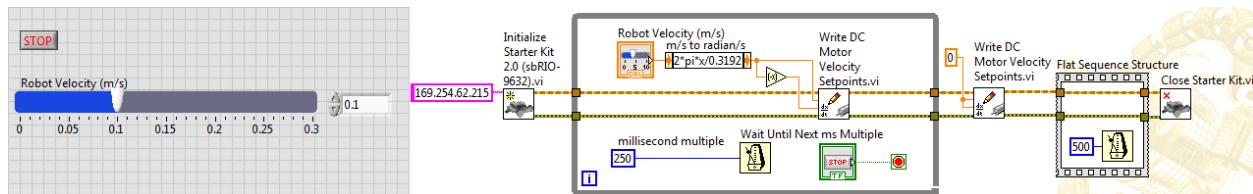
The other important change is a time delay before closing the reference. Add a single-frame Flat Sequence Structure after setting the motor velocities to 0. A Flat Sequence structure consists of one or more subdiagrams, or frames, that execute sequentially. A single flat sequence structure to force sequencing of a VI that doesn't have error terminals is necessary sometimes, but don't use flat or stacked sequence structures in your code when you can avoid them. When used incorrectly, they will slow execution and make the code difficult to read. Study the LabVIEW help for more information on the flat sequence structure.

Put a Wait Until Next Time Delay function in the structure with a constant of 500 ms. This delay will give the system time to set the motor velocities to 0 before the reference is deleted. Without it, the reference might be deleted before the motors are set to 0 and the robot will continue to run after the stop button is pressed. This is not intuitive as LabVIEW data flow requires that the motor velocity setpoint VI be executed before the close VI. But the motor velocity setpoint VI initiates communications through the sbRIO to the FPGA, to the motor driver, and finally to the motors, which takes several milliseconds. If data flow passes execution control to the close VI before this series of activities completes, the reference is destroyed and the activity terminates before completion.

In some situations you can create sequential code without using a sequence structure by wiring between icons. But, in this case the Wait function does not have any reference or error connections for sequential wiring.

Save the program and the project. Test the program. Turn on the Master and Motor Switches. Tether the robot to the network or to the host. Test at low speed so you don't collide with an obstacle and damage the robot. Mark a straight line of about 2 m or use a string or tape and determine the error in driving straight over the same distance at 5 RPM, 10 RPM, and 15 RPM.

The user might be more interested in giving the robot a linear velocity instead of a rotational velocity. Modify the Simple Drive VI as shown in Figure 0-6 using the information in the following.



**Figure 0-6 Simple Drive Linear VI front panel and block diagram**

Edit the existing RPM to rad/s Expression Node, or delete it and add an empty expression node from the functions palette. Determine the wheel circumference, and develop a new conversion equation that converts linear velocity of the robot in m/s to wheel rotational velocity in rad/s. Type it into the expression node. Modify the slider scale and label to travel at a linear velocity equivalent of about 30 RPM. Save the VI.

Circumference =  $\pi d = 12.5664 \text{ in} = 0.3192 \text{ m}$   
where  $d = 4 \text{ in}$

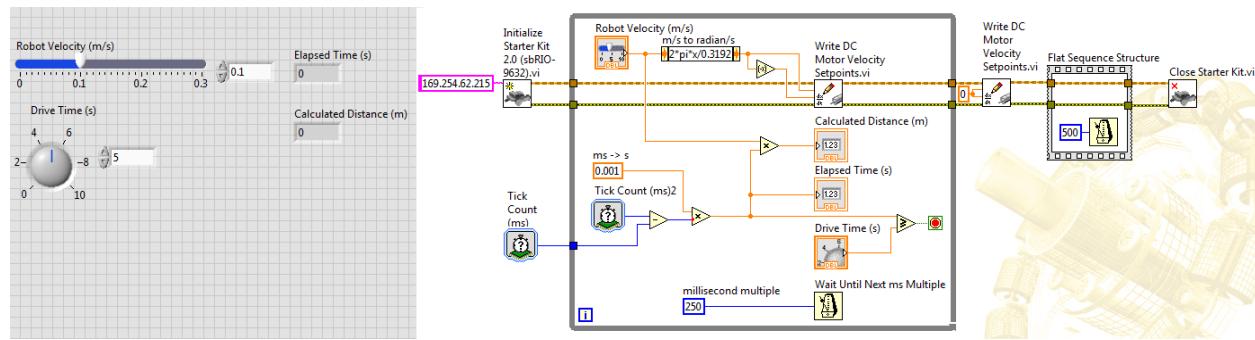
linear robot velocity (m/s) =  $0.3192 \text{ m/rotation} * \text{rotational wheel velocity (rad/s)} / 2\pi \text{ rad/rotation} / 60 \text{ s/min}$

rotational wheel velocity (rad/s) =  $2\pi$  rad/rotation \* linear robot velocity (m/s) / 0.3192 m/rotation

Slider scale maximum value (m/s) = 0.3192 m/rotation \* 30 rotations/min /  $2\pi$  rad/rotation / 60 s/min = 0.25 m/s

Save the program and the project. Test the program. Turn on the Master and Motor Switches. Tether the robot to the network or to the host. Test at low speed so you don't collide with an obstacle and damage the robot.

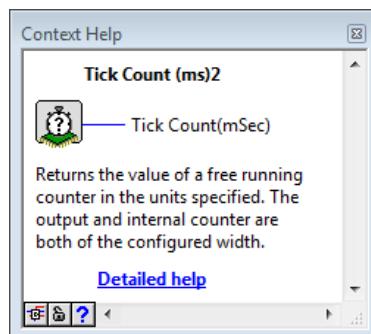
You can stop the VI automatically so you don't have to use the Stop button. Modify the Simple Drive Linear VI so it automatically stops after N seconds where the user enters a value for N as shown in Figure 0-7 and described in the following.



**Figure 0-7 Simple Drive Timed VI front panel and block diagram**

To do this, you need to measure the elapsed time and compare it with the user's input. Since the elapsed time is available, you can multiply it by the user's desired velocity and calculate the distance moved.

The main modification of the previous program is the Tick Count VI that measures time. Figure 0-8 shows the context help window for the Tick Count VI. Detailed help is available for this VI (see Figure 0-9) by clicking on the Detailed Help link or the ? in the context help window. This is the same information that is available by selecting LabVIEW Help from the Help item in the menu bar and searching for the Tick Count VI.



**Figure 0-8. Tick Count context help window**

**Tick Count Express VI**

Owning Palette: [Execution Control Express VIs and Structures](#)

Requires: DSP Module, FPGA Module, or Real-Time Module

Returns the value of a free running counter in the units specified. The output and internal counter are both of the configured width.

If you do not have the FPGA, Real-Time, or DSP modules, you can use the [Tick Count \(ms\)](#) function instead.

[Dialog Box Options](#)

[Block Diagram Outputs](#)

 Add to the block diagram  Find on the palette

**Dialog Box Options**

Parameter	Description
<b>Counter Units</b>	Unit of time the VI uses for the counter.  <b>Note (Windows)</b> The Windows operating system supports only millisecond resolution. If you select a <b>µSec</b> or <b>Tick</b> resolution under Windows, the VI does not provide exact timing and instead provides an approximation that averages to the requested time over the course of the timing interval. Windows does not support resolutions under one millisecond and rounds them up to one millisecond. <ul style="list-style-type: none"><li>– <b>Ticks</b>—Sets the counter units to a single clock cycle, the length of which is determined by the clock rate for which the VI is compiled.</li><li>– <b>µSec</b>—Sets the counter units to microseconds.</li><li>– <b>mSec</b>—Sets the counter units to milliseconds.</li></ul>
<b>Size of Internal Counter</b>	Specifies the maximum time a timer can track. To save space on the FPGA, use the smallest <b>Size of Internal Counter</b> possible for the FPGA VI.

**Block Diagram Outputs**

Parameter	Description
<b>Tick Count</b>	Returns the value of a free running counter at the time the VI wakes up. A free running counter rolls over when the counter reaches the maximum of <b>Size of Internal Counter</b> specified in the configuration dialog box.

**Figure 0-9. Tick Count detailed help window**

When you place the Tick Count VI on the block diagram, since it is an Express VI, a Dialog box opens automatically to configure it for either ticks, µsec, or msec. An Express VI is a VI whose settings you can configure interactively through a dialog box. The problem is that the configuration is not readily apparent on the block diagram reducing readability of the code. So always add a free label (by double clicking on the block diagram) or a label to indicate the configuration as shown in Figure 0-7 where the label reads Tick Count (ms) to show the VI was configured for ms units instead of ticks or µsec. Since this VI runs on the sbRIO in the real time operating systems and not on the MS Windows host, any of the three options are available to you. Use ms since its resolution is adequate for the purposes of this experiment.

Timing is precise with the real time OS on the sbRIO. Code runs deterministically in the real time OS. Real time means that responses occur in time, or on time. With non-real-time systems like MS Windows, there is no way to ensure that a response occurs within any time period, and operations may finish much later or earlier than expected. In other words, real-time systems are deterministic, which guarantees that operations occur within a given time. Real-time systems are predictable.

For a system to be a real-time system, all parts of it need to be real time. For instance, even though a program runs in a real-time operating system, it does not mean that the program

behaves with real-time characteristics. The program may rely on something that does not behave in real-time such as file I/O, which then causes the program to not behave in real-time.

Place a Tick Count Express VI from the Functions>>Real Time>>Timing palette in front of the loop and another inside the loop as shown in Figure 0-7. The VI in front of the loop will execute once when the program starts and will provide a base time. The base time value will flow through a tunnel in the loop and be available inside the loop. The time value from the VI inside the loop will change (update) at each iteration of the loop. By subtracting the two, you will measure the elapsed time. Create an indicator for the front panel to display the elapsed time.

Add a control to the front panel for the user to enter the drive time in seconds. Since the user enters drive time in seconds, convert the Tick count output in ms to s by multiplying by 0.001. This operation uses two different numeric data types. The Tick Count output is an unsigned long 32-bit integer, or U32, whose wire color is blue. To create a constant of 0.001, you need a floating point number instead of an integer. The most common is a double-precision (64-bit) floating point data type, or DBL, whose wire color is orange. You can create this type of constant by right clicking on one of the orange wires in the diagram and choosing Create>>Constant from the short cut menu. You can also get a DBL numeric constant from the functions palette. Change the value from 0 to 0.001 and drag it to the correct location.

If you wire two different numeric data types to a numeric function, like the multiply function used in this VI, LabVIEW does not create an error, instead, to reduce your programming effort, it automatically converts one of the terminals to the same representation as the other terminal. This action is called coercion. LabVIEW chooses the representation that uses more bits. If the number of bits is the same, LabVIEW chooses unsigned over signed. So in this instance, LabVIEW converts the U32 to a DBL. Coercion dots appear on block diagram nodes to alert you that LabVIEW converted the value passed into the node to a different representation. Coercion dots can cause a VI to use more memory and increase its run time. Try to keep data types consistent in the VIs you create. To use memory more efficiently, eliminate coercion dots at numeric terminals. Right-click the input value on the block diagram and select Representation from the shortcut menu to change the representation of the input value to the representation of the terminal. Leave the two different data types and the automatic conversion by LabVIEW in this case for instructional purposes.

Compare the elapsed time with the user's Drive Time using a Greater or Equal? Function from the functions palette and wire the Boolean data type output to the While loop conditional terminal to automatically stop the loop with the output is True.

Multiply the elapsed time by the user's robot linear velocity to calculate the distance and display it on the front panel with an indicator. Align the objects on the front panel and block diagram to improve readability.

Save the program and the project. Test the program. Turn on the Master and Motor Switches. Tether the robot to the network or to the host. Test at low speed so you don't collide with an

obstacle and damage the robot. Measure the distances moved at three different velocities and compare them with the calculated distances from the program. Explain the difference between the measured and calculated distance values.

## Experiment 3-2 Closed Loop Motor Control

The word “setpoint” in the name of the Velocity Setpoint VI, used in the previous section of this experiment, means that the inputs to the VI are target velocities of the two motors. When the target velocities are the same, the motors should have the same rotational velocities.

Therefore, the control system needs to know the value of the velocities and automatically send signals to the motors to maintain the target values. Velocity sensors called optical encoders connect to the computer via the blue wires extending to DIO 0&1 and DIO 2&3 in Figure 0-2.

An encoder is an electrical mechanical device that converts linear or rotary displacement into digital or pulse signals. An optical encoder uses a rotating disk, a light source, and a photodetector (light sensor). The disk is mounted on the rotating motor shaft and has patterns of opaque and transparent sectors. The encoder light source, pointed at a photodetector, passes through the disk sectors. As the disk rotates, these patterns interrupt the light emitted onto the photodetector, generating a digital or pulse signal output.

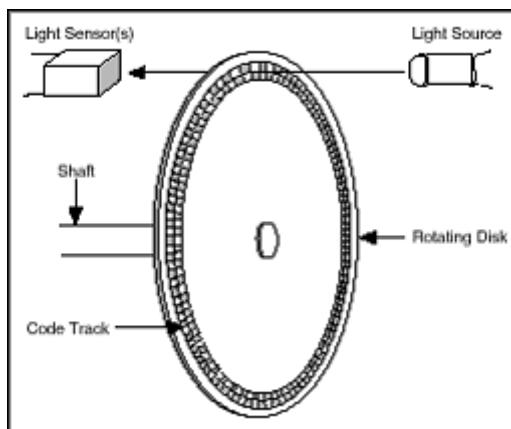
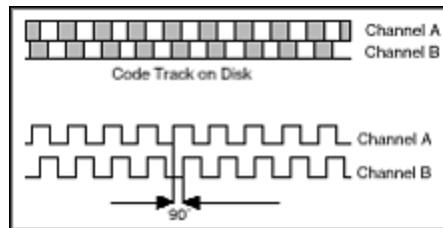


Figure 0-10. Optical Encoder

There are two general types of encoders - absolute and incremental encoders. An absolute encoder generates a unique word pattern for every position of the shaft. The tracks of the absolute encoder disk, generally four to six, commonly are coded to generate binary code, binary-coded decimal (BCD), or gray code outputs. An incremental encoder generates a pulse, as opposed to an entire digital word, for each incremental step. Although the incremental encoder does not output absolute position, it does provide more resolution at a lower price. For example, an incremental encoder with a single code track, referred to as a tachometer encoder, generates a pulse signal whose frequency indicates the velocity of displacement. However, the

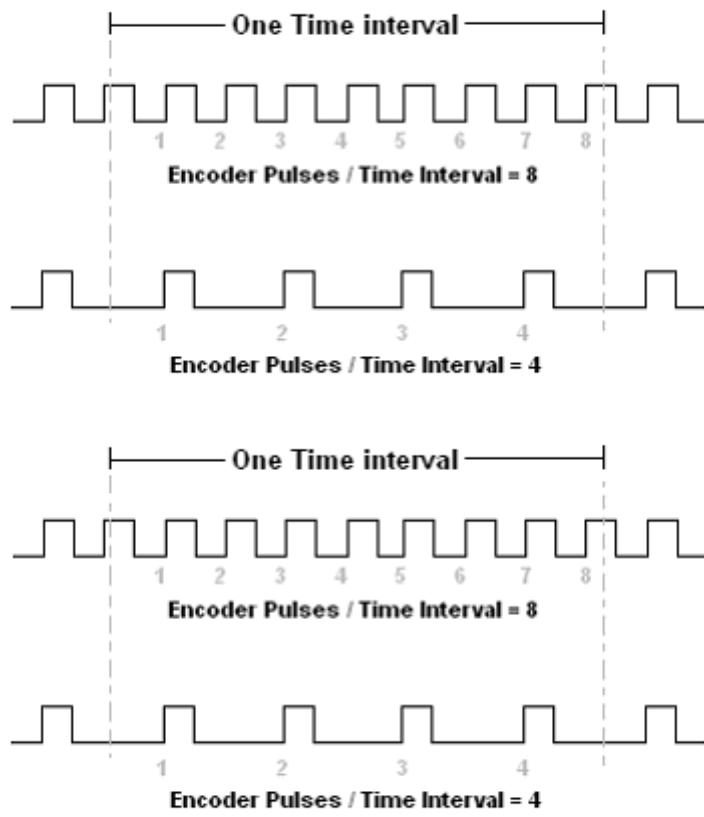
output of the single-channel encoder does not indicate direction. To determine direction, a two-channel, or quadrature, encoder uses two detectors and two code tracks.

The most common type of incremental encoder uses two output channels (A and B) and two code tracks with sectors positioned  $90^\circ$  out of phase, as shown in Figure 0-11. The two output channels indicate both position and direction of rotation. If A leads B, for example, the disk is rotating in a clockwise direction. If B leads A, then the disk is rotating in a counter-clockwise direction. Therefore, by monitoring both the number of pulses and the relative phase of signals A and B, you can track both the position and direction of rotation. Some quadrature detectors include a third output channel, called a zero or reference signal, which supplies a single pulse per revolution that can be used for precise determination of a reference position.



**Figure 0-11. Quadrature Encoder Output Channels A and B**

To determine the angular velocity and acceleration of a wheel from Quadrature Encoder signals you can count the number of pulses in a fixed time interval as shown in Figure 0-12.



**Figure 0-12. Velocity estimation with quadrature encoder signals**

Calculate the angular velocity with:

$$Velocity = \frac{\text{Encoder Pulses}}{\frac{\text{Pulses Per Revolution}}{\text{Fixed Time Interval (sec)}}} \times \frac{60 \text{ sec}}{1 \text{ min}} \text{ (RPM)}$$

Where, "Encoder Pulses" is the number of quadrature encoder pulses received in the Fixed Time Interval.

The following formula can be used to estimate acceleration:

$$Acceleration = \frac{\left( \frac{\text{Encoder Pulses}_n - \text{Encoder Pulses}_{n-1}}{\text{Pulses Per Revolution}} \right)}{\left( \text{Fixed Time Interval} \right)^2 \text{ (sec}^2\text{)}} \times \left( \frac{60 \text{ sec}}{1 \text{ min}} \right)^2 \text{ (RPM}^2\text{)}$$

The optical encoders on the Pitsco DC motors require a 5V supply and produce 100

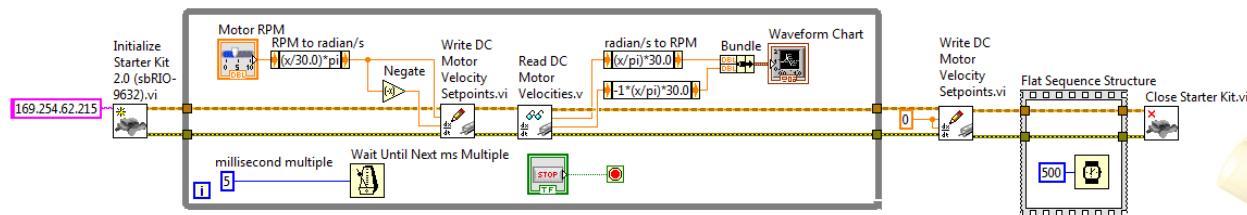
counts/revolution (CPR) and 400 pulses/revolution (PPR). Calculate the distance moved per count and pulse.

$$0.3192 \text{ m/revolution} / 100 \text{ counts/revolution} = 0.003192 \text{ m/count}$$

$$0.3192 \text{ m/revolution} / 400 \text{ pulses/revolution} = 0.000798 \text{ m/pulse}$$

Considering the 400 pulses/revolution and DaNi's wheel circumference the resolution in linear travel distance is 0.000798 m or 0.798 mm.

You can use the encoder information to develop a VI that reports the difference between the measured and calculated distances in the Simple Drive Timed VI. Modify the Simple Drive Linear VI developed above to graph the velocity measured by the encoders as shown in Figure 0-14.



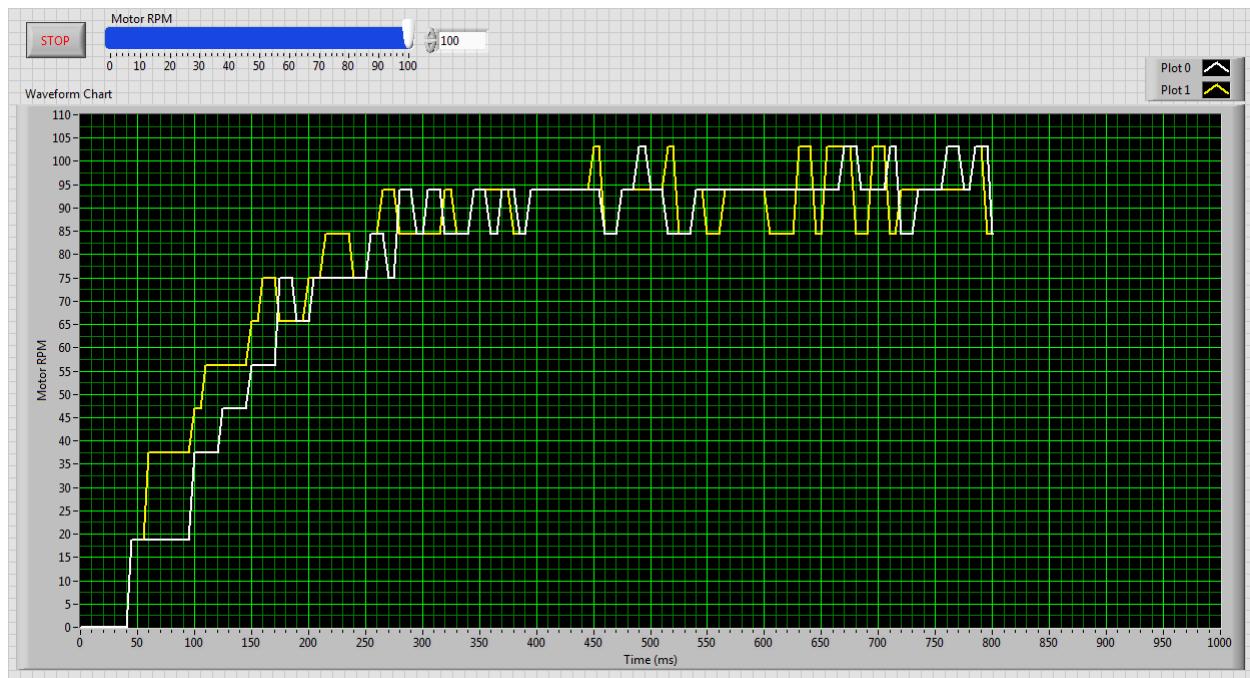
**Figure 0-13. Drive Velocity VI block diagram**

The major difference with the Simple Drive VI is adding the Read DC Motor Velocities VI from the functions>>Robotics>>Starter Kit>>2.0 palette. If you display the context help for the VI, you will see that the VI has the usual reference and error input and output connections. It also has left and right motor ccw in rads/s outputs. Search the functions palette for the radians/s to RPM expression node conversion. You can add the negate function to the right output or you can add a -1 multiplier to the expression node to accomplish the correction for the ccw direction as was done previously for the motor RPM to Write Motor Velocity Setpoint VI programming. To display both outputs on the same chart, add a Bundle Function by searching the functions palette. This function assembles the input elements into a cluster data type (that will be explained later). When the cluster is input to the chart, it displays the elements of the cluster as separate plots. Change the millisecond multiple constant to 5 to send velocity setpoint signals and sample the velocities faster to see the variability in the signal.

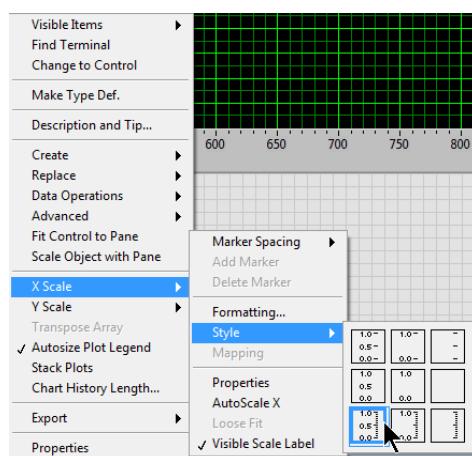
You will need to drive the robot faster than previous, so modify range of the slider on the front panel as shown in Figure 0-14. Customize the graph:

- Expand the plot legend to two plots by dragging the upper border of the plot legend one item higher
- Widen the plot lines by right clicking on each plot line and choosing Line Width
- Add a grid as done previously

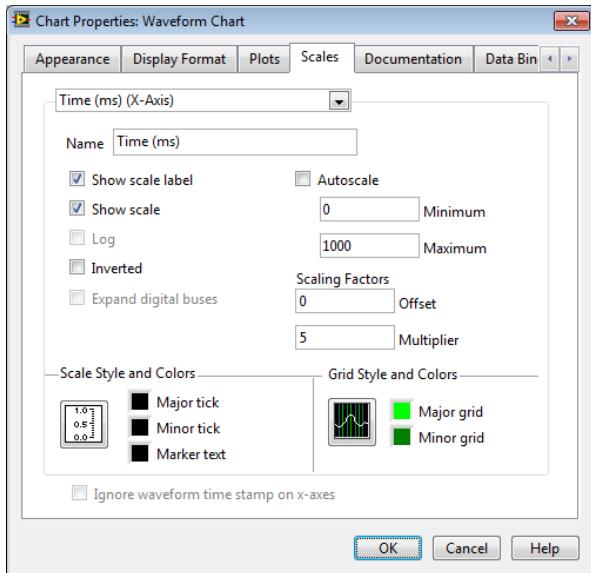
- Change the X scale style by right clicking the axis as shown in Figure 0-15, and
- Add a scale multiplier of 5 to convert sample number to time in ms by right clicking the graph and configuring the properties as shown in Figure 0-16. This is an approximation and not the exact  $\Delta t$  between points as it only accounts for the delay (wait) value and not the code execution time. A better value could be determined with the benchmarking code in the Simple Drive Timed VI, but the approximation is adequate for the purposes of this section of the experiment.



**Figure 0-14. Drive Velocity VI front panel**



**Figure 0-15. Chart X-Scale style configuration**



**Figure 0-16. Chart scale customization for the Drive Velocity VI**

Save the VI and the project. Run the VI tethered at a higher motor speed (e.g. 100 RPM) for a very short distance and observe the results in the graph. The results explain the error between the calculated and measured distances from running the Simple Drive Timed VI. The graph shows that velocity begins to rise at 40 ms and reaches 95 RPM at about 280 ms so the period of acceleration is about 200 ms although there is a small increase between 280 and 345 ms. If you approximate acceleration with a linear function:

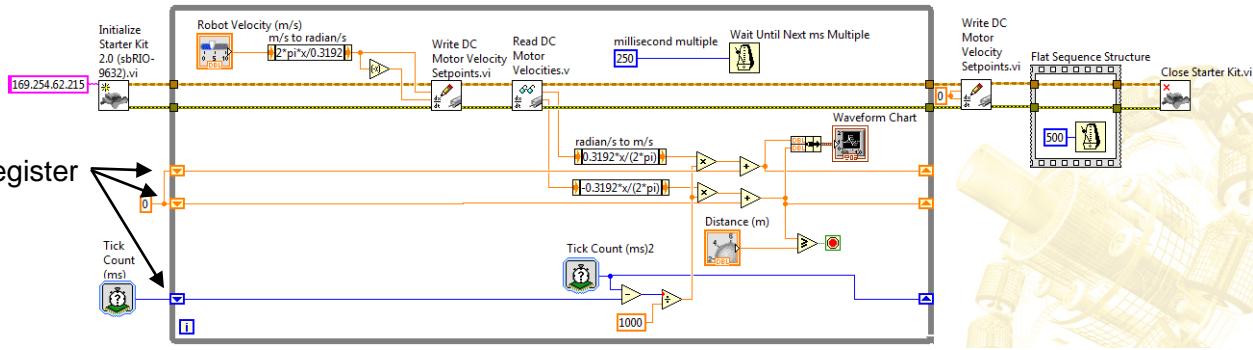
$$\Delta V / \Delta t \approx 2.5 \text{ m/s}^2$$

where  $\Delta V \approx 95 \text{ RPM} * 0.3192 \text{ m/revolution} / 60 \text{ s/min} = 0.5 \text{ m/s}$

$$\Delta t \approx 200 \text{ ms} = 0.2 \text{ s}$$

The calculated value in the Simple Drive Timed VI did not take this period into account. As the distance driven increases, this period would be a smaller portion and the error would decrease.

Modify the simple drive timed VI to stop automatically after a set distance measurement has been exceeded as shown in Figure 0-17 using the following information.

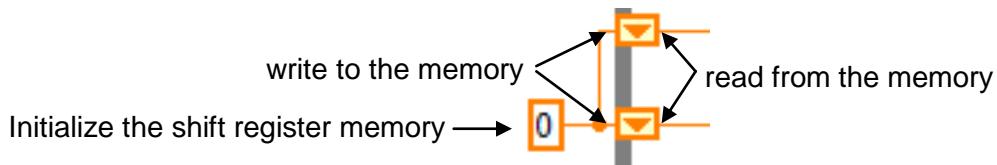


**Figure 0-17. Simple Drive Distance VI block diagram**

The main difference in this VI and the previous programs is the use of Shift Registers. Shift Registers reside on the borders of the loop. They are used instead of tunnels in this VI because the time and distances should be accumulated as the program executes. If tunnels were used, the values would reset and not accumulate. Create a Shift Register by right clicking the loop border and selecting Add Shift Register or by right clicking a tunnel and choosing Replace With Shift Register.

Shift Registers are very powerful tools because they give you access to computer memory. As shown in Figure 0-18, when you wire data to the left side of the Shift Register, you write the data to memory. When you wire from the right side of the Shift Register, you read the most recent data from the memory. When you write data to the Shift Register, you replace the data that was previously there. When you create a Shift Register on one side of a loop, another Shift Register icon appears on the other side automatically, or the cursor takes the shift register shape asking you to click on the opposite side of the loop to place the second icon. The two icons, one on the right border and one of the left, represent the same memory. So, when you wire (write) a new value to the right memory icon, that value is available to be wired (read) from on the left icon after the loop iterates. Consequently, you can think of Shift Registers as remembering the data from the previous iteration of the loop.

It is usually a good idea to initialize the memory as shown in Figure 0-18 otherwise you might read a value from a previous execution of the program. Initialize the distance memories with the value 0, and initialize the time memory with the program start time. The accumulated tick counts are subtracted from the current tick count to determine the elapsed time for the current iteration. The elapsed time for the current iteration is multiplied by the current velocity measured by the encoders to calculate the distance traveled during the current iteration. The distance traveled in the current iteration is added to the previous total distance for each encoder and written to the Shift Register on the right side of the loop. The distance for each encoder is accumulated in the Shift Registers and the right encoder distance is compared to the user defined stop target. The average, the maximum, or the minimum could have been compared as well.



**Figure 0-18. Shift Registers**

Save the VI and save the project. Run the VI with DaNI tethered to the host. Figure 0-19 shows the front panel and example results graph of driving at 0.1 m/s for a 1 m target distance.



**Figure 0-19. Simple Drive Distance VI front panel**

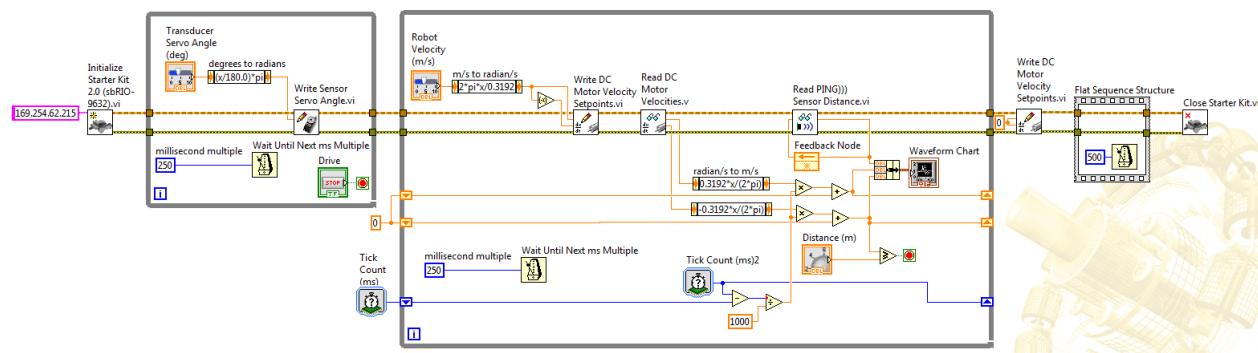
Use the VIs developed to characterize the encoders. Set a straight course of three different lengths, say 1 m , 2 m, and 3 m. Mark it on the floor with string or tape. Set DaNI at the start of the course and oriented to drive straight along the course. Set the stop distance at each of the three lengths. The X distance is the direction along the length or planned direction of travel. The Y distance is perpendicular to the length, or planned direction of travel. Run the VI so DaNI will drive along the course. Mark the points where DaNI stops. Measure the actual X and Y distances with a tape measure and compare the actual distance values with the values measured by the encoders on each wheel. What is the error? How does the error compare with the encoder resolution? Did DaNI travel straight? How does distance driven affect the error? How does velocity affect the error? Can you use the difference in encoder values between the wheels to determine the Y offset?

Does the error change if you reduce the wait time from 250 to 20 ms? Why?

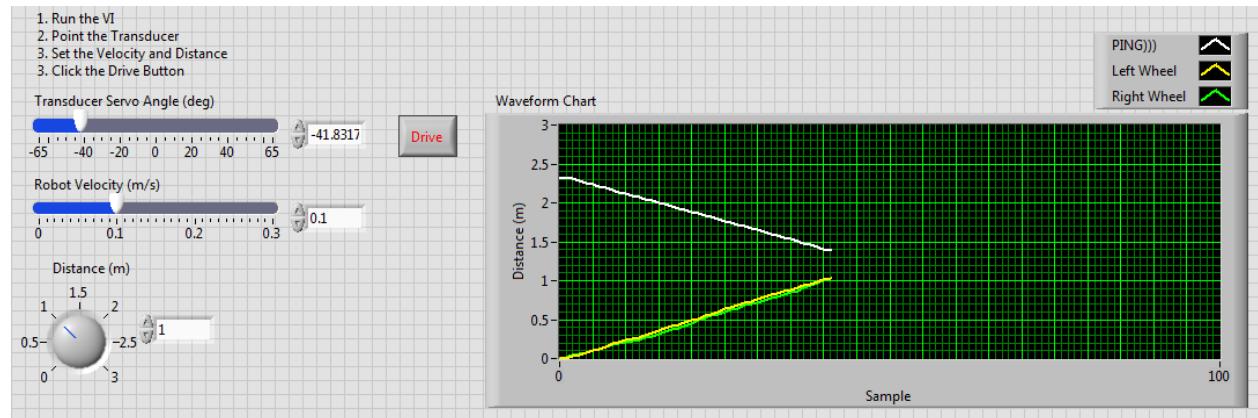
Use the 20 ms time and add a multiplier to correct the error in total distance driven. You will correct the error in orientation later. Repeat the experiment and report the change in the error.

If DaNI hit an obstacle and the floor was slick enough that the wheels spin, how would that affect the difference between encoder measured distance and actual distance?

Integrate the ultrasonic and encoder data by creating the VI shown in Figure 0-20. Arrange the front panel as shown in Figure 0-21. An additional loop was added to allow the user to calibrate the angle before driving. After calibration and setting the distance and velocity, the user clicks the Drive button and the main loop executes. Save the VI and the project. Run the VI tethered to the host. Figure 0-21 shows the results from driving toward a wall.



**Figure 0-20. Ultrasound vs Encoder VI block diagram**



**Figure 0-21. Ultrasound vs Encoder VI front panel**

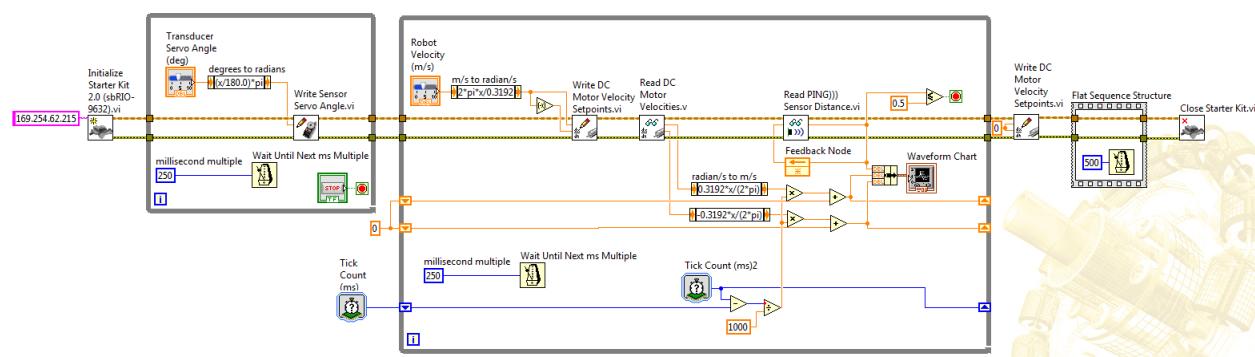
Repeat the previous experiment on the course with three distance targets adding a wall or other obstacle for ultrasonic measurement data. Compare the error in distance measurements between the encoders and PING))) at three different target distances and three different velocities. Which transducer measures distance better and faster at high speed? Explain.

Does the error change if you reduce the loop wait time from 250 ms to 20 ms? Why?

In what situations could you use PING))) data to measure distance traveled? Could you use PING))) data to differentiate between a situation where DaNI is stopped by a low obstacle and wheel spinning and normal driving?

Try driving with remote control a target distance without measuring the distance with a tape measure. Describe how transducers and computer control can assist humans during driving. Describe situations where humans need to supervise DaNI.

Modify the Ultrasound vs Encoder VI so that it will automatically stop DaNI when driving straight ahead and PING))) detects an obstacle that is 0.5 m away as shown in Figure 0-22. Save the VI and save the project. Test it with the motors off by placing an obstacle within 0.5 m to see if the program terminates. Then, test it by driving straight toward an obstacle that PING))) will detect.

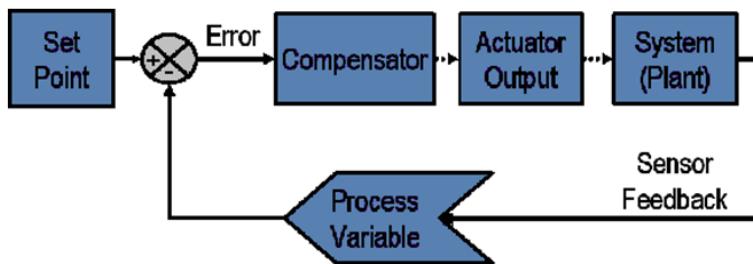


**Figure 0-22. Avoid Collision VI block diagram**

You used the Write DC Motor Velocity Setpoints VI in several previous programs to drive DaNI at a user requested velocity. The code compares the velocity measured with the encoder to the requested velocity, or setpoint. The code includes a control algorithm that automatically accelerates the motors to the setpoint and tries to maintain the setpoint. The code is included in the FPGA bitfile and is explained in the following.

The FPGA bitfile implements a Proportional-Integral-Derivative (PID) control algorithm. As the name suggests, PID algorithm consists of three parts: proportional, integral and derivative. Each part includes a coefficient which is varied to get optimal response. The PID algorithm reads the encoder and calculates the error between the target and the measured values. Then it computes the desired motor output to correct the error by calculating proportional, integral, and derivative responses and summing those three components. In a typical control system, the *process variable* is the system parameter that needs to be controlled, in this instance rotational velocity in radians/second. A transducer, in this instance the encoders, measures the process variable and provides feedback to the control system. The *set point* is the desired or command value for the process variable, such as 10 rad/s. At any given moment, the difference between the process variable and the set point is used by the control system algorithm (*compensator*), to determine the desired actuator output to drive the system (plant). For instance, if the measured velocity process variable is 9 rad/s and the desired set point is 10 rad/s, then the *actuator output* specified by the control algorithm might be to increase the motor velocity. This is called a closed loop control system, because the process of reading sensors to

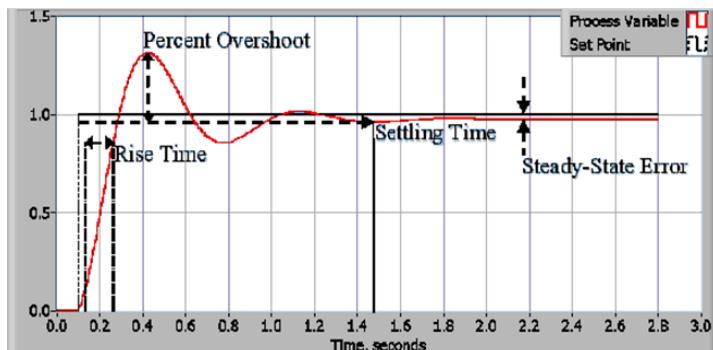
provide constant feedback and calculating the desired actuator output is repeated continuously and at a fixed loop rate as shown in Figure 0-23.



**Figure 0-23. Block diagram of a typical closed loop system.**

In many cases, the actuator output is not the only signal that has an effect on the system. For instance the robot might drive from one type of surface onto another which causes a change in the velocity. Such a change is referred to as *disturbance*. You design the control system to minimize the effect of disturbances on the process variable.

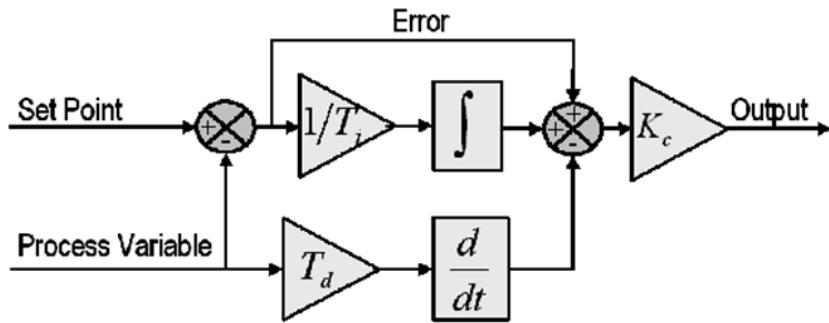
Control system performance can be measured by applying a step function as the set point command variable, and then measuring the response of the process variable. Commonly, the response is quantified by measuring defined waveform characteristics. Rise Time is the amount of time the system takes to go from 10% to 90% of the steady-state, or final, value. Percent Overshoot is the amount that the process variable overshoots the final value, expressed as a percentage of the final value. Settling time is the time required for the process variable to settle to within a certain percentage (commonly 5%) of the final value. Steady-State Error is the final difference between the process variable and set point. Deadtime is a delay between when a process variable changes, and when that change can be observed. The interval of time between calls to a control algorithm is the Loop Cycle Time. Systems that change quickly or have complex behavior require faster control loop rates.



**Figure 0-24. Response of a typical PID closed loop system.**

The PID algorithm addresses each of these measures. The proportional component depends

only on the difference between the set point and the process variable. This difference is referred to as the Error term as shown in **Error! Reference source not found..**



**Figure 0-25. Block diagram of a basic PID control algorithm.**

The *proportional gain* ( $K_c$ ) determines the ratio of output response to the error signal. For instance, if the error term has a magnitude of 10, a proportional gain of 5 would produce a proportional response of 50. In general, increasing the proportional gain will increase the speed of the control system response and decreases the rise time. However, if the proportional gain is too large, the process variable will begin to oscillate. If  $K_c$  is increased further, the oscillations will become larger and the system will become unstable and may even oscillate out of control.

The integral component sums the error term over time. The result is that even a small error term will cause the integral component to increase slowly. The integral response will continually increase over time unless the error is zero, so the effect is to drive the Steady-State error to zero. Steady-State error is the final difference between the process variable and set point.

The derivative component causes the output to decrease if the process variable is increasing rapidly. The derivative response is proportional to the rate of change of the process variable. Increasing the *derivative time* ( $T_d$ ) parameter will cause the control system to react more strongly to changes in the error term and will increase the speed of the overall control system response. Most practical control systems use very small derivative time ( $T_d$ ), because the Derivative Response is highly sensitive to noise in the process variable signal. If the sensor feedback signal is noisy or if the Control Loop Rate is too slow, the derivative response can make the control system unstable

The process of setting the optimal gains for P, I and D to get an ideal response from a control system is called *tuning*. There are different methods of tuning including the trial and error and the Ziegler Nichols methods.

In the trial and error method, the I and D terms are set to zero first and the proportional gain is increased until the output of the loop oscillates. As one increases the proportional gain, the system becomes faster, but care must be taken not make the system unstable. Once P has been set to obtain a desired fast response, the integral term is increased to stop the oscillations. The integral term reduces the steady state error, but increases overshoot. Some amount of

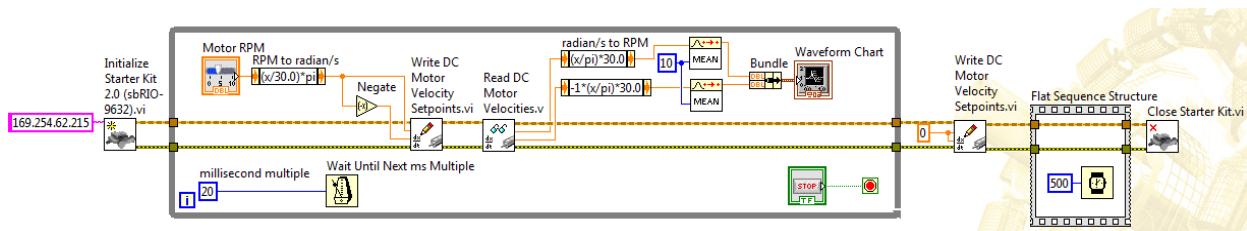
overshoot is always necessary for a fast system so that it could respond to changes immediately. The integral term is adjusted to achieve a minimal steady state error. Once the P and I have been set to get the desired fast control system with minimal steady state error, the derivative term is increased until the loop is acceptably quick to its set point. Increasing derivative term decreases overshoot and yields higher gain with stability but would cause the system to be highly sensitive to noise.

The Ziegler-Nichols method is similar to the trial and error method wherein I and D are set to zero and P is increased until the loop starts to oscillate. Once oscillation starts, the critical gain  $K_c$  and the period of oscillations  $P_c$  are noted. The P, I and D are then adjusted as per Table 0-1.

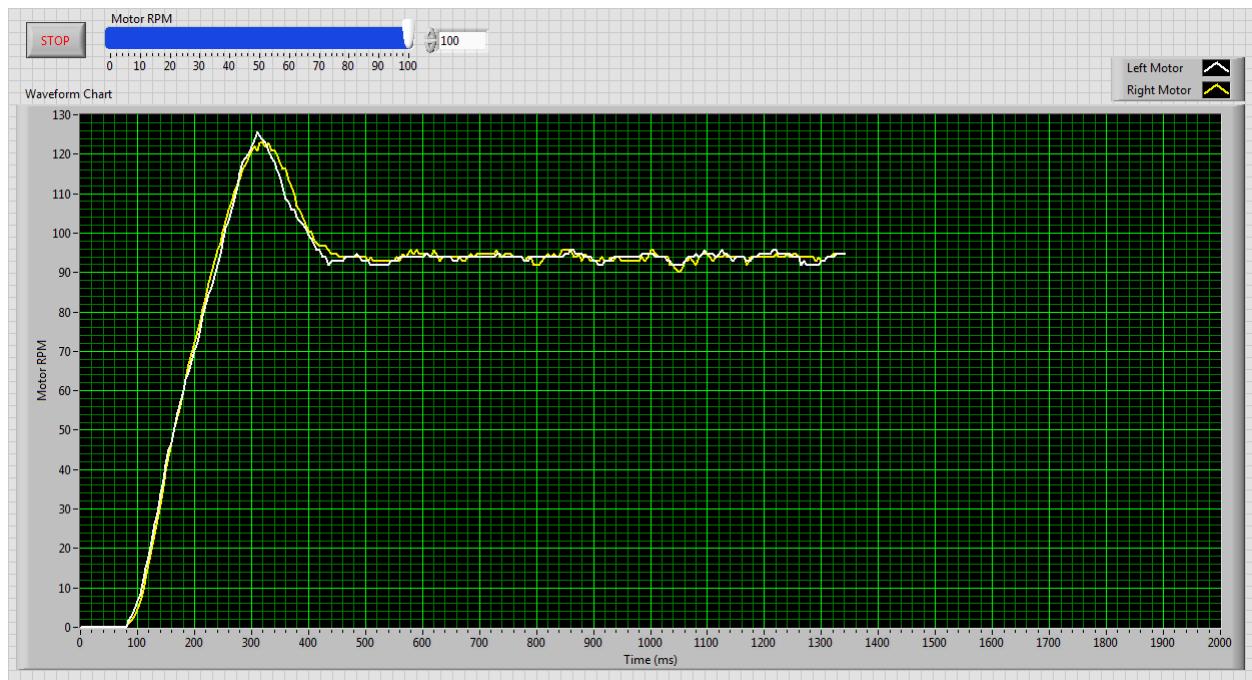
**Table 0-1. Ziegler-Nichols tuning PID relations**

Control	P	Ti	Td
P	$0.5K_c$	-	-
PI	$0.45K_c$	$P_c/1.2$	-
PID	$0.60K_c$	$0.5P_c$	$P_c/8$

Evaluate the rise time, percent overshoot, settling time and steady state error of the DaNI control algorithm. Set the DaNI frame on a box or small platform so that the wheels are not touching a surface and can turn freely. Smooth the data from the Drive Velocity VI with the Mean PtByPt VI (Functions>>Signal Processing>>Point by Point>>Prob & Stat palette) as shown in Figure 0-26. Save the VI and the project. Run the VI for a short time at a high velocity to obtain results similar to Figure 0-27.



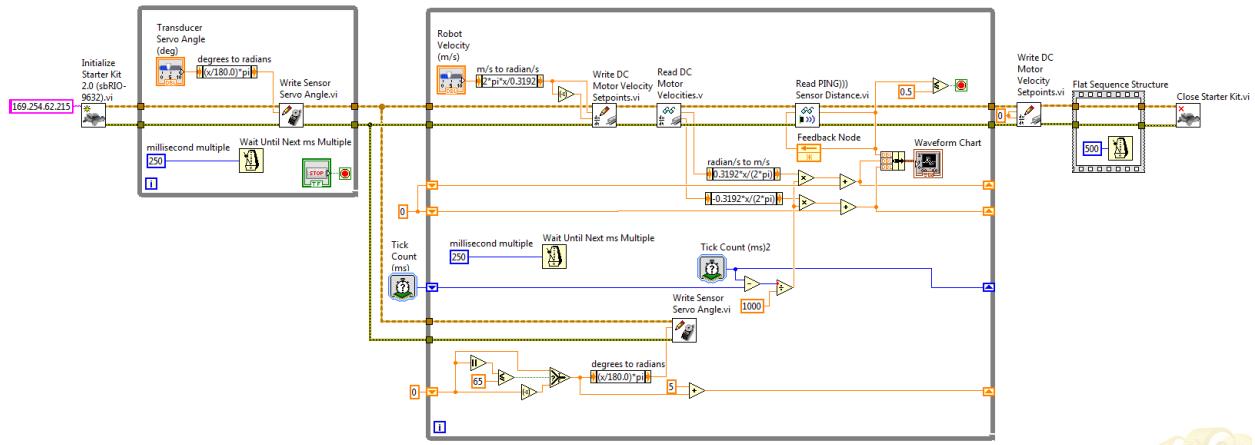
**Figure 0-26. Drive Velocity Smoothed Vi block diagram**



**Figure 0-27. Drive Velocity Smoothed front panel**

The previous discussion focused on the drive motors. It is also important to understand the control of the servo motor that pans the PING))) mount. To become more familiar with the servo motor, study the information on the Pitsco web site for the 180° servo, part number W39080. Note that the servo requires input from a controller with controllers capable of generating a pulse range from 600 usec to 2400 usec. Refer to Figure 2-1 and note that the motor driver isn't necessary since the servo input is a pulse signal from DIO7. The servo motor shaft can be positioned to specific angular positions by the length of the pulse signal. As long as the signal exists on the input line, the servo will maintain the angular position. As the coded signal changes, the angular position changes. The servo includes a potentiometer that is connected to the output shaft and control circuitry to monitor the current angle of the servo motor.

Create a VI that will pan the PING))) mount while driving straight ahead like the one shown in Figure 0-28. Note the use of shift registers to accumulate the servo angle value and the Comparison function to reverse the direction when the absolute value of the servo angle exceeds 65°. Save the VI and the project. Test it with the motors off. Then, drive toward an obstacle.



**Figure 0-28. Pan and Avoid Collision VI block diagram**

# Experiment 4 - Kinematics

## Instructor's Notes

This experiment requires that the previous experiments be completed. Similar experimental area and tools used in the previous experiments are used here.

## Goal

Discover the steering frame. Add motor control of turning and rotating to provide the capability to drive from point A to point B.

Learn about hierarchical programming and state machine architectures to build more sophisticated programs to sequence control tasks like rotate and drive to navigate from point A to point B.

## Required Components

Experimental area similar to that used in previous experiments.

Linear distance measuring tool like a ruler, meter stick, or tape measure.

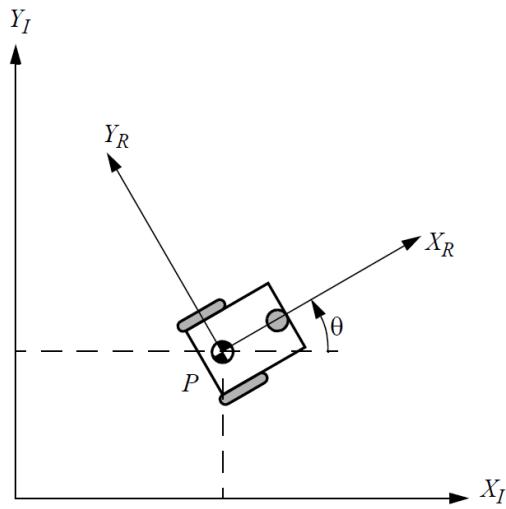
Angle measuring tool like a protractor.

## Background

Students should study Siegwart et al (2011) chapter 3

## Experiment 4-1 Turning and Rotating

Kinematics is the fundamental understanding of how a mechanism (DaNI in this instance) moves. It doesn't consider the forces involved, just the motion. Kinematics is critical to determining where a robot can go and how to get it there. In this instance, motion means driving from point A to point B given the position and orientation in a reference coordinate system such as one shown in Figure 0-1. Global coordinates describe the area where the robot operates, and local coordinates describe the orientation and position of the robot. The total dimensionality of the DaNI differential drive robot chassis on the plane is three, two for position in the plane and one for orientation along the vertical axis, which is orthogonal to the plane.



**Figure 0-1. Local and global coordinate frames (Siegwart et al (2011))**

Using the orientation of the local coordinate system in Figure 0-1, and to review the previous experiment, what is the equation that defines the distance moved in straight forward driving in terms of linear robot speed (velocity in the x direction, the robot linear forward velocity)? What is the equation that expresses the relationship between motor rotational velocity and wheel distance traveled?

$$L_w = \pi d/\text{revolution} * \dot{\theta}_w * t$$

Where  $L_w$  = wheel distance traveled,  
 $d$  = wheel diameter = 4 in for DaNI, and  
 $\dot{\phi}_w$  = wheel (= motor for DaNI) rotational velocity (radians/s)

If both wheels are driving forward at the same velocity with no slippage, what is the equation that expresses the relationship between the wheel distance traveled and the robot linear velocity  $x_r$ ?

$$x_r = L_w/t$$

Consider turning in addition to driving straight forward. What is the equation that defines the angle  $\theta$  rotated about the left wheel and the robot rotational velocity,  $\dot{\theta}_r$ , in terms of the linear

velocity of the right wheel,  $\dot{x}_{r1}$  (wheel 1)? (The right wheel drives while the left wheel does not drive or roll but remains at the same point.)

$$\theta = 360^\circ / 2\pi l * \text{wheel distance traveled}$$

Where  $l$  = distance between each wheel and  $P = 6.5$  in for DaNI

Circle perimeter distance =  $2\pi l$

$$\text{Wheel distance traveled} = \pi d / \text{rotation} * \dot{\phi}_w * t$$

What equation relates wheel rotational velocity,  $\dot{\phi}_w$ , (Rad/s) to robot rotation velocity,  $\omega$ , (°/s) when turning about one wheel?

$$\omega = (\dot{\phi}_w \text{ rad/s}) (1/2\pi \text{ rev/rad}) (4\pi \text{ in/rev}) (360/2 * 13\pi \text{ °/in}) = 8.8149 \dot{\phi}_w$$

Create a VI by modifying the Simple Drive Distance VI that will stop DaNI after it turns a user defined number of degrees about the left wheel as shown in Figure 0-2 and Figure 0-3.

Connect a constant of 0 to the left wheel Write Setpoint VI. Change the conversion as shown per the equation above. Add an absolute value for stopping for either a cw or a ccw turn. Modify the front panel as shown in Figure 0-3.

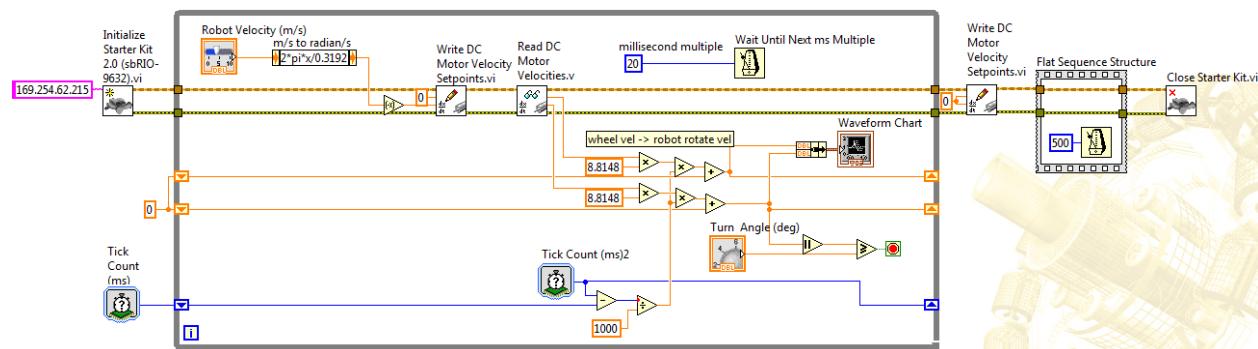
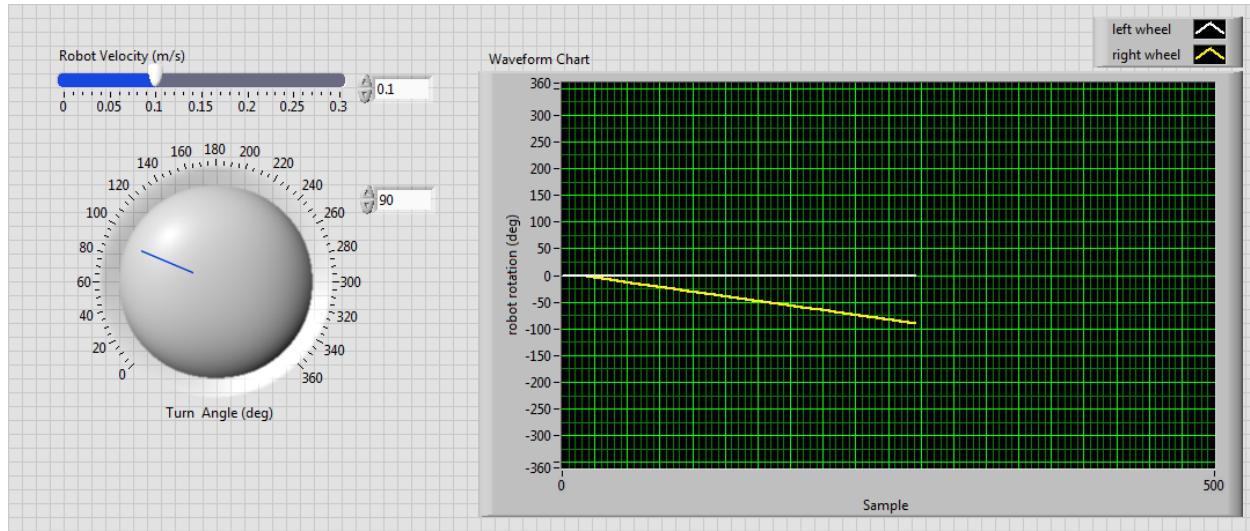


Figure 0-2. Simple Turn VI block diagram

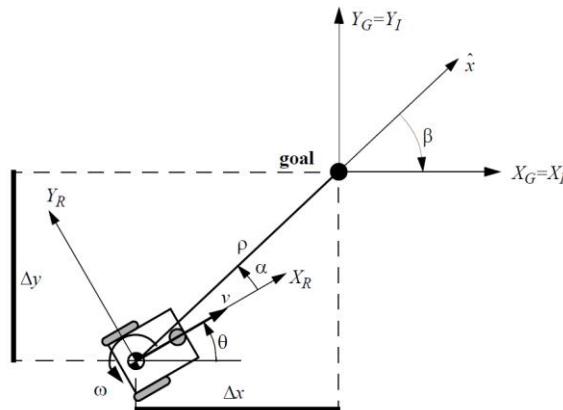


**Figure 0-3. Simple Turn VI front panel**

Measure the error in position after turns of  $90^\circ$ ,  $180^\circ$ , and  $360^\circ$ . Explain any error.

Turn  $90^\circ$  in the same direction four times and compare the error with turning  $360^\circ$ . Explain any difference between the two errors.

If you want to move from point A (the start pose) to point B (the goal pose) in Figure 0-4, you might first point the robot at point B by rotating in place. What must the two motors do to turn in place, so that only the orientation, and not the coordinates, of the robot change?

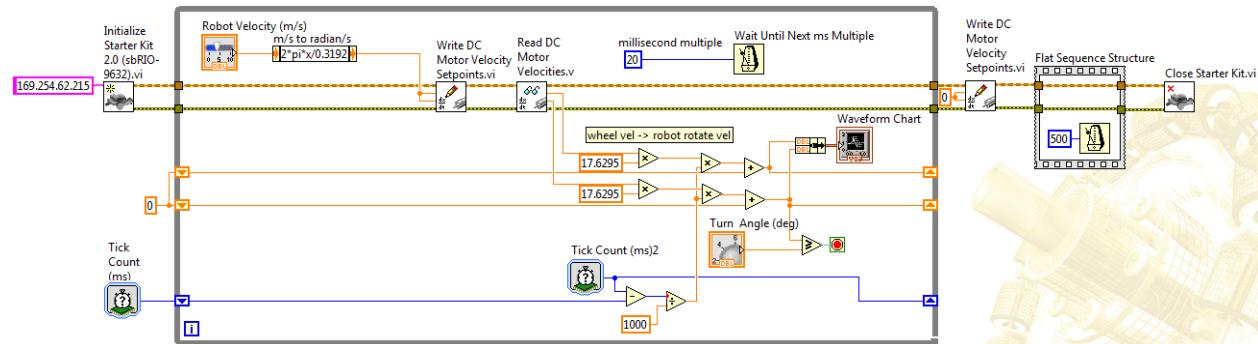


**Figure 0-4. Frames to move from start to goal (after Siegwart et al (2011))**

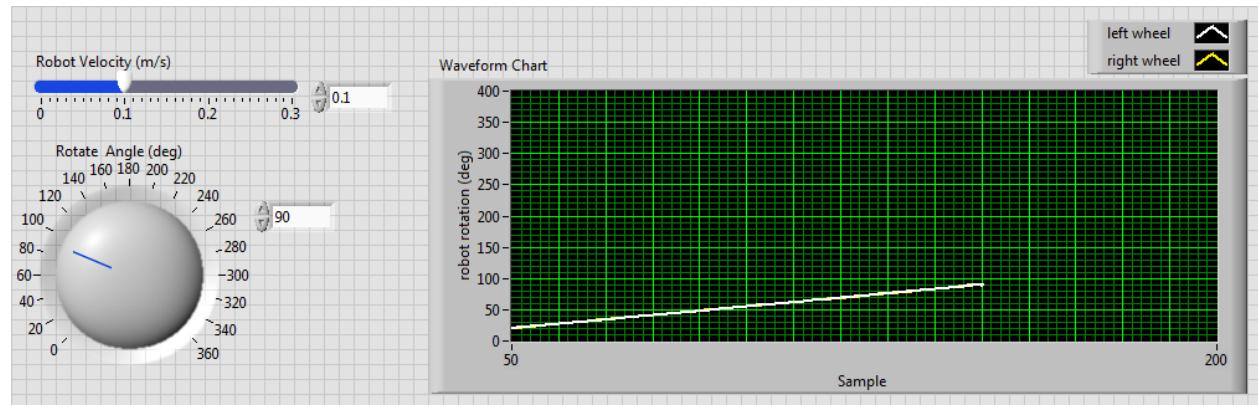
What equation relates wheel rotational velocity,  $\dot{\phi}_w$ , (Rad/s) to robot rotation velocity,  $\omega$ , ( $^\circ/s$ ) when turning the robot in place?

$$\omega = (\phi_w \text{ rad/s}) (1/2\pi \text{ rev/rad}) (4\pi \text{ in/rev}) (360/2*13\pi \text{ °/in}) = 17.6295 \phi_w$$

Modify the Simple Drive Distance VI to create a VI, like the one shown in Figure 0-5, that will rotate the robot in place a user selected number of degrees and stop. Delete the Negate function so the wheels will turn in opposite directions.



**Figure 0-5. Simple Rotate VI block diagram**



**Figure 0-6. Simple Rotate VI front panel**

Measure the error in position after rotations of  $90^\circ$ ,  $180^\circ$ , and  $360^\circ$ .

Compare the error in rotating with the error in turning and explain the difference.

Add error correction multipliers to the Rotate and Turn VIs. Do the corrections work equally well for all differential drive robots? For all DaNI robots? Why?

Repeat the  $90^\circ$ ,  $180^\circ$ , and  $360^\circ$  rotation tests at three different speeds and compare the results. Explain the difference in the error at different speeds.

## Experiment 4-2 User Choice: LabVIEW Case Structure and Enum Data Type

Use the information in the following to modify the Simple Turn VI to Create a VI where the user has the option of rotating or turning either cw or ccw. Modify the front panel by changing the label on the knob to Angle (deg). Add an enumerated control for the user to select one of four actions: Turn cw, Turn ccw, Rotate cw, Rotate ccw.

The enumerated data type (enum) passes integer numeric values but is very user friendly because it displays text (string data type). To create the control, open the Controls>>Modern>>Ring & Enum palette and drag the Enum control onto the front panel. Then right click the enum control and choose Properties or Edit Items. The window shown in Figure 0-7 opens. Choose the Edit Items tab and type the names of the four text items in the order shown. If you type them in the wrong order, it is easy to modify the list with the Move Up and Move Down buttons.

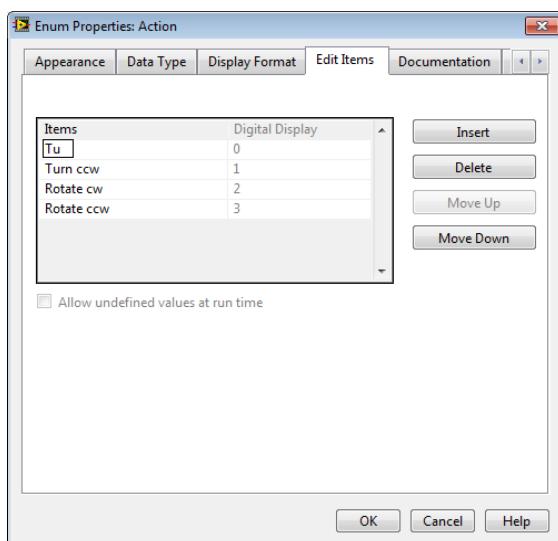
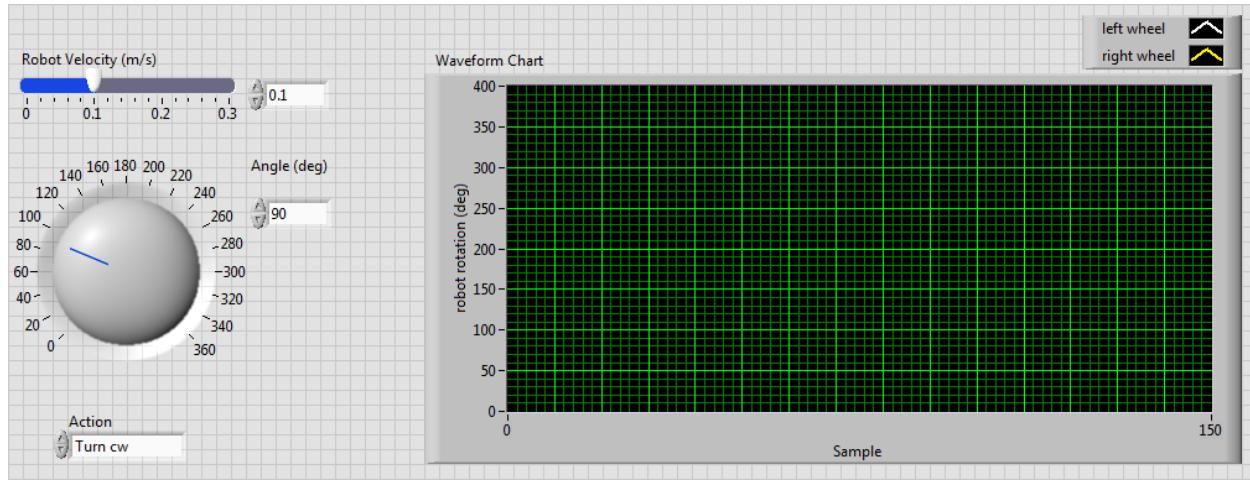


Figure 0-7. Enum control properties window Edit Items tab

The enum will appear on the front panel as shown in Figure 0-8.

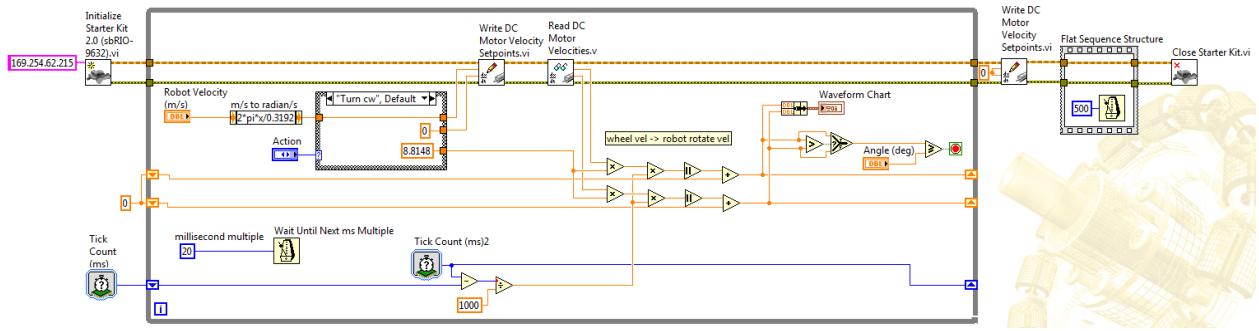


**Figure 0-8. Simple Rotate or Turn VI front panel**

In this instance, Turn cw has the numeric value 0, Turn ccw is one, Rotate cw is two, and Rotate ccw is three. The user doesn't see or need to remember the numeric values. The user just selects the string from the list that represents the action of interest. The numeric representation of the enumerated type control is an 8-, 16-, or 32-bit unsigned integer. Right-click the control and select Representation from the shortcut menu to change the representation of the control. You cannot enter undefined values in enumerated type controls, and you cannot assign specific numeric values to items in enumerated type controls. You are limited to the four items listed above unless you edit the list.

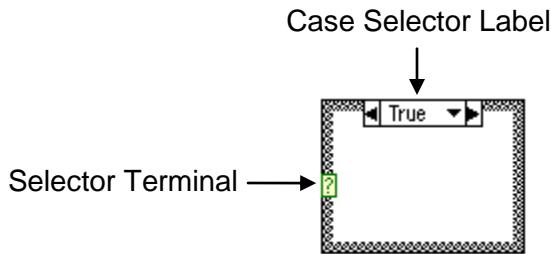
When you wire an enumerated type control to the selector terminal of a Case structure (explained in the following), LabVIEW matches the cases to the string values of items in the control, not the numeric values. When you wire an enumerated type control to the selector terminal of a Case structure, you can right-click the structure and select Add Case for Every Value to create a case for the string value of each item in the control.

Modify the block diagram of the Simple Rotate VI as shown in Figure 0-9 using the information in the following. The major change is adding a Case structure from the Functions>>Structures palette. The Case structure supports alternatives or branches in the flow of code. It is similar to switch statements or if...then...else statements in text-based programming languages. You place it on the block diagram using the same procedure as the While loop by clicking to pin the upper left corner and moving the cursor to the lower right corner and clicking again. You can make space on the block diagram before adding the structure by holding down the Ctrl key, clicking and dragging the cursor in the direction you need space, downward in this instance.



**Figure 0-9. Simple Rotate and Turn VI block diagram**

When you add the Case structure, it has the configuration shown in Figure 0-10. The structure contains one or more subdiagrams, or cases, exactly one of which executes when the structure executes. The value wired to the selector terminal determines which case to execute and can be Boolean, string, integer, enumerated type, or error cluster. The configuration shown in Figure 0-10 is Boolean which is the default configuration when the structure is first placed on the block diagram. It has two cases, True and False. If the input to the Selector Terminal is True, the code in the True case executes. If False, the code in the False case executes.



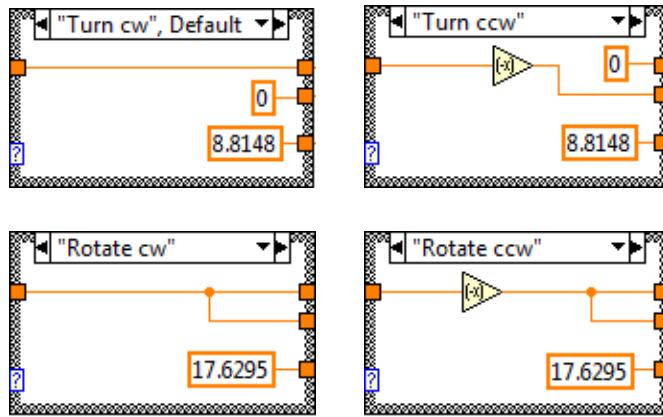
**Figure 0-10. Default Case structure**

You need four cases to match the enum control created above. You could right-click the structure border to add or delete cases, using the Labeling tool to enter value(s) in the case selector label and configure the value(s) handled by each case. If you enter a selector value that is not the same type as the object wired to the selector terminal, the value appears red. This indicates that the VI will not run until you delete or edit the value. To avoid errors and to save time, the best way to configure for enum data type is to wire the enum control icon to the Selector Terminal. The structure will automatically morph to the enum data type configuration shown in Figure 0-9. However, only two cases will be present as that is all there were in the default configuration. To add the other cases, right click the border of the structure and choose Add Case For Every Value from the short cut menu.

Because of the possible round-off error inherent in floating-point arithmetic, you cannot use floating-point numbers as case selector values. If you wire a floating-point value to the case, LabVIEW rounds the value to the nearest integer. If you type a floating-point value in the case

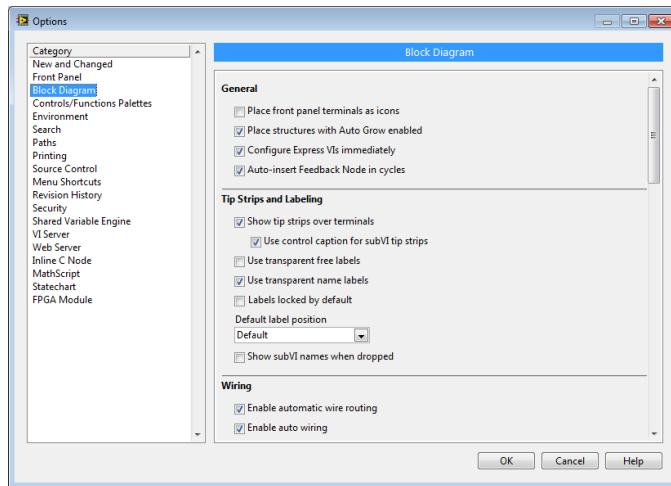
selector label, the value appears red to indicate that you must delete or edit the value before the structure can execute.

Create code in the four cases as shown in Figure 0-11. You can wire into and out of the Case structure through tunnels, the same as with a loop. The input tunnels are on the left border. In this instance, there is only one input tunnel: the Robot Velocity converted to wheel (motor) ccw velocity in rads/s as shown in Figure 0-9. This value is the same in all cases. The output tunnels are on the right border. There are three outputs in this instance: the left ccw motor velocity setpoint, the right ccw motor velocity setpoint, and the conversion factor for motor velocity in rads/s to robot rotate velocity in °/s. The left ccw motor velocity setpoint is equal to the input in the Turn and Rotate cw cases. It is 0 in the Turn ccw case as the other wheel drives and turns about the left wheel. It is negative in the Rotate ccw case as the wheel turns cw in this instance. The right ccw motor velocity setpoint is 0 in the Turn ccw case as the other wheel drives and turns about the right wheel. It is negative in the Turn and Rotate ccw cases. It is equal to the input in the Rotate cw case. The conversion constants are different in the Turn and Rotate cases since the turn radius is twice the rotate radius. You can incorporate error correction multipliers in the cases as well if appropriate. When you wire to the border in the first case, a tunnel appears, but it is not filled with color as shown. After you wire all four cases to this tunnel, the tunnel will fill with color. A common error with Case structures is neglecting to complete the wiring in all the cases. When you wire to an object outside the structure, an error results if you do not connect a source in all the cases. You can configure the output tunnels to use the default value instead of wiring to them; but it is not recommended as it makes programs difficult to debug and reduces readability.



**Figure 0-11. Code in the four cases**

Notice that the size of the control and indicator icons on the block diagram are smaller in Figure 0-9 than in previous block diagram figures. The sizes were reduced to create space for additional code and still allow the diagram to fit on the monitor. You can make this change individually by right clicking on each icon and unchecking View As Icon. Or, you can change it in by configuring the programming environment by choosing Tools from the menu bar and choosing Options in the drop down menu. The dialog box shown in Figure 0-12 opens. Uncheck Place front panel terminals as icons to use the smaller terminal shape instead of the large icon for future controls and indicators. You will have to make the change individually for existing objects.



**Figure 0-12. Tools>>Options configuration dialog box**

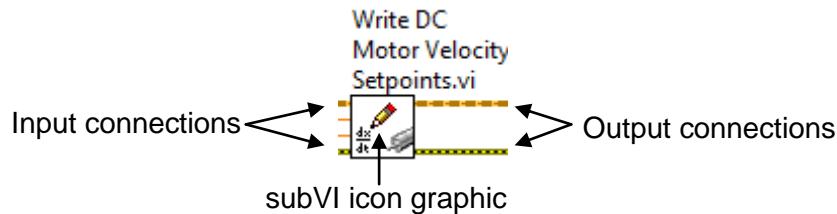
Rearrange the wires and icons on the block diagram as shown in Figure 0-9, adding the Absolute Value function to stop the VI automatically in either cw or ccw directions.

## Experiment 4-3 Using Hierarchical Programming to Drive from Start to Goal

You can now integrate the concepts with the previous experiment and drive from point A to point B considering the reference frames in Figure 0-4. You can do this by executing two separate programs, the Simple Drive Distance VI and the Simple Turn and Rotate VI. If DaNI needs to turn  $90^\circ$  ccw ( $\alpha$  in Figure 0-4) and drive forward 1m (pin Figure 0-4) to reach the goal point, execute the rotation VI and then the drive VI. Executing two VIs is a little cumbersome, so combine them into one program by creating two sub VIs.

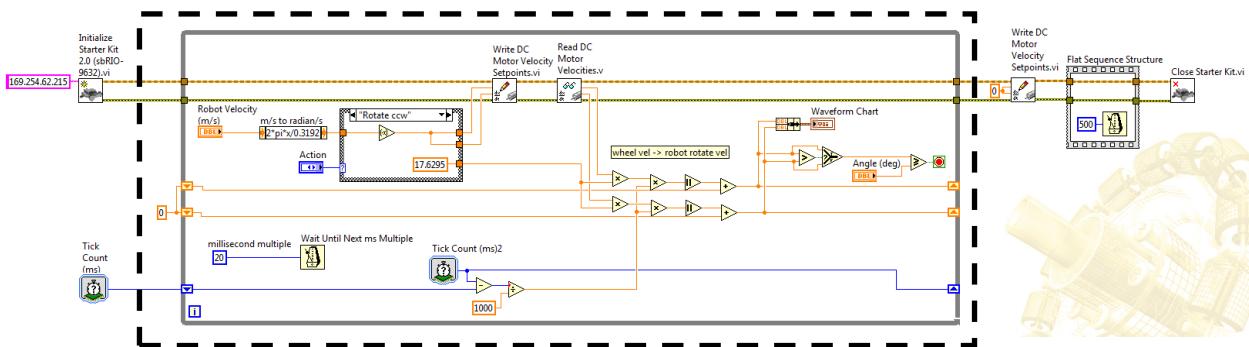
After you build a VI, you can use it in another VI. A VI called from the block diagram of another VI is called a subVI. You can reuse a subVI in other VIs. To create a subVI, you need to build a connector pane and create an icon.

Figure 0-13 shows an example from the Turn and Rotate VI that calls the Write DC Motor Velocity Setpoints VI. The icon is a graphic that shows a pencil for writing and a motor. There are four connections for input wires on the left of the icon and two connections for outputs on the right side of the icon. The input and output connections correspond to controls and indicators respectively on the subVI front panel. The subVI controls and indicators receive data from and return data to the block diagram of the calling VI. A subVI node corresponds to a subroutine call in text-based programming languages. A block diagram that contains several identical subVI nodes calls the same subVI several times.



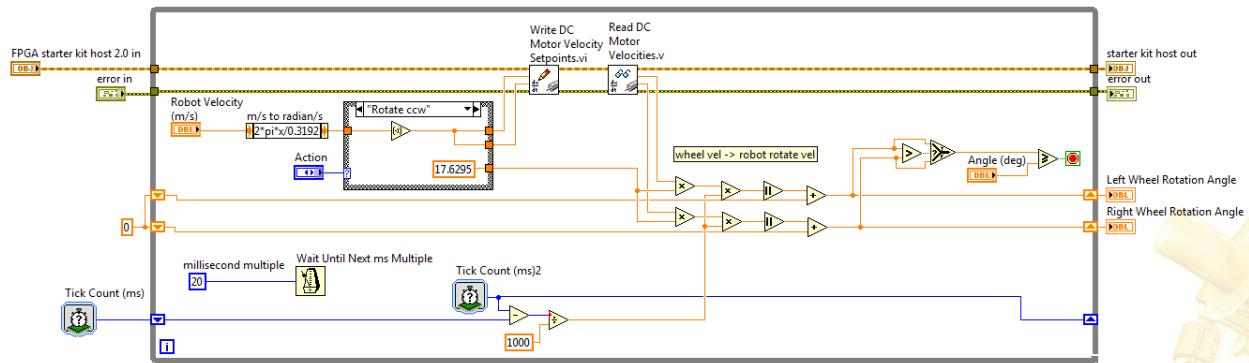
**Figure 0-13. Example SubVI Icon and connections**

Not all of the code in the two VIs needs to be placed in the subVIs. Place the code in the dashed rectangle in Figure 0-14 in the subVI and place the code outside in the main VI. As explained in the previous paragraph, you will add controls and indicators to the subVI front panel to pass data into and out of the main VI for these objects.



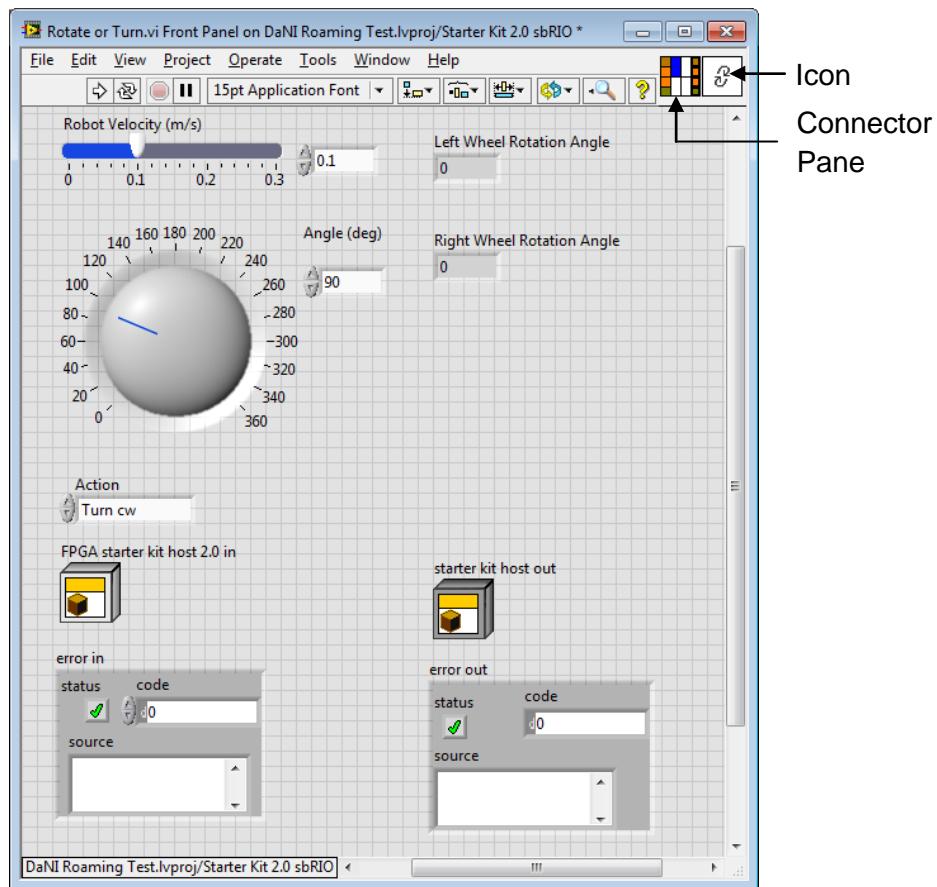
**Figure 0-14. Divide code between main and subVI**

Delete the objects that will not be in the subVI, or cut and paste them to a main VI and save and close it. You will create the main VI later. Delete the chart on the subVI front panel as you will not display the front panel of the subVI while it executes. Add indicators to the Shift Registers on the right border of the loop, as shown, to display the final angle rotated. You could create an array of the angle values from each loop iteration and pass the array to the calling VI for graphing if you want, but arrays aren't covered until later in this set of experiments. Right click the tunnels where the objects were deleted and create controls or indicators as shown in Figure 0-15. Change the error control and indicator and the starter kit labels as shown.



**Figure 0-15. Controls and indicators for subVI to main VI data transfer**

Arrange the front panel objects similar to Figure 0-16.



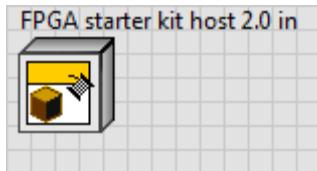
**Figure 0-16. Rotate SubVI front panel**

You need to build the connector pane identified in Figure 0-16. The connector pane is a set of terminals that corresponds to the controls and indicators of that VI, similar to the parameter list of a function call in text-based programming languages. The connector pane defines the inputs and outputs you can wire to the VI so you can use it as a subVI. A connector pane receives data at its input terminals, passes the data to the block diagram code through the front panel controls, and receives the results at its output terminals from the front panel indicators. Each rectangle on the connector pane represents a terminal. Use the rectangles to assign front panel controls and indicators as inputs and outputs. The default connector pane pattern is 4 x 2 x 2 x 4. You can select a different pattern by right-clicking the connector pane and selecting Patterns from the shortcut menu.

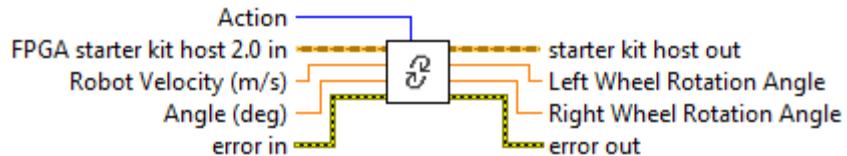
To link the controls and indicators on the connector pane, click on a rectangle, it will turn black as shown in Figure 0-17, and then click on an object as shown in Figure 0-18. The connector pane rectangle will change from black to the color that represents the data type of the front panel object. Repeat this for the remaining front panel objects, placing the controls on the left and indicators on the right, to make the assignments shown in Figure 0-19. It is customary to reserve the upper left and right rectangles for reference wires and the lower left and right ones for error wires.



**Figure 0-17. Click on a rectangle in the connector pane to assign it to a control or indicator**



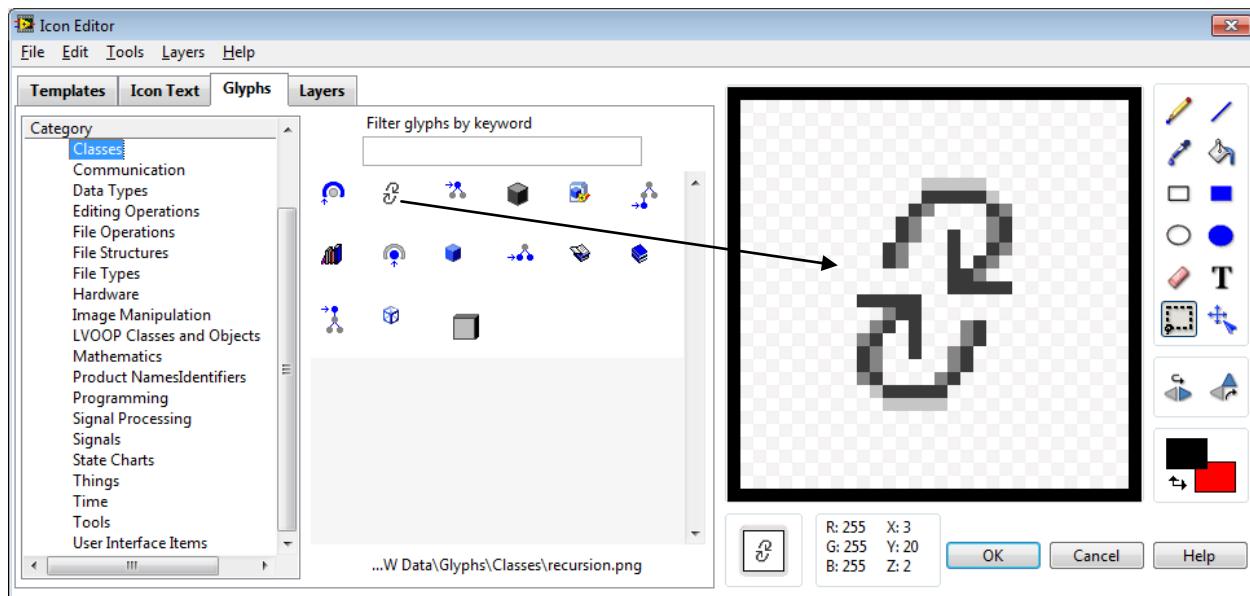
**Figure 0-18. Click on a front panel object after clicking on the connector pane rectangle to link the two**



**Figure 0-19. Rotate SubVI connector pane assignments**

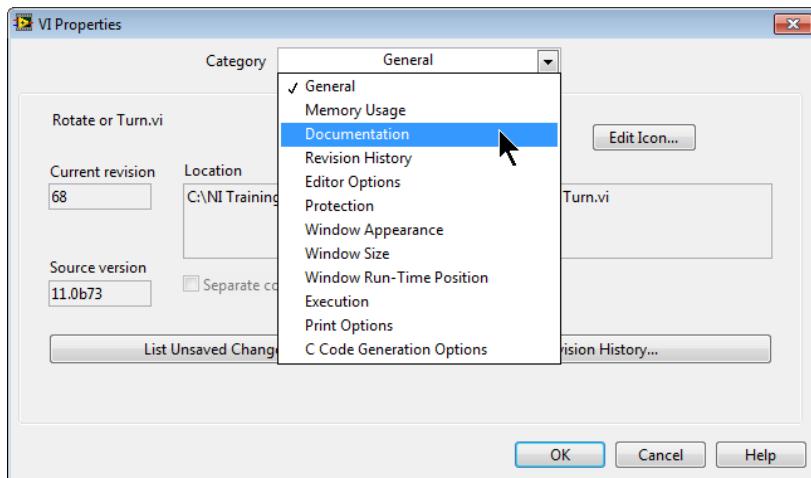
You can designate which inputs and outputs are required, recommended, and optional to prevent users from forgetting to wire subVI terminals. For terminal inputs, required means that the block diagram on which you placed the subVI will be broken if you do not wire the required inputs. Required is not available for terminal outputs. For terminal inputs and outputs, recommended or optional mean that the block diagram on which you placed the subVI can execute if you do not wire the recommended or optional terminals. If you do not wire the terminals, the VI does not generate any warnings. Right-click a terminal on the connector pane and select This Connection Is from the shortcut menu. A checkmark indicates the terminal setting. Select Required, Recommended, or Optional. You also can select Tools»Options»Front Panel and place a checkmark in the Connector pane terminals default to required checkbox. This option sets terminals on the connector pane to required instead of recommended. This applies to connections made using the wiring tool and to subVIs created using Create SubVI.

LabVIEW gives the icon a default graphic, but it is better to create your own that better represents the functionality of the subVI. Right click the icon on the front panel and choose Edit Icon. The icon editor shown in Figure 0-20 appears. Use the tools on the right pane that resemble the PC Paint drawing program to select and delete the LabVIEW default graphic. Leave the border. Select a graphic from the Glyphs library as shown and position it within the border. If there isn't an appropriate glyph, you can use the tools to draw a graphic. You can add text as well.



**Figure 0-20. Icon Editor**

LabVIEW automatically creates context help information for the subVI, so after you place it on a block diagram, you can right click on it, choose help (or move the mouse over the icon and hit Ctrl H), and the help window showing information similar to that in Figure 0-19 appears. In the Context Help window, the labels of required terminals appear bold, recommended terminals appear as plain text, and optional terminals appear dimmed. The labels of optional terminals do not appear if you click the Hide Optional Terminals and Full Path button in the Context Help window. You can add additional information to the help by adding documentation to the VI. Access the documentation dialog by choosing File>>VI Properties and selecting documentation from the Category drop down menu as shown in Figure 0-21. You can do this for any VI.



**Figure 0-21. Documentation**

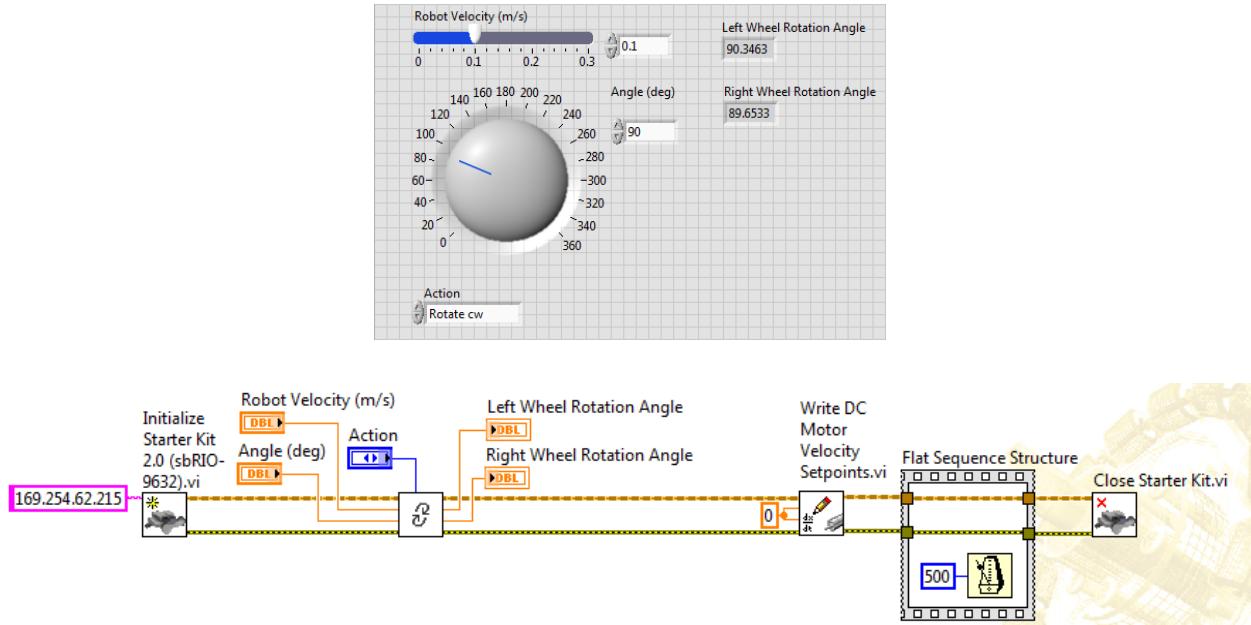
There are several different ways to place the subVI on the block diagram of a main or calling VI. You can click the Select a VI icon or text on the Functions palette, navigate to a VI, double-click the VI or drag the VI to the block diagram to create a subVI call to that VI. If necessary, display the Functions palette by selecting View»Functions Palette. Wire the subVI terminals to other nodes on the block diagram. You also can place an open VI on the block diagram of another open VI. Use the Positioning tool to click the icon in the upper right corner of the front panel or block diagram of the VI you want to use as a subVI, and drag it to the block diagram of the other VI. You can also drag a VI or control from the file system (like the Windows Explorer) to a block diagram. If you are using a LabVIEW project, you also can place a VI from the Project Explorer window on the block diagram of another open VI. Select the VI you want to use as a subVI from the Project Explorer window, and drag it to the block diagram of the other VI.

You can edit a subVI by using the Operating or Positioning tool to double-click the subVI on the block diagram. When you save the subVI, the changes affect all calls to the subVI, not just the current instance.

When LabVIEW calls a subVI, ordinarily the subVI runs without displaying its front panel. If you want a single instance of the subVI to display its front panel when called, right-click the subVI icon and select SubVI Node Setup from the shortcut menu. If you want every instance of the subVI to display its front panel when called, select File»VI Properties, select Window Appearance from the Category pull-down menu, and click the Customize button.

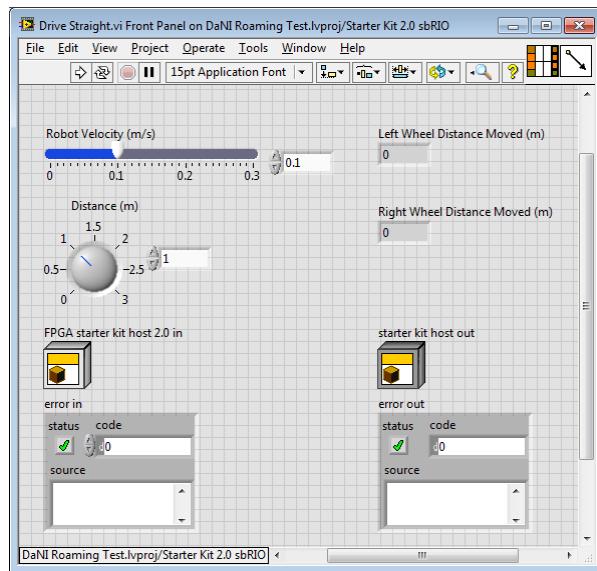
Once you create a subVI you can use it in many applications and in several locations in a calling VI. But first, you should test it thoroughly. It might not function on its own, so you might have to develop a separate VI, sometimes called a wrapper VI, just for testing. You can test the subVI in this instance by developing the main (calling) VI in two phases. The first phase is testing the rotate functionality with the wrapper VI shown in Figure 0-22. Create this VI by copying code from previous VIs, inserting the subVI icon and connecting the reference and error wires. Then

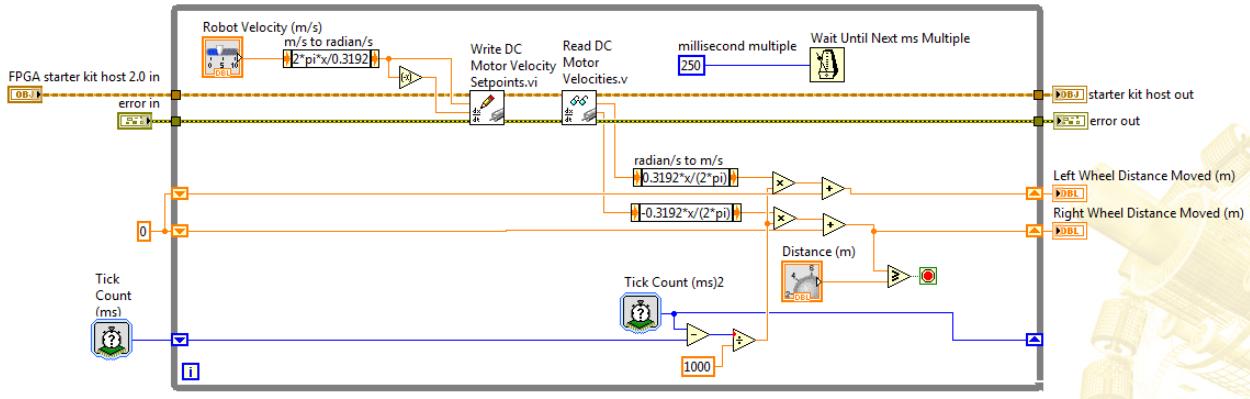
right click the other subVI icon terminals and create controls or indicators as appropriate. Test the subVI thoroughly with several different angles for each action.



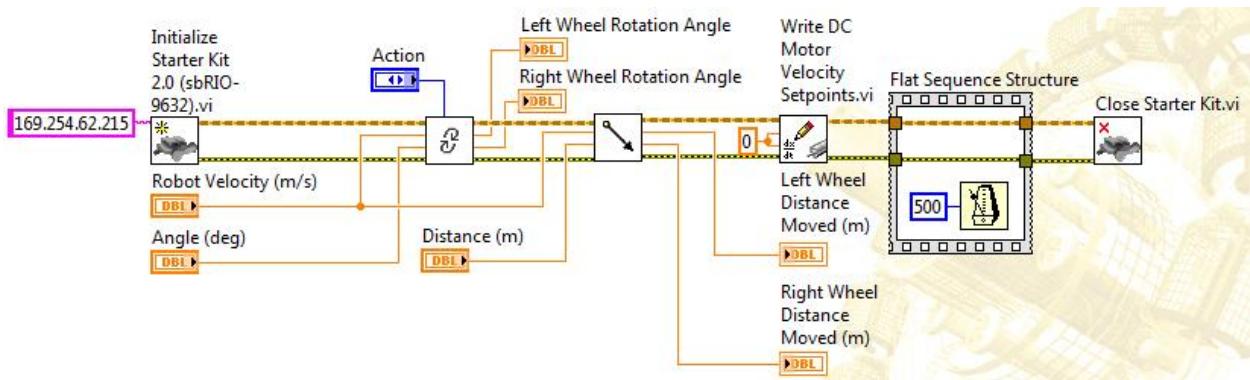
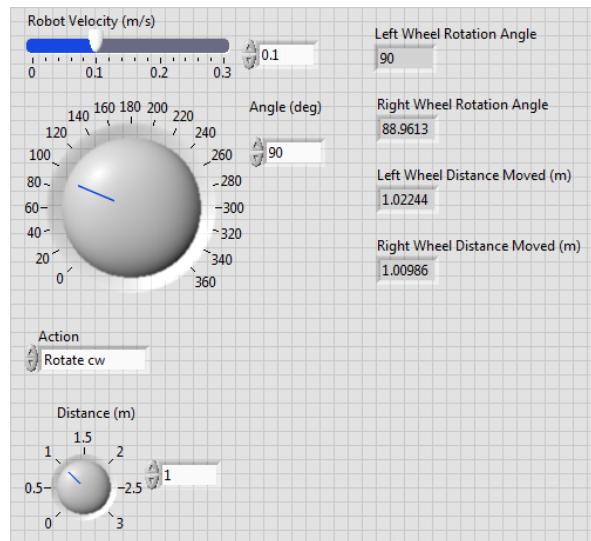
**Figure 0-22. First phase Rotate and Drive VI subVI tester**

Use the techniques from the previous sub VI to create another sub VI for driving straight from the Simple Drive Distance VI as shown in Figure 0-23. Test it separately from the Rotate VI in the wrapper. After testing, combine the VIs as shown in Figure 0-24 to create a VI that will rotate and drive to move DaNI from point A to point B.





**Figure 0-23. Drive Straight subVI**

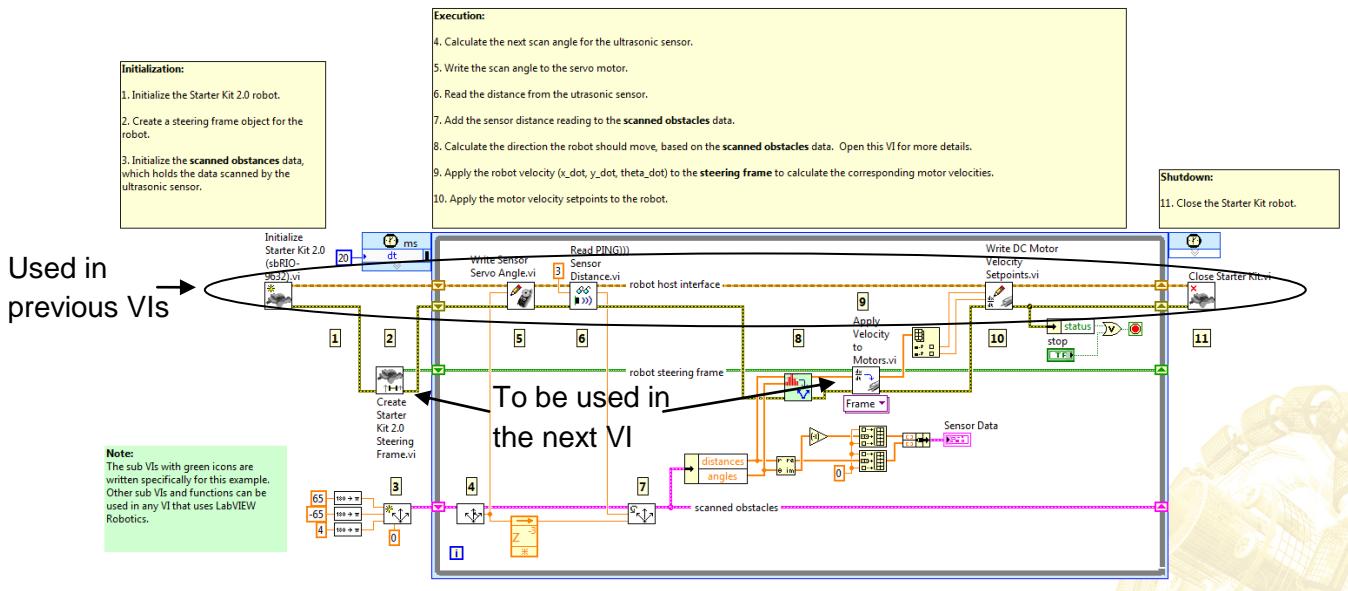


**Figure 0-24. Rotate and Drive VI**

This VI accepts angle and distance inputs. You could modify it to accept poses including orientation and coordinates at A and B and develop a subVI to calculate the angle and distance.

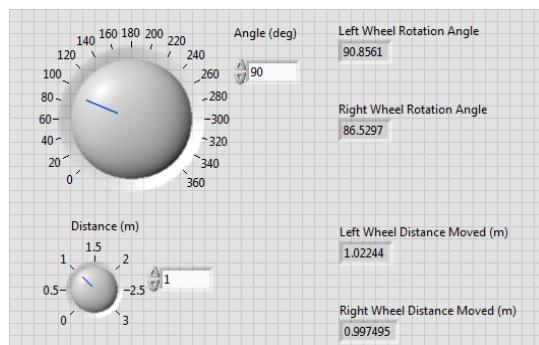
# Experiment 4-4 Steering Frame

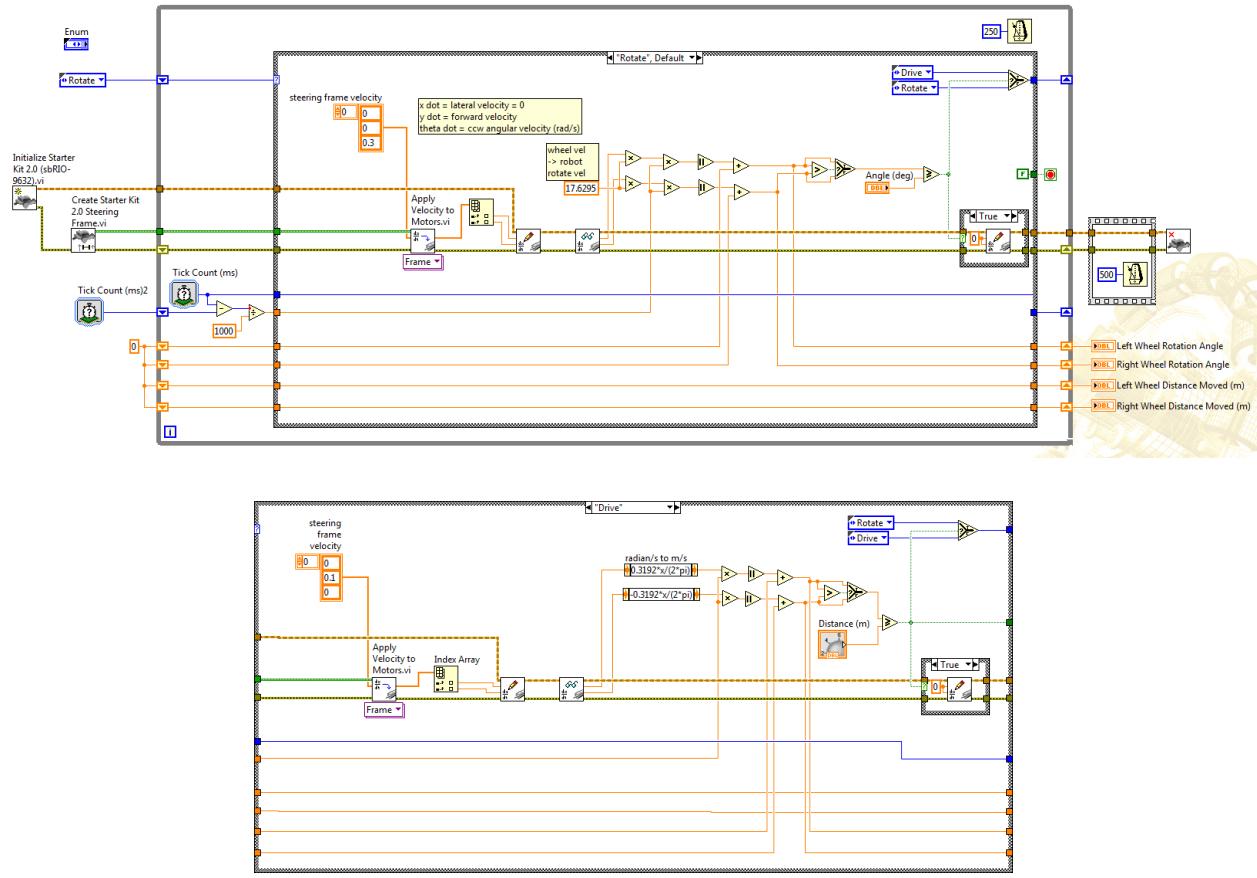
Review the block diagram of the Roaming VI used in experiment 1 shown in Figure 0-25. This section of the experiment will explain how it moves from point A to point B in contrast to the code you developed in the previous section. You discovered how to use all of the VIs circled in the upper path in previous experiments. This section will explain two of the VIs below this path: Create Starter Kit 2.0 Steering Frame VI and Apply Velocity to Motors VI.



**Figure 0-25. Roaming VI block diagram showing VIs used previously and VIs to be studied in this experiment**

Build the VI shown in Figure 0-26 and study the following information to get some experience with the Steering Frame and Apply Velocity VIs. In addition, you will learn about the LabVIEW State Machine Architecture. The important differences with previous programs are explained in the following.

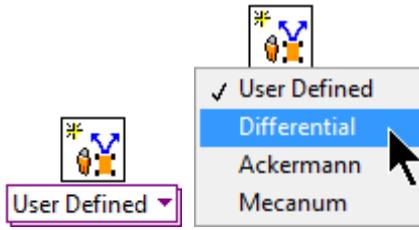




**Figure 0-26. Steering Frame Drive State Machine**

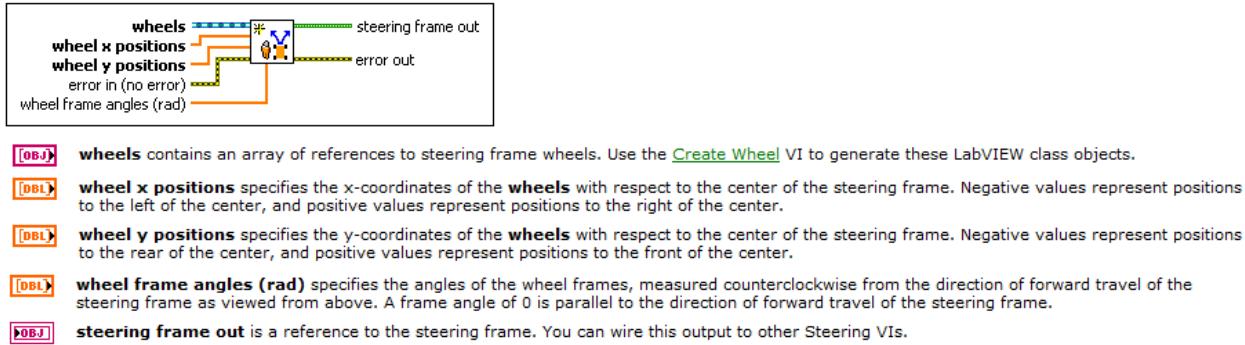
Begin with the Create Steering Frame VI. It is available from the Functions>>Robotics>>Starter Kit>>2.0 palette. It is a specific instance of the general Create Steering Frame VI that can be applied to a variety of mobile robots. The Create Steering Frame, defines the wheel type and the wheel geometry relative to the robot frame.

The general instance, shown in Figure 0-27 is available in Functions>>Robotics>>Steering palette. It has a drop down menu allowing you to select different instances of the VI as shown. VIs with this capability are called polymorphic. The drop down allows you to create a steering frame for several of the steering types described in Siegwart et al (2011). The type appropriate for DaNI 2.0 is differential.



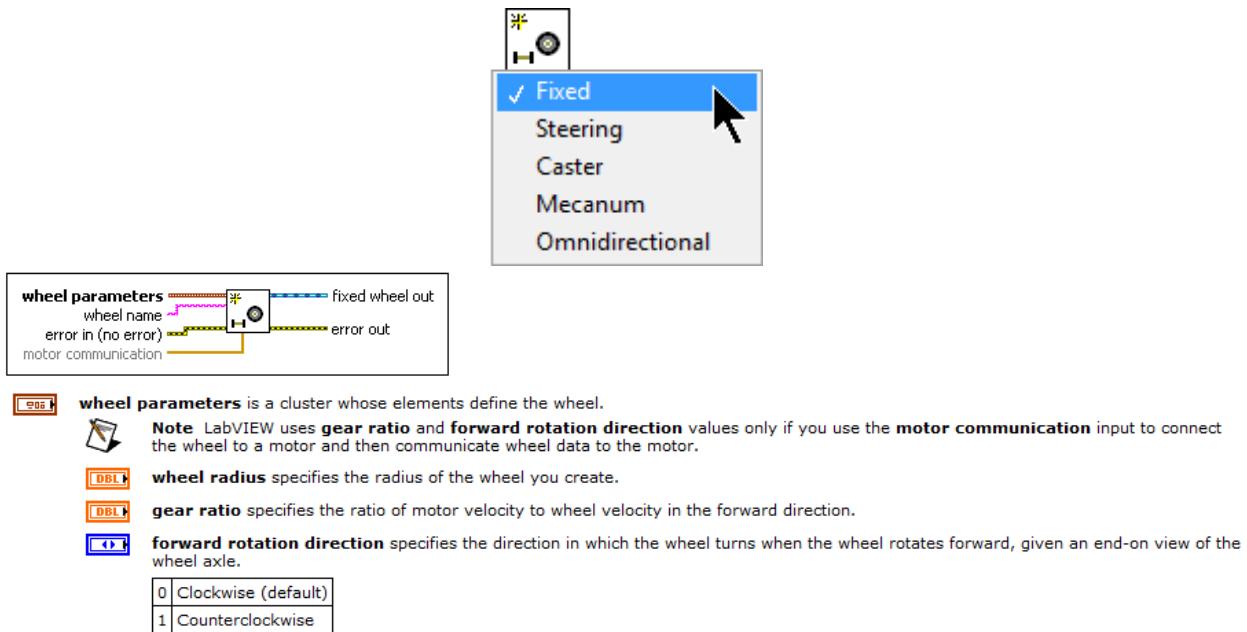
**Figure 0-27. General Steering Frame Polymorphic VI**

You can define your own instance of steering frame by selecting the default instance whose help information is shown in Figure 0-28. The User Defined instance gives insight into the type of information that the steering frame writes to computer memory. For example, you can specify the types of wheels on the robot with the Create Wheel VI described below. You can specify the x and y positions of all of the wheels relative to the center of the steering frame. You can specify the angles of the wheels relative to the direction of travel. When this VI executes, it places all of the information in computer memory for use by other VIs in a program and creates a reference number that can be wired to down stream VIs so they can access the memory.



**Figure 0-28. User Defined Steering Frame Help Information**

The Create Steering Frame VI accepts input from the Create Wheel VI where you can choose a polymorphic instance as shown in Figure 0-29. If you choose the Fixed Wheel instance, as is appropriate for DaNI 2.0, you can enter the wheel parameters including radius, gear ration, and forward rotation direction as shown.

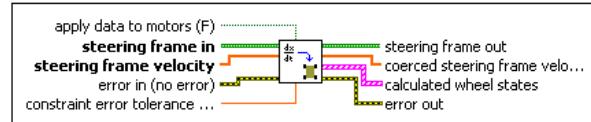


**Figure 0-29. Create Wheel Polymorphic VI showing help for the wheel parameters**

The specific instance of the create steering frame VI in Functions>>Robitics>>Starter Kit>>2.0 palette that is shown on the block diagram of Figure 0-26 does all of the configuration for you using information specific to DaNI 2.0 so the block diagram icon connections are limited to error wires and the reference out wire.

The Apply Velocity to Motors VI converts the robot center velocity to wheel angular velocities. You can choose between two polymorphic instances: Frame and Arc. Frame is the appropriate instance for DaNI 2.0. The help information for this VI is shown in Figure 0-30.

### Apply Steering Frame Velocity to Wheels



- TF** **apply data to motors** determines whether this VI applies the **calculated wheel states** this VI calculates for each wheel to any corresponding motors. The default is FALSE.  
If you used the [Motor Communication](#) VIs to connect wheels on the steering frame with motors, you must specify TRUE to update the communication sessions with the new motor velocity and angle setpoints this VI calculates.
- OBJ** **steering frame in** is a reference to the steering frame on which to operate. Use the [Create Steering Frame](#) VI to generate this LabVIEW class object.
- DBL** **steering frame velocity** specifies the velocity of the steering frame in the form of  $[x\_dot, y\_dot, theta\_dot]$  where  $x\_dot$  is the velocity in the lateral, or right, direction,  $y\_dot$  is the velocity in the forward direction, and  $theta\_dot$  is the angular velocity in the counterclockwise direction.
- DBL** **constraint error tolerance** is the maximum allowable difference between the value this VI calculates for the **steering angle** of a wheel and possible values for the steering angle given kinematic constraints of the steering frame. If the difference is greater than **constraint error tolerance**, LabVIEW returns a warning with the name of the last wheel whose **steering angle** is out of range. The default is 1E-12.
- Err** **error in** describes error conditions that occur before this node runs. This input provides [standard error in](#) functionality.
- Steering frame out** is a reference to the steering frame. You can wire this output to other Steering VIs.
- DBL** **coerced steering frame velocity** contains a coerced value of the **steering frame velocity** if any kinematic constraints of the steering frame cause the velocity to change from the input value. Otherwise, **coerced steering frame velocity** is the same as **steering frame velocity**.
- FW** **calculated wheel states** is an array of clusters that define the state of each wheel on the steering frame.
- ABC** **wheel name** contains the identifier for the wheel.
  - DBL** **forward angular velocity** contains the rate of rotation, specified in radians per second, of the wheel in the forward direction.
  - DBL** **caster angular velocity** contains the rate of rotation of a caster wheel around the wheel pivot that attaches to the steering frame. If the wheel is not a caster wheel, LabVIEW returns 0.
  - DBL** **caster wheel angle (rad)** contains the orientation of a caster wheel with respect to the wheel pivot that attaches to the steering frame. If the wheel is not a caster wheel, LabVIEW returns 0.
  - DBL** **steering angle (rad)** contains the orientation of the wheel with respect to the wheel frame. If the wheel is not a steering wheel, LabVIEW returns 0.
- Err** **error out** contains error information. This output provides [standard error out](#) functionality.

**Figure 0-30. Apply Steering Frame Velocity to Wheels help information**

The steering frame velocity input is an array data type (explained in the next section of this experiment) of three double precision floating point values:

$x\_dot$  = lateral velocity which is 0 for a differential drive robot like DaNI 2.0  
 $y\_dot$  = forward velocity  
 $\theta\_dot$  = angular velocity

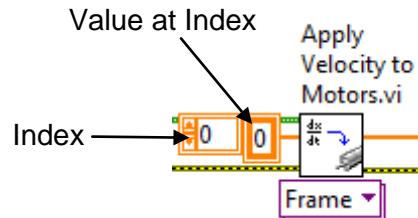
If a velocity that is higher than the maximum velocities set for DaNI 2.0, the value is coerced down to the maximum allowed velocities. The maximum forward velocity is 0.5 and the maximum angular velocity is 2 rads/s. Note that these values are set at 0, 0.1, and 0.3 respectively in Figure 0-26.

## Experiment 4-5 Grouping Steering Frame and Other Data in LabVIEW with Arrays and Clusters

The array data type groups elements together in memory and gives them a common name. In this instance, one name "steering frame velocity" represents all three values. You don't need three separate names. In addition, these three values are stored in contiguous (adjacent)

memory. An array consists of elements and dimensions. Elements are the data that make up the array. A dimension is the length, height, or depth of an array. An array can have one or more dimensions and as many as  $(2^{31}) - 1$  elements per dimension, memory permitting. The array used here is one dimensional. Refer to LabVIEW Help for information on multidimensional arrays. You can build arrays of numeric, Boolean, path, string, waveform, and cluster data types (explained below). The array of interest here is numeric. Array elements are ordered. An array uses an index so you can readily access any particular element. The index is zero-based, which means it is in the range 0 to  $n - 1$ , where  $n$  is the number of elements in the array. In this instance there are 3 elements so  $n = 3$ . There are three index values 0, 1, and 2. Y dot is at index 1. So to refer to the value of Y dot, use steering frame velocity(1).

To create the array constant for the block diagram, place the Apply Velocity to Motors VI on the block diagram, right click the connection for the array and choose create constant. The constant will appear as shown in Figure 0-31. You could also select an array constant on the Functions palette, place the array shell on the block diagram, and place a DBL numeric constant in the array shell. When it is first placed on the block diagram, the array has two rectangles for displaying and changing values. The left rectangle with the increment and decrement buttons in the index. If the elements in the array have different values, when you increment the index, the value at the index will change to show what value is stored at that index.



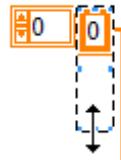
**Figure 0-31. Array constant**

Because there are index and value rectangles in array objects, you must use the mouse carefully when selecting. You can select the entire array by carefully choosing the outer border, or you can select the element value only as shown in Figure 0-32. Select the entire array and move it to the position shown in Figure 0-26.



**Figure 0-32. Selecting array components**

Since the array is small, you can display all three elements without cluttering the block diagram, so grab the lower handle and drag it down as shown in Figure 0-33. If you try to display a column or row that is out of the range of the array dimensions, the array appears dimmed to indicate that there is no value defined, and LabVIEW displays the default value of the data type.



**Figure 0-33. Show all three elements of the array**

To give values to the elements, double click the element and type in the value. Notice that there are two cases in Figure 0-26 with different values. One case is the rotate case where the lateral and forward velocity values are 0. The other case is the drive case where the lateral and angular values are 0.

The coerced steering frame output from the Apply Velocity to Motors VI that is wired to the Motor Velocity Setpoints VI is also an array, but it only has two values, the setpoints for the left and right motors. To make this match the inputs required for the Motor Velocity Setpoints VI, the values have to changed from an array data type to individual DBL floating point scalar values. This can be done with the Index Array Function as shown in Figure 0-26.

Arrays are very useful data structures, so LabVIEW contains a large number of functions as shown in Figure 0-34 to manipulate them.

Palette Object	Description
<a href="#">Array Constant</a>	Use this constant to supply a constant array value to the block diagram.
<a href="#">Array Max &amp; Min</a>	Returns the maximum and minimum values found in <b>array</b> , along with the indexes for each value.
<a href="#">Array Size</a>	Returns the number of elements in each dimension of <b>array</b> .
<a href="#">Array Subset</a>	Returns a portion of <b>array</b> starting at <b>index</b> and containing <b>length</b> elements.
<a href="#">Array To Cluster</a>	Converts a 1D array to a cluster of elements of the same type as the array elements. Right-click the function and select <a href="#">Cluster Size</a> from the shortcut menu to set the number of elements in the cluster.
<a href="#">Array To Matrix</a>	Converts an array to a matrix of elements of the same type as the array elements. Wire data to the <b>Real 2D Array</b> input to determine the polymorphic instance to use or <a href="#">manually select</a> the instance.
<a href="#">Build Array</a>	Concatenates multiple arrays or appends elements to an n-dimensional array.
<a href="#">Cluster To Array</a>	Converts a cluster of elements of the same data type to a 1D array of elements of the same data type.
<a href="#">Decimate 1D Array</a>	Divides the elements of <b>array</b> into the output arrays, placing elements into the outputs successively.
<a href="#">Delete From Array</a>	Deletes an element or subarray from <b>n-dim array</b> and returns the edited array in <b>array w/ subset deleted</b> and the deleted element or subarray in <b>deleted portion</b> .
<a href="#">Index Array</a>	Returns the <b>element or subarray</b> of <b>n-dimension array</b> at <b>index</b> .
<a href="#">Initialize Array</a>	Creates an n-dimensional array in which every element is initialized to the value of <b>element</b> .
<a href="#">Insert Into Array</a>	Inserts an element or subarray into <b>n-dim array</b> at the point you specify in <b>index</b> .
<a href="#">Interleave 1D Arrays</a>	Interleaves corresponding elements from the input arrays into a single output array.
<a href="#">Interpolate 1D Array</a>	Linearly interpolates a decimal <b>y value</b> from an <b>array of numbers or points</b> using a <b>fractional index or x</b> value.
<a href="#">Matrix To Array</a>	Converts a matrix of elements to an array of elements of the same data type. Wire data to the <b>Real Matrix</b> input to determine the polymorphic instance to use or <a href="#">manually select</a> the instance.
<a href="#">Replace Array Subset</a>	Replaces an element or subarray in an array at the point you specify in <b>index</b> .
<a href="#">Reshape Array</a>	Changes the dimensions of an array according to the values of <b>dimension size 0..m-1</b> .
<a href="#">Reverse 1D Array</a>	Reverses the order of the elements in <b>array</b> .
<a href="#">Rotate 1D Array</a>	Rotates the elements of <b>array</b> the number of places and in the direction indicated by <b>n</b> .
<a href="#">Search 1D Array</a>	Searches for an <b>element</b> in a <b>1D array</b> starting at <b>start index</b> . Because the search is linear, you need not sort the array before calling this function. LabVIEW stops searching as soon as the element is found.
<a href="#">Sort 1D Array</a>	Returns a sorted version of <b>array</b> with the elements arranged in ascending order.
<a href="#">Split 1D Array</a>	Divides <b>array</b> at <b>index</b> and returns the two portions.
<a href="#">Threshold 1D Array</a>	Interpolates points in a 1D array that represents a 2D non-descending graph. This function compares <b>threshold y</b> to the values in <b>array of numbers or points</b> starting at <b>start index</b> until it finds a pair of consecutive elements such that <b>threshold y</b> is greater than the value of the first element and less than or equal to the value of the second element.
<a href="#">Transpose 2D Array</a>	Rearranges the elements of <b>2D array</b> such that <b>2D array[i,j]</b> becomes <b>transposed array[j,i]</b> .

**Figure 0-34. Array functions**

Another output from the Apply Velocity to Motors VI, calculated wheel states, is a cluster data type. Like arrays, clusters group data elements. But, unlike arrays whose elements must be of the same type, cluster elements can be different types. Clusters also differ from arrays in that they are a fixed size. Like an array, a cluster is either a control or an indicator. A cluster cannot contain a mixture of controls and indicators. An example of a cluster is the LabVIEW error cluster, which combines a Boolean value, a numeric value, and a string. A cluster is similar to a record or a struct in text-based programming languages.

Bundling several data elements into clusters eliminates wire clutter on the block diagram and reduces the number of connector pane terminals that subVIs need. The connector pane has, at most, 28 terminals but that pattern is very difficult to wire. The default 4x2x2x4 has 12. If your front panel contains too many controls and indicators that you want to pass to another VI, group some of them into a cluster and assign the cluster to a terminal on the connector pane.

Most clusters on the block diagram have a pink wire pattern and data type terminal, but error clusters have a dark yellow wire pattern and data type terminal, and clusters of numeric values, sometimes referred to as points, have a brown wire pattern and data type terminal. You can wire brown numeric clusters to Numeric functions, such as Add or Square Root, to perform the same operation simultaneously on all elements of the cluster.

You can unbundle all cluster elements at once using the Unbundle function. You can use the Unbundle By Name function to unbundle individual or a subset of cluster elements by name. The Unbundle by Name function is good to use because it self documents your code.

Cluster elements have a logical order unrelated to their position in the shell. The first object you place in the cluster is element 0, the second is element 1, and so on. If you delete an element, the order adjusts automatically. The cluster order determines the order in which the elements appear as terminals on the Bundle and Unbundle functions on the block diagram. You can view and modify the cluster order by right-clicking the cluster border and selecting Reorder Controls In Cluster from the shortcut menu.

To wire clusters to each other, both clusters must have the same number of elements. Corresponding elements, determined by the cluster order, must have compatible data types. For example, if a double-precision floating-point numeric value in one cluster corresponds in cluster order to a string in the another cluster, the wire on the block diagram appears broken and the VI does not run. If the numeric values are different representations, LabVIEW coerces them to the same representation.

Use the Cluster functions to create and manipulate clusters. For example, you can perform tasks similar to the following:

- Extract individual data elements from a cluster.
- Add individual data elements to a cluster.
- Break a cluster out into its individual data elements.

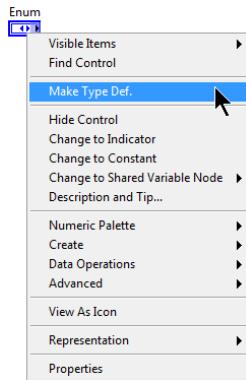
Create a cluster indicator for the application by right clicking on the connector and choosing create indicator. You can also create a control or indicator on the front panel by adding a cluster shell to the front panel, as shown in the following front panel, and dragging a data object or element, which can be a numeric, Boolean, string, path, refnum, array, or cluster control or indicator, into the cluster shell.

## **Experiment 4-6 LabVIEW State Machine Architecture to Drive from Start to Goal with the Steering Frame**

The previous VI for traveling from Point A to B, used a sequence of 2 subVIs. If you have a large number of tasks to sequence, or if the sequence changes conditionally, the state machine architecture can be combined with modular programming. The state machine is simple but powerful. It makes code readable, maintainable, and scalable. It is simply created from a loop, a Case structure, an enumerated constant, and Shift Registers.

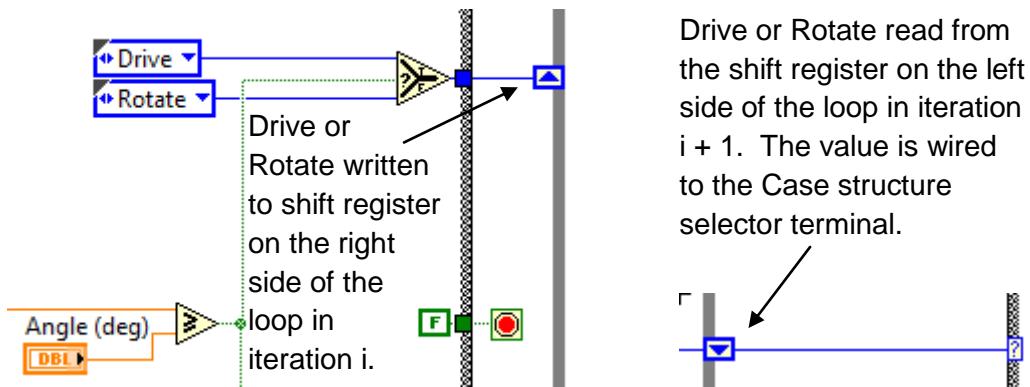
To create a state machine, add a loop to the block diagram and place a Case structure inside the loop. Then create an enumerated control on the front panel as explained previously with the

names of the states as the items. In this application, there are only two states: rotate and drive. Right click the control on the front panel or the icon on the block diagram and choose Make Type Def as shown in Figure 0-35, so if you change the control, all of the constants on the block diagram will automatically update. Type Def means Type Defined Control. This is a great time saver if you have a state machine with hundreds of constants.



**Figure 0-35. Change the control to a type defined control**

Figure 0-26 shows a constant, not a control, wired to a shift register. Right click the enum control and choose Create >> Constant. Wire the constant through a tunnel to the Selector terminal of the Case Structure. Right click the tunnel and choose Replace with Shift Register and wire as shown in Figure 0-26. The Case Labels change from True and False to Rotate and Drive. In the rotate state, the code should command DaNI 2.0 to turn so it is oriented toward the goal. The Drive state should execute after the Rotate state. So, at the end of each iteration, the code checks if DaNI 2.0 has rotated far enough. If not, it runs the rotate code again to rotate a little farther. If it has reached the target, it passes execution to the Drive state with the state transition code shown in Figure 0-36. If the comparison between the measured angle and the target is True, Drive is written to the shift register. If it is False, Rotate is written to the shift register. At the next iteration of the loop, the value in the shift register is read and sent to the Case Structure Selector Terminal as shown.



**Figure 0-36. State transition code**

In the Drive State, the True value from the comparison terminates the program if driving to only one point, but you can envision further iterations of the states to drive a series of points.

In addition to the Shift Register for the state transition, there are other Shift Registers to share data between states and accumulate drive and angle data for both motors. You can reduce the number of wires crossing the code and the number of Shift Registers by grouping data into arrays or clusters.

The remaining code in the VI is copied from previous programs, specifically the subVIs in the Rotate and Drive VI. Complete the code in the state machine, save the project, save the VI, and test it.

Instead of the subVI code in the two states, modify the subVIs and place them in the states to integrate modular programming with the state machine architecture.

What is the effect of changing the wait constant to 20 ms? If the forward and angular velocities are increased, how would it affect the value of the wait constant?



# **Experiment 5 – Perception with PING)))**

## **Instructor's Notes**

This experiment requires that the previous experiment be completed. Similar experimental area and tools used in the previous experiment are used here.

## **Goal**

Discover how to extract simple features like lines from ultrasonic data.

## **Required Components**

Similar experimental area to that used in previous experiments.

Linear distance measuring tool like a ruler, meter stick, or tape measure.

Angle measuring tool like a protractor.

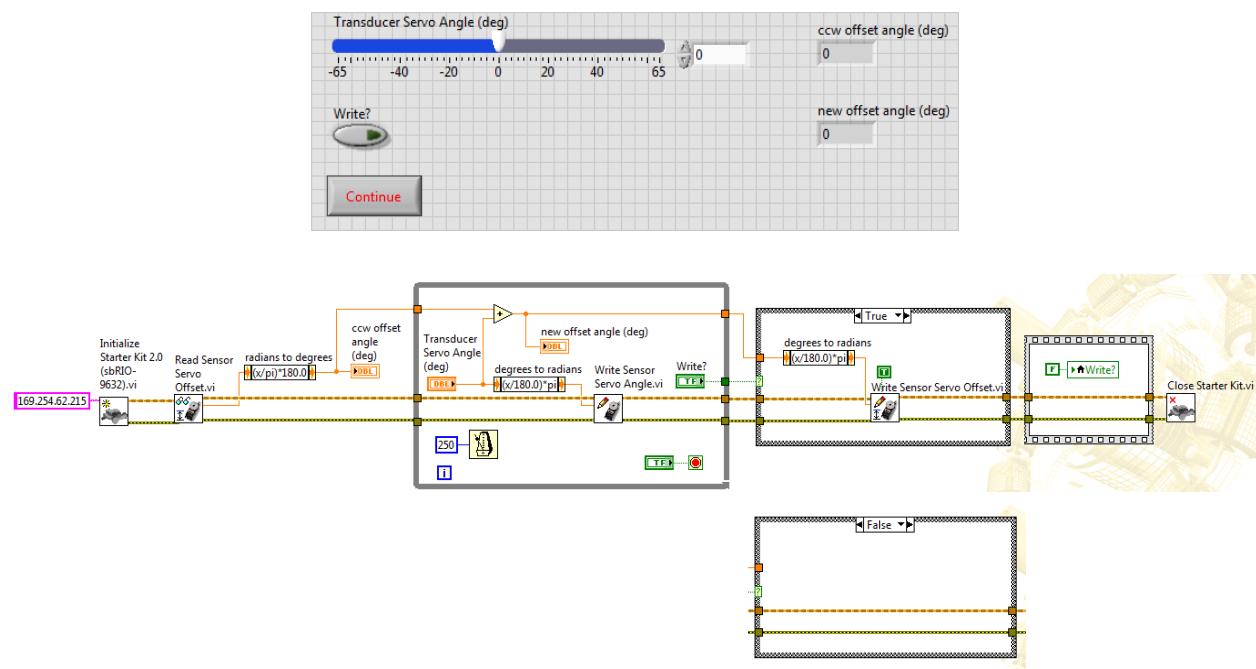
## **Background**

Students should study Siegwart et al (2011) chapter 4. This experiment requires that the previous experiments have been completed.

## **Experiment 5-1 Calibrating PING)))'s Orientation and File IO**

Perception means perceiving or “seeing” things in the environment where DaNI operates. Previous experiments revealed that PING))) can see some of the obstacles in the environment but there are many things it can’t perceive. PING))) locates obstacles relative to DaNI’s orientation and position in the local coordinate frame. In order to accomplish this you wrote a VI to control the servo relative to DaNI’s orientation. Recall experiment 2 where you characterized PING))) and experiment 3 where you discovered how to control the servo, and especially the Pan and Avoid Collision VI in Figure 0-28.

It is possible to write the offset of PING)))'s orientation to a file so that each time you start an experiment, PING))) is oriented forward. Create the VI in Figure 0-1. This VI displays the current offset before the loop executes. The user can adjust the servo angle until PING))) is oriented so it points forward in the loop. After the user is satisfied with the orientation, the user can write the value to a file on the SbRIO by change the value of the Write? Boolean control to True and then pressing the Continue button to terminate the loop. When the loop terminates, the new offset angle value is passed out of a loop tunnel to a tunnel in a Case structure. The Write? value is passed through a loop tunnel to the Case structure selector terminal. If the value is True, the Write Sensor Servo Offset VI with a True constant input executes. If the value of Write? is False, no code is executed in the False Case.



**Figure 0-1. Calibrate and Save Servo Offset Angle VI**

The Write Sensor Servo Offset VI writes the new offset angle to a file by calling the Save Offset Angle to File SubVI whose block diagram is shown in Figure 0-2. As shown by the three functions in sequence, it opens a file, writes to it, and closes the file. These functions are in the Functions>>Programming>>File I/O palette. The Open/Create/Replace function opens an existing file, creates a new file, or replaces an existing file, programmatically or interactively using a file dialog box. You can specify the operation to perform by choosing one of the items in the enum constant input to the function:

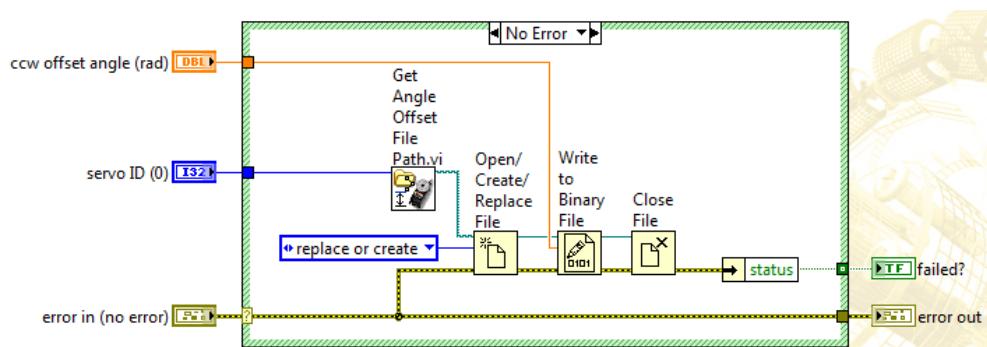
- open (default)—Opens an existing file.
- replace—Replaces an existing file.
- create—Creates a new file.
- open or create—Opens an existing file or creates a new file if one does not exist.
- replace or create—Creates a new file or replaces a file if it exists.

- replace or create with confirmation—Creates a new file or replaces a file if it exists and the user gives permission.

The input is set to create or replace In this instance. There is a second input allowing you to select read/write (the default), read-only, or write-only that is not used in this instance since the default is appropriate.

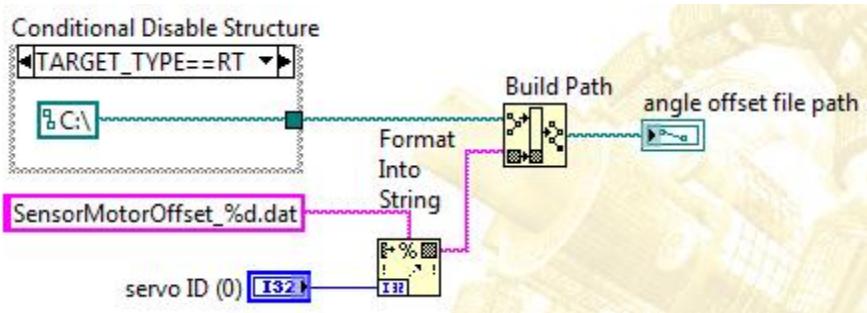
The next function in the sequence, Write to Binary File is used to save space on the sbRIO. There are several file types available in LabVIEW, and two of the most common are binary and text files. Use text (or ASCII) files if you want to make your data available to other applications, such as Microsoft Excel, because they are the most common and the most portable. Use binary files if you need to perform random access file reads or writes or if speed and compact disk space are crucial, because they are more efficient than text files in disk space and in speed. You will use text files later in this experiment to write data to the host hard drive.

The last function in the sequence, Close File, destroys the reference, frees space on the sbRIO for other uses, and prevents memory leaks.



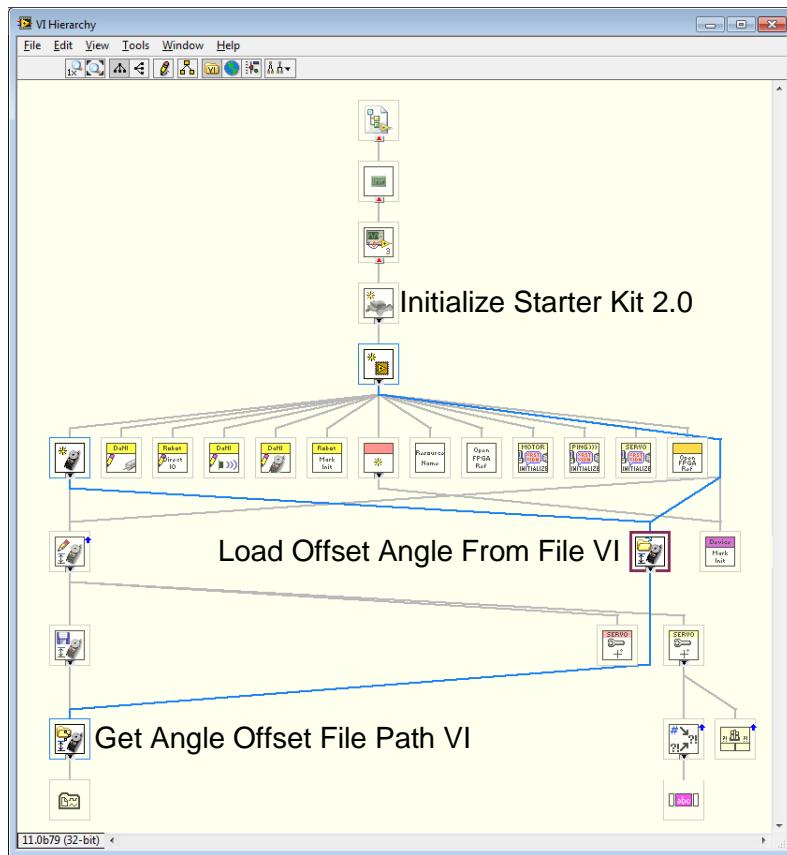
**Figure 0-2. Save Offset Angle to File SubVI block diagram**

The Get Angle Offset File Path, whose block diagram is shown in Figure 0-3, provides the file path input information for the Open/Create/Replace File function. It is generic for a variety of robots. It uses the Conditional Disable Structure to select between an RT target that contains a hard drive with C:\ designation and a user defined target. The RT target is appropriate for this instance. The structure contains a path data type constant with value of C:\. The Conditional Disable Structure is similar to a Case Structure, except you use it to disable specific sections of code on the block diagram based on some user-defined condition. The output is a portion of the file path. More information is available in LabVIEW Help. In this instance it is concatenated with SensorMotorOffset\_%d.dat and the servoid whose representation is changed from a 32-bit signed integer to a string with the Format Into String function. The servo ID is necessary in general because the subVI could be used in an application where the robot had multiple servo motors.



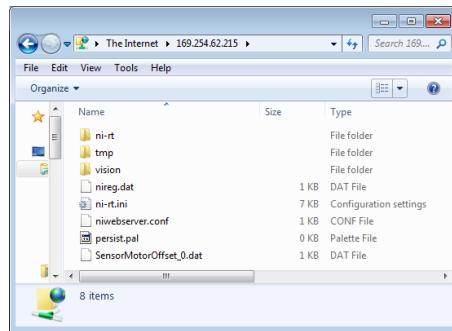
**Figure 0-3. Get Angle Offset File Path subVI**

When the Initialize Starter Kit 2.0 VI executes, it calls a series of subVIs in a hierarchy as shown in the annotated VI Hierarchy window (Tools>>VI Hierarchy). One of the subVIs, The Get Angle Offset File Path inputs the path to another subVI, the Load Offset Angle from File, which places the offset value in memory, so whenever you execute a VI that contains the Initialize VI, the last offset value saved to the file is automatically used to point the servo and PING))).



**Figure 0-4. Load servo offset information path in the VI Hierarchy window**

You can use ftp to locate the sensor motor offset and other files on the sbrio. Open Windows Explorer. Enter `ftp://<IP address>/` as shown in Figure 0-5. Since the file is binary you can't view the contents without writing a translator in LabVIEW.



**Figure 0-5. Displaying files on the sbRIO with Windows Explorer**

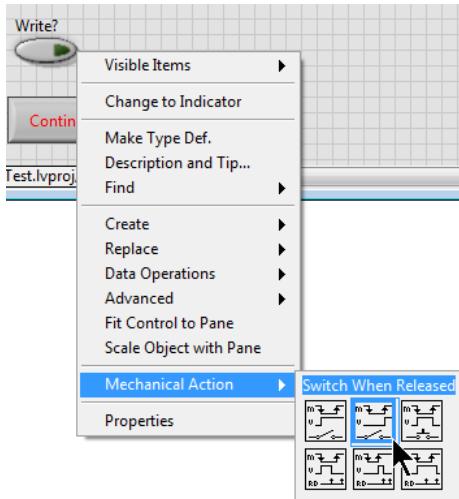
After the user adjusts the offset to the desired position, they press the Write? Boolean control. Unless you modify its mechanical action to switch when released, it will change back to False before the user can click the Continue button.

LabVIEW Boolean controls may behave in one of six different ways or mechanical actions. You can change the action by right clicking the control, selecting Mechanical Action, and selecting an action from the window as shown in Figure 0-6. There are two basic mechanical actions: switch and latch.

- Switch will return to its default state when the user clicks it. You might think of this as a traditional light switch.
- Latch will return to its default state when the user clicks it or when its value has been read by LabVIEW. You might think of this as a push button that pops back up automatically.

Each Boolean control can be configured to change state when pressed, when released, or until released.

- When-pressed changes state on a click and remains there until another click.
- Shen-released changes state on a click release and remains there until another until another click release (switch or latch) or when read by LabVIEW (latch).
- Until-released changes state on a click that is held down and changes when the click is released (switch or latch) or when read by LabVIEW (latch).



**Figure 0-6. Change the mechanical action of the Boolean Write? control**

The switch mechanical action requires that you click it again to change it back to the default value of False. So you don't have to remember to do that every time you run the VI, you can add some code to automatically reset it to False. This code is shown in the flat sequence structure where a False Boolean constant is wired to a local variable.

When you run a VI all of the front panel object properties are stored in memory. The local variable gives the code access to read from or write to the Write? Boolean control value. With a local variable, you can write to or read from a control or indicator on the front panel. Writing to a local variable is similar to passing data to any other terminal. However, with a local variable you can write to it even if it is a control or read from it even if it is an indicator. In effect, with a local variable, you can access a front panel object as both an input and an output.

Right-click the front panel object or the block diagram terminal and select Create»Local Variable from the shortcut menu to create a local variable that you can add to the block diagram. Notice that the icon inherits the label of the control. You can right click the local variable icon and change it to read from or write to. In this instance choose write to. Right click the input connection and choose create>>constant to create a Boolean False constant. This will automatically set the value of the front panel control back to False.

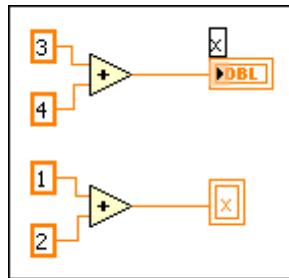
Be careful when using local variables. People with text programming experience who are beginning to learn LabVIEW tend to over use them. They are inherently not part of the LabVIEW dataflow execution model. Block diagrams can become difficult to read when you use local variables. Overusing local variables, such as using them to avoid long wires across the block diagram or using them instead of data flow, slows performance.

Since using a local variable gives you access to memory, you should initialize it. To initialize a local or global variable, verify that the variable contains known data values before the VI runs. Otherwise, the variables might contain data that causes the VI to behave incorrectly. If the

variable relies on a computation result for the initial value, make sure LabVIEW writes the value to the variable before it attempts to access the variable for any other action. If you do not initialize the variable before the VI reads the variable for the first time, the variable contains the default value of the associated front panel object.

A race condition can occur when two or more pieces of code execute in parallel and have access to a shared piece of memory like a local variable. If each piece of code is independent, there is no way to distinguish the order LabVIEW uses to access the shared resource. Race conditions can be dangerous because they can appear and disappear when the timing of a program changes. Although race conditions can exist any time more than one action updates the value of the same stored data, race conditions often occur with the use of local variables or an external file.

Figure 0-7 shows a race condition. Remember that the data flow paradigm in LabVIEW depends on wires, but there are no wires connecting the upper and lower operations. The output of this VI, the value of local variable  $x$ , depends on which operation runs first.  $3 + 4$  might run first, or  $1 = 2$  might run first since there are no wires connecting them. Because each operation writes a different value to  $x$ , there is no way to determine whether the outcome will be 7 or 3. This is known as a race condition. The VI might return 3 most of the time, but every now and then, it might return 7.



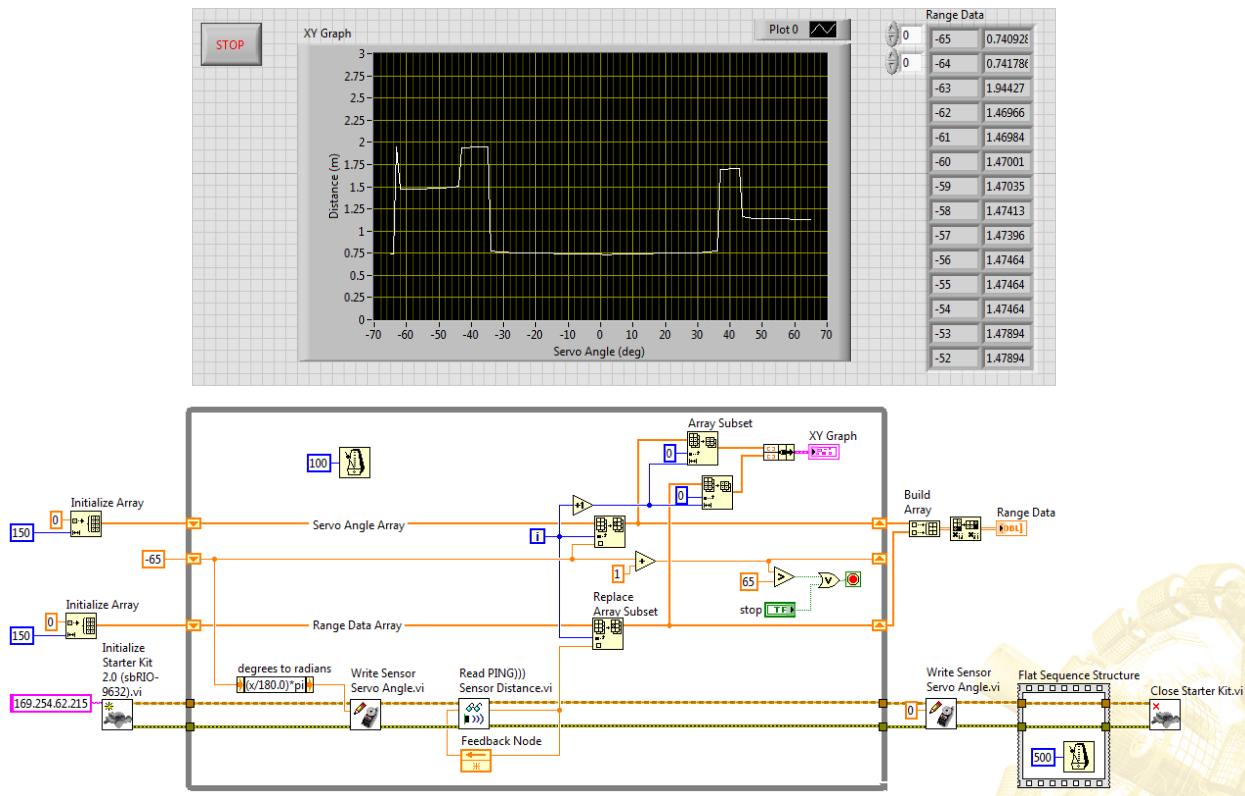
**Figure 0-7. Race condition**

Local variables make copies of data buffers. When you read from a local variable, you create a new buffer for the data from its associated control. If you use local variables to transfer large amounts of data from one place on the block diagram to another, you generally use more memory and, consequently, have slower execution speed than if you transfer data using a wire. If you need to store data during execution, consider using a shift register.

After completing the code for the servo calibration VI, save it and save the project. Run the VI. Point PING))) precisely in the forward direction and write the offset to file. Test the result by opening another VI that calls the initialize VI, like the Steering Frame Drive State Machine VI. Use ftp to view the date and time the servo offset file was written to the sbRIO.

# Experiment 5-2 Displaying Perception Data with an XY Graph

Use information from the VIs developed previously and additional information in the following to build the VI shown in Figure 0-8 that will acquire and display range data to obstacles using polar coordinates at  $1^\circ$  sample intervals while DaNI remains stationary and PING))) pans from  $-65^\circ$  to  $65^\circ$ .



**Figure 0-8. Acquire and Display Range Data VI**

Code the lower sequence that controls the servo angle and reads the range data by copying code from previous VIs like the Pan and Avoid Collision VI shown in Figure 0-28. You would like to display the data in polar coordinates (servo angle,  $\theta$ , on the x axis versus distance,  $\rho$ , on the y axis) on the front panel, but the chart and graph previously used are for time dependent data. You can use an XY Graph. The following lists the variety of charts and graphs in LabVIEW:

- **Waveform Graphs and Charts**—Display data typically acquired at a constant rate.
- **XY Graphs**—Display data acquired at a non-constant rate and data for multivalued functions.

- Intensity Graphs and Charts—Display 3D data on a 2D plot by using color to display the values of the third dimension.
- Digital Waveform Graphs—Display data as pulses or groups of digital lines.
- Mixed Signal Graphs—Display data types accepted by waveform graphs, XY graphs, and digital waveform graphs. Also accept clusters that contain any combination of those data types.
- 2D Graphs—Display 2D data on a 2D front panel plot.
- 3D Graphs—Display 3D data on a 3D front panel plot.

The XY graph is a general-purpose, Cartesian graph that plots data with a varying time base. The XY graph displays any set of points, evenly sampled or not. The XY graph can display plots containing any number of points in several data types. In this instance, you have a single plot. The XY graph accepts a cluster that contains an x array and a y array for single-plot XY graphs. So, your code needs to create two arrays, one of angle values for the x coordinates and one of distance values for the y coordinates.

The best programming practice when creating arrays is to use the initialize array function. Initializing will reserve memory prior to loop execution and won't require the operating system to reallocate memory while the loop is executing. Place two Initialize Array functions on the block diagram as shown in Figure 0-8. Configure them for single dimension arrays of 130 elements with initial value of 0 as shown.

Place code in the loop that replaces the 0 values with the most recent acquired data using two Replace Array Subset functions as shown. Wire the initialized arrays to shift registers and wire from the shift registers to the Replace Array Subset functions. Wire the output of the While loop Iteration terminal to the index connections to increment the location or index in the array as the loop iterates. When  $i = 0$ , the first 0 value will be replaced. When  $i = 1$ , the second will be replaced, etc. Wire the values of the angle and range acquired in the current iteration into the new element connections as shown. To display the values on the front panel, combine the 1D arrays into a 2D array on the output of the loop with a Build Array function configured without concatenation as shown and create an indicator.

Since you initialized the arrays to 130 elements, and the VI might terminate with less than 130 elements, it is possible that the graph would display several elements with the initialized value of 0. To avoid this, add two Array Subset functions on the block diagram. Configure the functions to output a subset of the array starting at element 0 and extending to length  $i + 1$  as shown in Figure 0-8.

Add an XY Graph to the front panel. Add a bundle function (cluster palette) to the block diagram. Wire the output of the two arrays to the bundle function and wire the cluster output from the bundle function to the XY graph. Configure the graph as shown on the front panel.

Add termination code that will automatically terminate after panning from  $-65^\circ$  to  $65^\circ$  or if the user presses the stop button.

Save the VI and the project and test it. Place an object like a box in front of DaNI and display the data acquired similar to that shown on the front panel of Figure 0-8.

## Experiment 5-3 Communicating Perception Data to the Host with Network Streams

Even though you can do a lot of analysis on the sbRIO, you may want to transfer the data to the host computer so you can do additional analysis there. The current VI runs on the sbRIO and displays information on the host monitor with front panel communication. With this technology, the host computer and the RT target execute different parts of the same VI. On the host computer, LabVIEW displays the front panel of the VI while the RT target executes the block diagram.

Front panel communication is a good communication method to use during development, because you can quickly monitor and interface with VIs running on an RT target. With network communication, a host VI runs on the host computer and communicates with the VI running on the RT target using specific network communication methods such as network-published shared variables, Network Stream functions, or other protocols like TCP. You might use network communication for the following reasons:

- To run another VI on the host computer.
- To control the data exchanged between the host computer and the RT target. You can customize the communication code to specify which front panel objects to update and when. You also can control which components are visible on the front panel because some controls and indicators might be more important than others.
- To control timing and sequencing of the data transfer.
- To perform additional data processing or logging.

Use the Network Streams functions for network communication when you need to transfer every point of data. Use cases include:

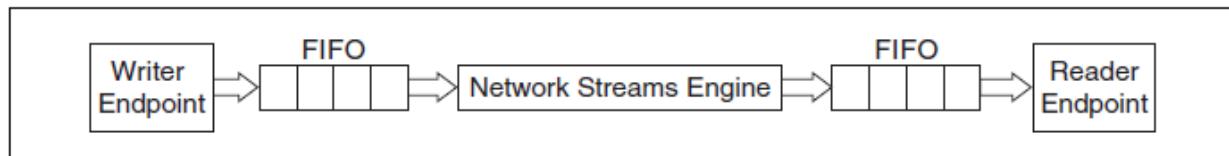
- Transferring data losslessly between RT target and host computer
- Transferring data from an RT target to host computer for logging data to file
- Transferring data from an RT target to host computer for data processing and analysis that requires more memory than the RT target has available

Stream data continuously between two LabVIEW applications with network streams. A network stream is a lossless, unidirectional, one-to-one communication channel that consists of a writer and a reader endpoint. You can use network streams to stream any data type between two applications, but the following data types stream at the fastest rates:

- Numeric scalars
- Booleans
- 1D arrays of numeric scalars
- 1D arrays of Booleans

You can accomplish bidirectional communication by using two streams, where each computer contains a reader and a writer that is paired to a writer and reader on the opposite computer. Each endpoint uses a FIFO buffer to transfer data. A buffer is a region of memory that stores data. FIFO means first in and first out. With a FIFO buffer, the writer adds data to the buffer in a sequence, and the reader removes it in the same sequence. The writer and reader can operate at different rates and no data will be lost if the buffer size is large enough and the applications are thoroughly tested. Network streams creates a buffer on the writer and another one on the reader. As shown in Figure 0-9, data flows in the following order:

1. The writer endpoint writes data to a FIFO buffer, on the sbRIO in this instance.
2. The Network streams software (called the Network Streams Engine or NSE) transfers data over a network (via the ethernet cable in this instance) to another FIFO buffer on the reader endpoint, the host computer in this instance.
3. The reader endpoint reads the data from that FIFO buffer.

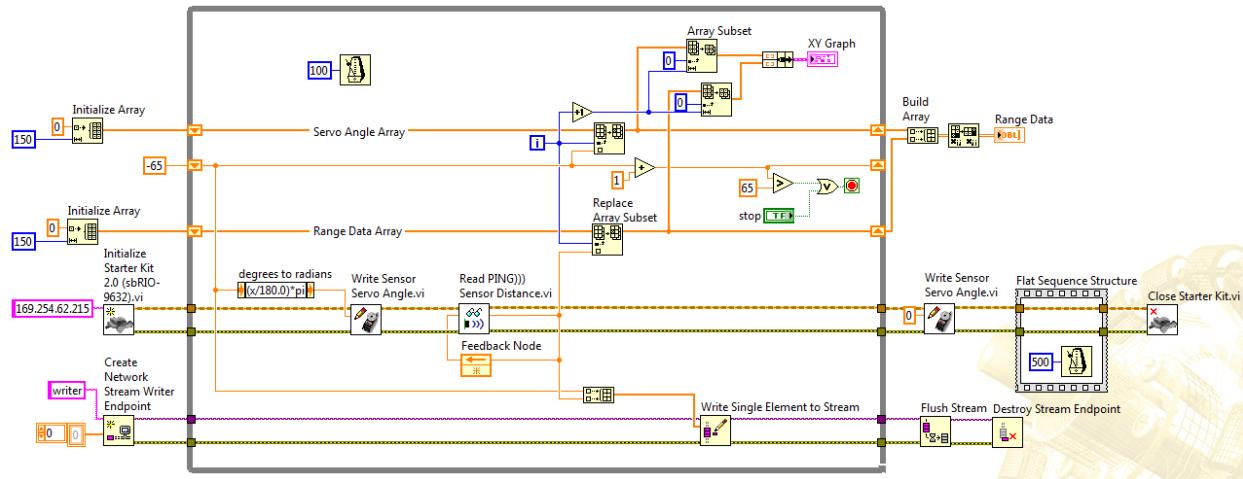


**Figure 0-9. Network streams concept diagram**

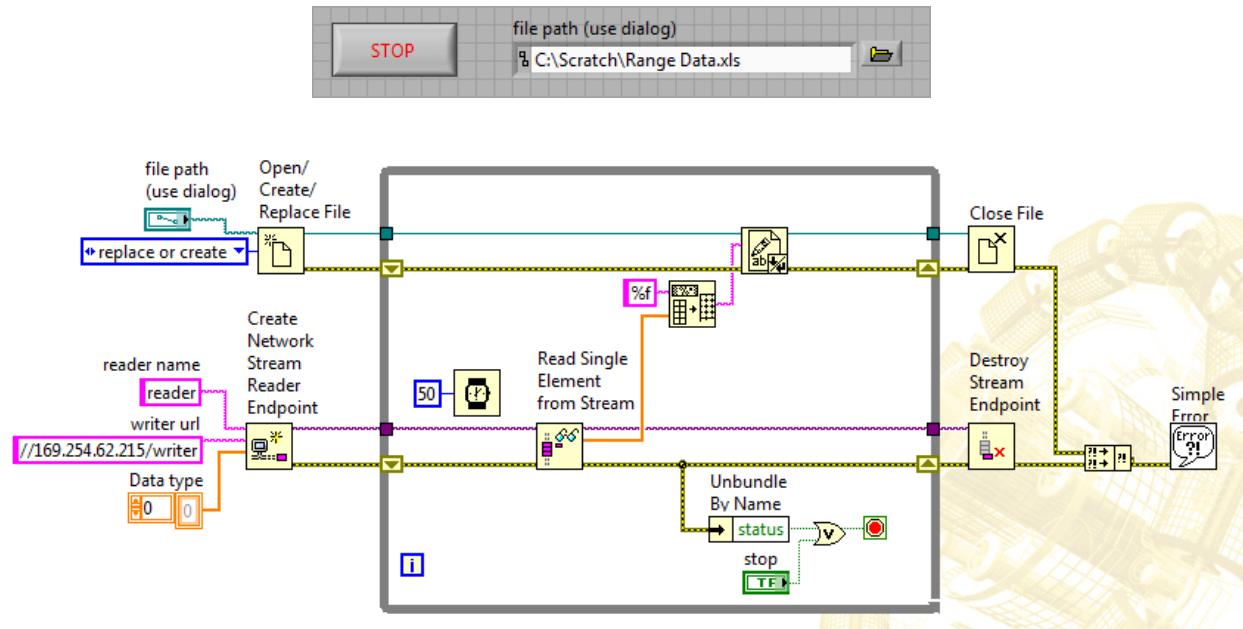
Complete the following steps to create and stream data with a network stream:

1. Create the endpoints and establish the stream connection.
2. Read or write data.
3. Destroy the endpoints.

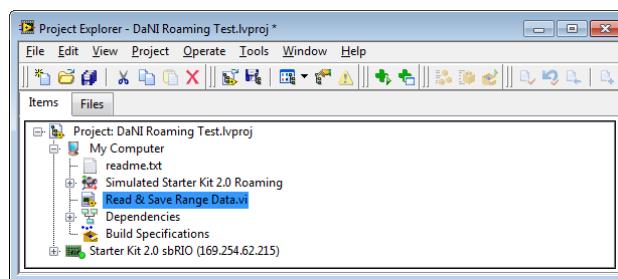
Use information from the VIs developed previously and additional information in the following to modify the Acquire and Display Range Data VI to build the Acquire Display & Write Range Data VI shown in Figure 0-10 and the Read Display & Save Range Data VI shown in Figure 0-11. The Read Display & Save Range Data VI should reside under My Computer in the project as shown in Figure 0-12 and not under the sbRIO target.



**Figure 0-10. Acquire Display & Write Range Data VI on the real-time sbRIO**



**Figure 0-11. Read & Save Range Data on the Windows host computer**



**Figure 0-12. Project showing Read Display & Save Range Data under the host computer**

LabVIEW identifies each stream endpoint with a URL (uniform resource locator). To connect two endpoints and create a valid network stream, you must specify the URL of a remote endpoint with the Create Network Stream Endpoint Reader or Create Network Stream Endpoint Writer function. The URL of a stream endpoint must contain at least

```
//host_name/endpoint_name
```

where:

- host\_name is the project alias, DNS name, or IP address of the computer on which the endpoint resides. In addition, the string “localhost” can be used as a pseudonym for the computer on which the code is currently executing. If host\_name is left blank, localhost is inferred.
- endpoint\_name is the actual name of the stream endpoint.

More detailed information on endpoint URLs can be found in the LabVIEW Help under the topic *Specifying Network Stream Endpoint URLs*.

A network stream is the connection between a writer and reader endpoint. Create the endpoint using the Create Network Stream Writer Endpoint and Create Network Stream Reader Endpoint functions. The endpoint name is supplied through the *writer name* or *reader name* input to the create function. LabVIEW uses this name to generate a URL for identifying the endpoint resource and must be unique. In addition to creating the endpoint resources, the create function also links the writer and reader endpoints together to form a fully functioning stream. The endpoint that is created first will wait until the other endpoint has been created and is ready to connect, at which point both create functions will exit. The stream is then ready to transfer data. Used in this manner, the create function effectively acts as a network rendezvous where neither create function will progress until both sides of the stream are ready or until the create call times out.

In order to establish a stream between the two endpoints, three conditions must be satisfied:

1. Both endpoints must exist. In addition, if both create functions specify the remote endpoint URL they are linking against, both URLs must match the endpoints being linked.
2. The data type of the two endpoints must match.
3. One of the endpoints must be a write only endpoint, and the other endpoint must be a read only endpoint.

If condition one isn't initially satisfied upon executing the create function, it will wait until the timeout expires, and then return a timeout error if the condition still hasn't been satisfied. The VIs in Figure 0-10 and Figure 0-11 use the default timeout of -1 meaning no time limit. If both endpoints specify the other endpoint to link against and only one of the URLs match, one create function will return an error, and the other will continue to wait until the timeout expires. At least one of the create endpoint functions needs to specify the URL of the other endpoint in either the *reader url* or *writer url* input. Failure to do so will guarantee that both create calls will eventually return a timeout error. The reader and writer endpoints can both specify each other's URLs in their create function, but only one is necessary.

Stream endpoints use a FIFO buffer to transfer data from one endpoint to the other. The buffer size is specified at the time of creation, and the buffer is allocated as part of the create call. The size of the buffer is specified in units of elements, where an element in the buffer represents a single value of the type specified by the data type of the stream. The maximum number of elements that can be read or written in a single call is limited by the endpoint buffer size. A request to read or write a number of elements that is larger than the endpoint buffer size will immediately result in an error. There is no simple formula for determining the optimal buffer size. You will often have to experiment with different buffer sizes during development to determine which settings work best for the needs of your application. Thoroughly test the application to confirm the buffer sizes. The VIs in Figure 0-10 and Figure 0-11 use the default size of 4096.

Network streams offer both single element and multiple element interfaces when reading and writing data. Single element reads and writes allow one element at a time to be added or removed from the buffer, while multiple element reads and writes allow data to be added or removed in batches. The VIs in Figure 0-10 and Figure 0-11 use single element interfaces.

Cease communication from either endpoint by calling the Destroy Stream Endpoint function. To ensure that all data written to the stream has been received by the reader before destroying the writer endpoint, first call the Flush Stream function and wait for the appropriate *wait condition* to be satisfied before destroying the writer endpoint. Only writer endpoints may call the flush function. The flush function first requests that any data still residing in the writer endpoint buffer be immediately transferred across the network. It then waits until either the specified wait condition has been met or the timeout expires. The Flush Stream function has two different wait conditions:

- All Elements Read from Stream
- All Elements Available for Reading

When you specify "All Elements Read from Stream," the flush function will wait until all data has been transferred to the reader endpoint and all data has been read from the reader endpoint before returning. The "All Elements Available for Reading" option will wait until all data has been transferred to the reader endpoint, but it will not wait until all data has been read from the

reader endpoint. Failure to call flush before destroying the writer endpoint means any data in transit or any data still residing in the writer endpoint buffer could be lost. The VIs in Figure 0-10 and Figure 0-11 use the default “All Elements Read from Stream”.

The Read Display & Save VI uses the Format Into String function to convert the data from DBL to string. It converts an array of any dimension to a table in string form, containing tabs separating column elements, a platform-dependent EOL (end of line) character separating rows, and, for arrays of three or more dimensions, headers separating pages. It requires a format string input. There are a large number of formats available that are explained in LabVIEW Help in the Format String Syntax topic. In this instance the simple %f for floating point is appropriate.

You have used error wires in the RT VIs created in previous experiments for the sbRIO without an explanation of error handling in LabVIEW. Additional error handling code was added to the Read & Save VI since it runs in the MS Windows OS on the host computer. From the previous discussion, you noticed that errors are generated when using Network Streams. Errors are also generated when handling files and in other I/O operations like serial, instrumentation, and data acquisition. Error checking tells you why and where errors occur. Without it, you know only that the VI does not work properly.

The error in and error out information are cluster data types that include the following components:

- Status is a Boolean value that reports TRUE if an error occurred.
- Code is a 32-bit signed integer that identifies the error numerically. A nonzero error code coupled with a status of FALSE signals a warning rather than an error.
- Source is a string that identifies where the error occurred.

By default, LabVIEW automatically handles any error when a VI runs by suspending execution, highlighting the subVI or function where the error occurred, and displaying an error dialog box. To disable automatic error handling for a subVI or function within a VI, wire its error out parameter to the error in parameter of another subVI or function or to an error out indicator. Note that the error out and error in parameters have been wired in previous VIs and in Figure 0-11.

Error handling in LabVIEW follows the dataflow model. Just as data values flow through a VI, so can error information. Wire the error information from the beginning of the VI to the end. Include an error handler VI at the end of the VI to determine if the VI ran without errors. The block diagram shown in Figure 0-11 uses the Simple Error Handler. Use the error in and error out clusters in each VI you use or build to pass the error information through the VI. The error clusters are flow-through parameters.

As the VI runs, LabVIEW tests for errors at each execution node. If LabVIEW does not find any errors, the node executes normally. If LabVIEW detects an error, the node passes the error to the next node without executing that part of the code. The next node does the same thing, and so on. At the end of the execution flow, LabVIEW reports the error.

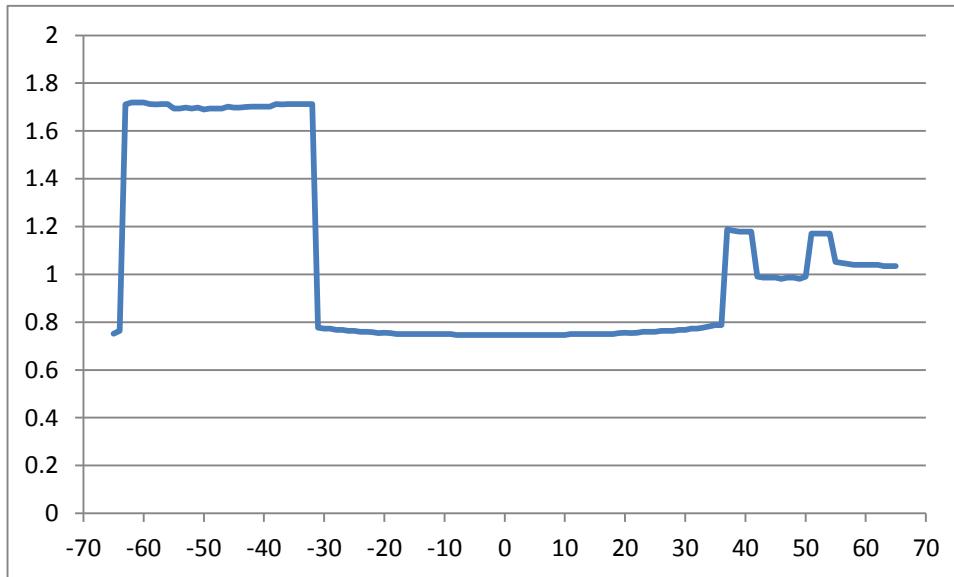
You can choose other error handling methods. For example, if an I/O VI on the block diagram times out, you might not want the entire application to stop and display an error dialog box. You also might want the VI to retry for a certain period of time. In LabVIEW, you can make these error handling decisions on the block diagram of the VI.

Some VIs, functions, and structures that accept Boolean data also recognize an error cluster. For example, you can wire an error cluster to a Boolean function or to the Boolean inputs of the Select or Stop functions to handle errors using logical operations. You can wire an error cluster to the conditional terminal of a While Loop to stop the iteration of the loop. If you wire the error cluster to the conditional terminal, only the TRUE or FALSE value of the status parameter of the error cluster passes to the terminal. If an error occurs, the loop stops. When you wire an error cluster to the selector terminal of a Case structure, the case selector label displays two cases—Error and No Error—and the border of the Case structure changes color—red for Error and green for No Error. If an error occurs, the Case structure executes the Error subdiagram.

Complete the modifications to the Acquire Display & Write VI and complete the Read & Save VI. Save the VIs and save the project. Test the VIs by running both simultaneously. An error dialog might appear when the Acquire Display & Write VI terminates and destroys the stream. Just click continue and the Read & Save VI will terminate. The file created by the Read & Save VI should be similar to Figure 0-13. Graph the data in a program like MS Excel as shown in Figure 0-14 and compare it with the physical objects in front of DaNI. You can run the Acquire and Display Range Data VI created previously that uses front panel communication in the same obstacle environment and compare the XY graph with the MS Excel graph.

-65	0.751059
-64	0.76325
-63	1.710917
-62	1.718815
-61	1.718815
-60	1.718815
-59	1.711947
-58	1.711088
-57	1.712118
-56	1.712118
-55	1.693402
-54	1.693402
-53	1.697695
-52	1.69323
-51	1.697695
-50	1.689453
-49	1.693917
-48	1.693402
-47	1.693574
-46	1.702159
-45	1.697695
-44	1.697695
-43	1.699927
-42	1.701816
-41	1.702331
-40	1.701816
-39	1.701816
-38	1.712118
-37	1.711088
-36	1.712118
-35	1.712118
-34	1.711947
-33	1.711947
-32	1.712118
-31	0.776815
-30	0.772522

**Figure 0-13. Sample Data from the Range Data.xls file**

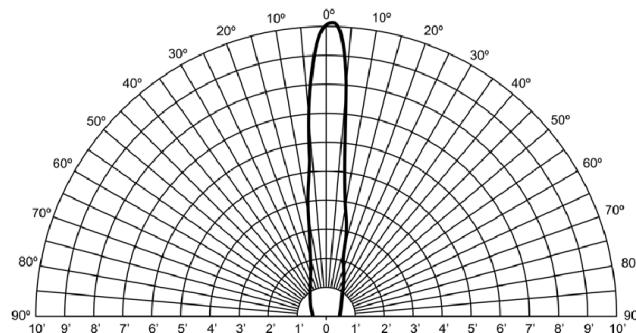


**Figure 0-14. MS Excel graph of range data file**

## Experiment 5-4 Feature Extraction - Identify Edges of an Obstacle

Refer to Siegwart et al (2011) chapter 4.7 and consider how to write a program that would automatically identify the obstacles in the range data set from the previous experiment. Since the range data is from panning in the xy plane, the only feature that can be extracted from the data are straight or curved lines.

The data points are the shortest distance to any obstacle in PING)))'s FOV so there is some error associated with each data point. If you recall from the specifications on the Parallel web site, the field of view varies with distance from the transducer as shown in Figure 0-15. At 1m, the FOV is about  $\pm 10^\circ$ , and the error is about  $1\text{m} \cdot \sin 10^\circ = 0.17 \text{ m}$ .



**Figure 0-15. PING))) FOV versus distance**

To write a program that will automatically identify the lines in the data from one scan of -65 to 65, you might begin with pseudo code that lists the steps the program must execute. The basic process is to identify the beginning and end points of a line starting at the beginning of the data set. Assuming straight, linear, features, one algorithm used by Siegwart et al (2011) is the split and merge algorithm based on least squares fit. The pseudo code is:

Simplify the data by replacing the angle values with 0 – 130.

Calculate the line for the first two points:

0	0.751059
1	0.76325

Evaluate the line for these points. The equation for the line between these two points is:  $y = 0.0122x + 0.7511$  in the form  $y = mx + b$  where  $b$  is the offset and  $m$  is the slope ( $0.76325 - 0.751059$ ).

Write the first point to a file or array of end points.

Iterate through a test to determine the end of the line.

Calculate the next y coordinate at ( $x = 2$ ) with the above equation. The result is  $y_2 = 0.77544$ .

Define a threshold for comparison of the difference between the predicted and actual values. For example, use 0.05 for the threshold.

If the difference is less than the threshold, the line continues. If not, you have found an end point and you start a new line.

The next point is:

2	1.710917
---	----------

The difference is 0.935 which exceeds the threshold so write the previous point as the end of the line and start a new line with the previous point.

Repeat the process until all of the range data has been processed.

Create a VI that will read the data and identify all of the line features by writing their start and end points to a file.

## **Experiment 5-5 Obstacle Avoidance**

Write a VI that will:

Create a file of endpoints (you can use the one from the previous section)

Read the line end points,

Determine

the closest end point,

the direction of the line,

which way to rotate to avoid the feature,

how much to rotate to avoid the feature (considering the 0.17 m error described previously),

how far to drive

and drive DaNI past the feature.

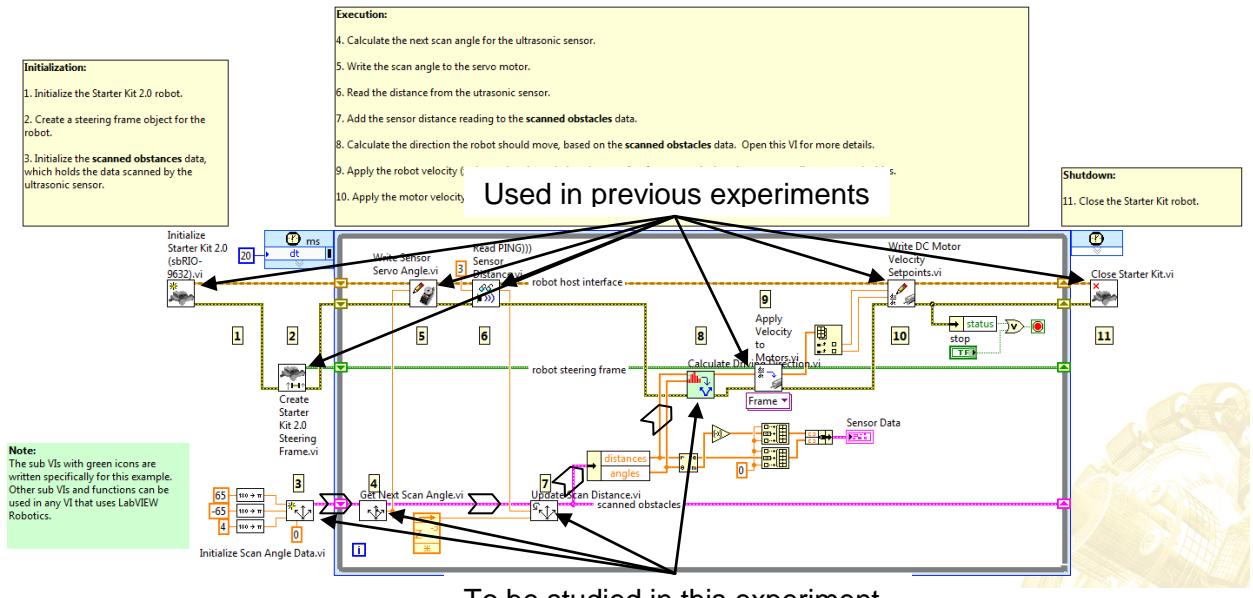
Test the VI in a simple environment with one obstacle in front of DaNI at the start.

## **Experiment 5-6 Follow a Wall**

Modify the VI from the previous experiment to follow the line closest to the robot which you will assume to be a wall. You only need to drive a very short distance parallel to the line. Set a safe operating distance from the “wall.” Start with DaNI facing the wall. Drive to the safe distance. Turn to an orientation parallel to the wall. Drive a short distance and stop.

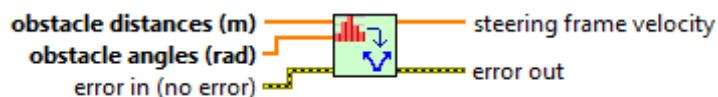
## **Experiment 5-7 Gap feature extraction in the Roaming VI**

Figure 0-16 shows the subVIs in the Roaming VI studied in previous experiments. The following will explain the remaining VIs that control panning, build a data structure of range data, analyze the data to extract gap features, rate the features to determine the largest gap, and calculate the driving direction to the largest gap.

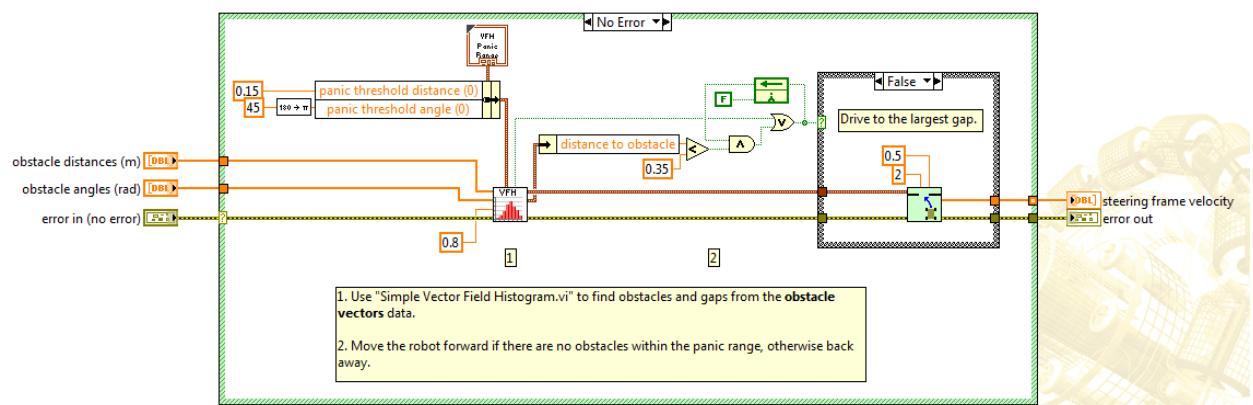


**Figure 0-16. Path to calculate the driving direction in the Roaming VI**

The Calculate Driving Direction VI accepts two arrays of distances and angles from the obstacles data structure as shown in Figure 0-17, and using a hierarchy of subVIs, determines the gaps, scores the gaps, calculates the direction the largest gap, and outputs the steering frame velocity.

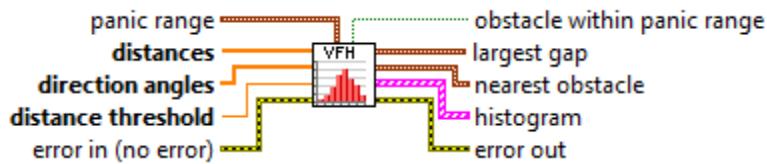


**Figure 0-17. Calculate Driving Direction VI input and output connections**



**Figure 0-18. Calculate Driving Direction block diagram**

The Calculate Driving Direction VI calls the Simple Vector Field Histogram (VFH) VI (Functions>>Robotics>>Obstacle Avoidance palette) whose connections are shown in Figure 0-19 that identifies obstacles and gaps, or open areas, in the robot environment. You can use the gap information in a variety of robots, not just DaNI 2.0, to implement reactionary motion. The VFH VI outputs the largest gap data, which is a cluster of two values, the location of the largest gap and the size of the gap. Largest gap describes the largest open area in the robot environment. Angle to gap contains the direction of the open area with respect to the robot sensor. Size of gap contains the angular size, or visual diameter, of the open area, measured as an angle.



**Figure 0-19. Simple Vector Histogram VI input and output connections**

The VFH VI also determines if there is an obstacle within the panic range. Panic range is an area of the environment to avoid, i.e. the obstacle is too close to continue driving forward and avoiding. The Roaming VI code commands DaNI to back away if there are obstacles in the panic range. The panic range data structure is a cluster of two elements. The panic threshold distance specifies the maximum distance at which obstacle within panic range is TRUE. Panic threshold angle specifies the maximum angle at which obstacle within panic range is TRUE. Obstacle within panic range is TRUE if an object within panic threshold angle is nearer to the sensor than the panic threshold distance.

Distances and direction angles inputs are obtained from the scanned obstacles data structure. Distances is an array of range data to obstacles. Direction angles is an array of angles with respect to the center of the sensor corresponding to each of the distances. Positive values of angles represent locations to the right of the center of the sensor, and negative values represent positions to the left of the center of the sensor.

The Distance threshold input defines obstacles. This VI ignores any objects at distances greater than distance threshold.

The nearest obstacle output is a data structure composed of a cluster of two elements. Angle to obstacle contains the direction of the nearest obstacle with respect to the sensor. Distance to obstacle contains the distance between the sensor and the nearest obstacle.

The VFH outputs data to a multiplot XY graph in the Histogram data structure. One plot is the location of the largest gap and the other is the distances to and locations of obstacles.

The VFH block diagram is shown in Figure 0-20. Panic range distance and angle values of 0.15 m and 45° and the distance threshold value of 0.8 m are provided via constants in the Calculate Driving Direction caller shown in Figure 0-18.

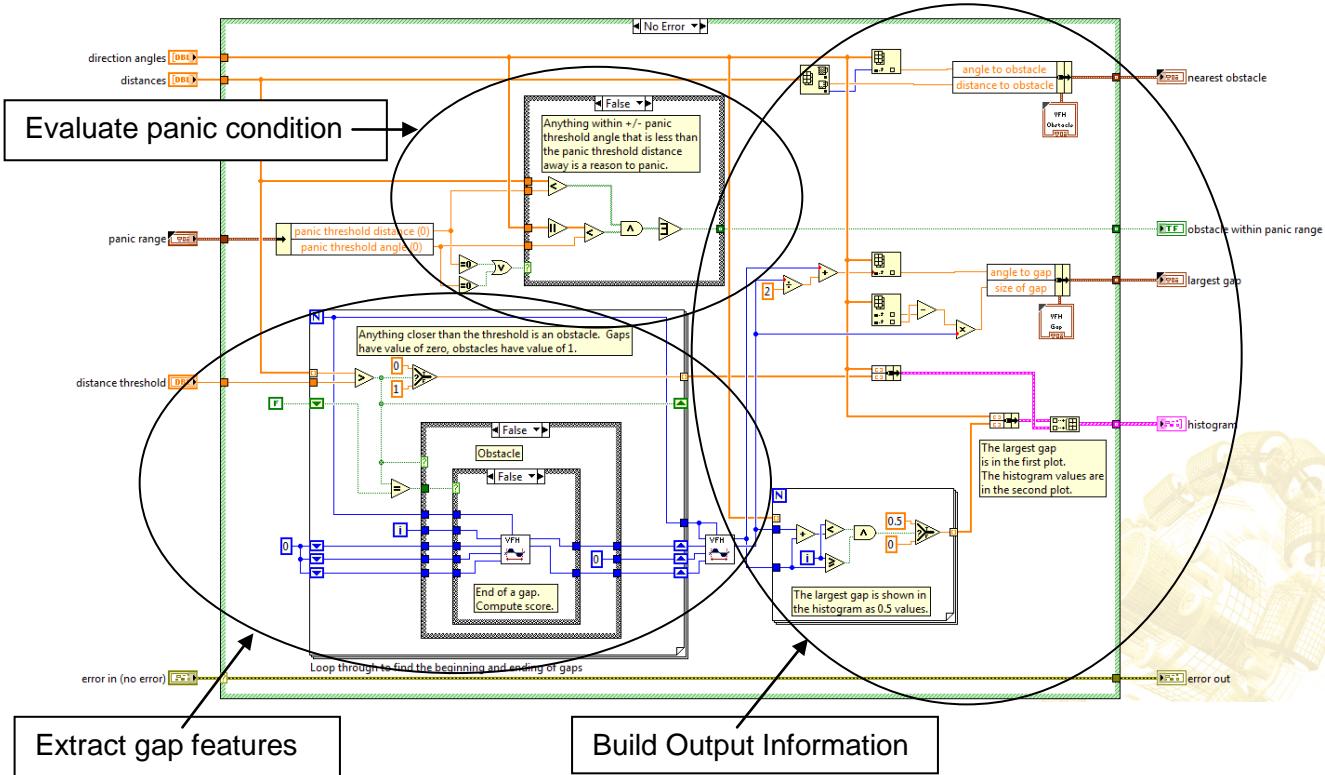
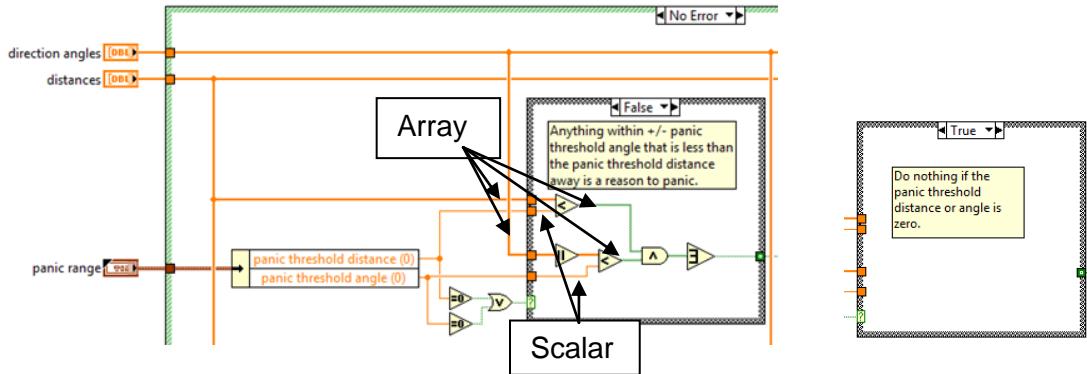


Figure 0-20. VFH block diagram

Figure 0-21 shows the segment of code that evaluates the panic condition. The panic range cluster is unbundled with an Unbundle By Name function into the panic threshold distance and panic threshold angle DBL scalars. Many LabVIEW functions are polymorphic so you can wire different data types to them. Some Comparison functions have two modes for comparing arrays or clusters of data. In Compare Aggregates mode, if you compare two arrays or clusters, the function returns a single Boolean value. In Compare Elements mode, the function compares the elements individually and returns an array or cluster of Boolean values, the mode used in this instance. In the False case of the Case structure, the upper Less? Function has one input that is the 1D array of distances. This is the thick array wire going to the upper connection on the function. The other input is the panic threshold distance DBL scalar, a thin scalar wire on the lower connection. In this configuration, the function compares all of the array values to the distance value and returns an array of Booleans. Note the thick output wire.



**Figure 0-21. Panic code segment of the VHF block diagram**

The direction angles array in Figure 0-21 is input to an Absolute Value function where all of the negative element element values in the array are changed to positive. The lower Less? Function makes a similar comparison between the absolute value of the distance angle array values and the panic threshold angle. The Boolean arrays from the two Less? functions are wired to a logical AND function. If both inputs of a logical AND are TRUE, the function returns TRUE. Otherwise, it returns FALSE. The output is wired to an OR Array Elements Function that returns a single value of FALSE if all the elements in Boolean array are FALSE or if Boolean array is empty. Otherwise, the function returns TRUE. If TRUE is output, the caller drives DaNI away from the nearest obstacle, if it returns FALSE, it drives DaNI toward the largest gap. If either the threshold distance or angle is 0, the True case executes which creates a FALSE output using a default-if- unwired tunnel.

Figure 0-22 shows the segment of code that extracts gap features. It uses a For loop to automatically extract each element of the distances array. The For loop border is shaped differently from the While loop used in previous VIs so you can easily distinguish between them. A For loop has a loop iteration terminal like a While loop, but it doesn't have a conditional terminal in its default configuration. Instead, it has a count terminal. The count terminal is an input terminal whose value indicates how many times to repeat the subdiagram. When the loop had finished iteration count times, it terminates automatically without the need for a Stop button or other input. Normally the count terminal is wired, but not in this instance. In this instance, the distances array passes through an input tunnel on the left side of the loop. This causes the count to inherit the number of elements in the array. In addition, the auto-indexing property of the input tunnel, disassembles the array and passes the elements into the loop one at a time, starting with the first element whose index is 0. Remember that the loop iteration value is also 0 on the first interation. You can disable auto indexing by right clicking the tunnel and choosing Disable Auto Indexing. In this instance, auto indexing is enabled on the array input (the diagram shows the thick array wire outside the tunnel and the thin scalar wire inside the loop), and disabled on the threshold input. Note the difference in the tunnels. The enabled tunnel has a white background with two brackets: [ ]. The disabled tunnel is filled with the data type color.

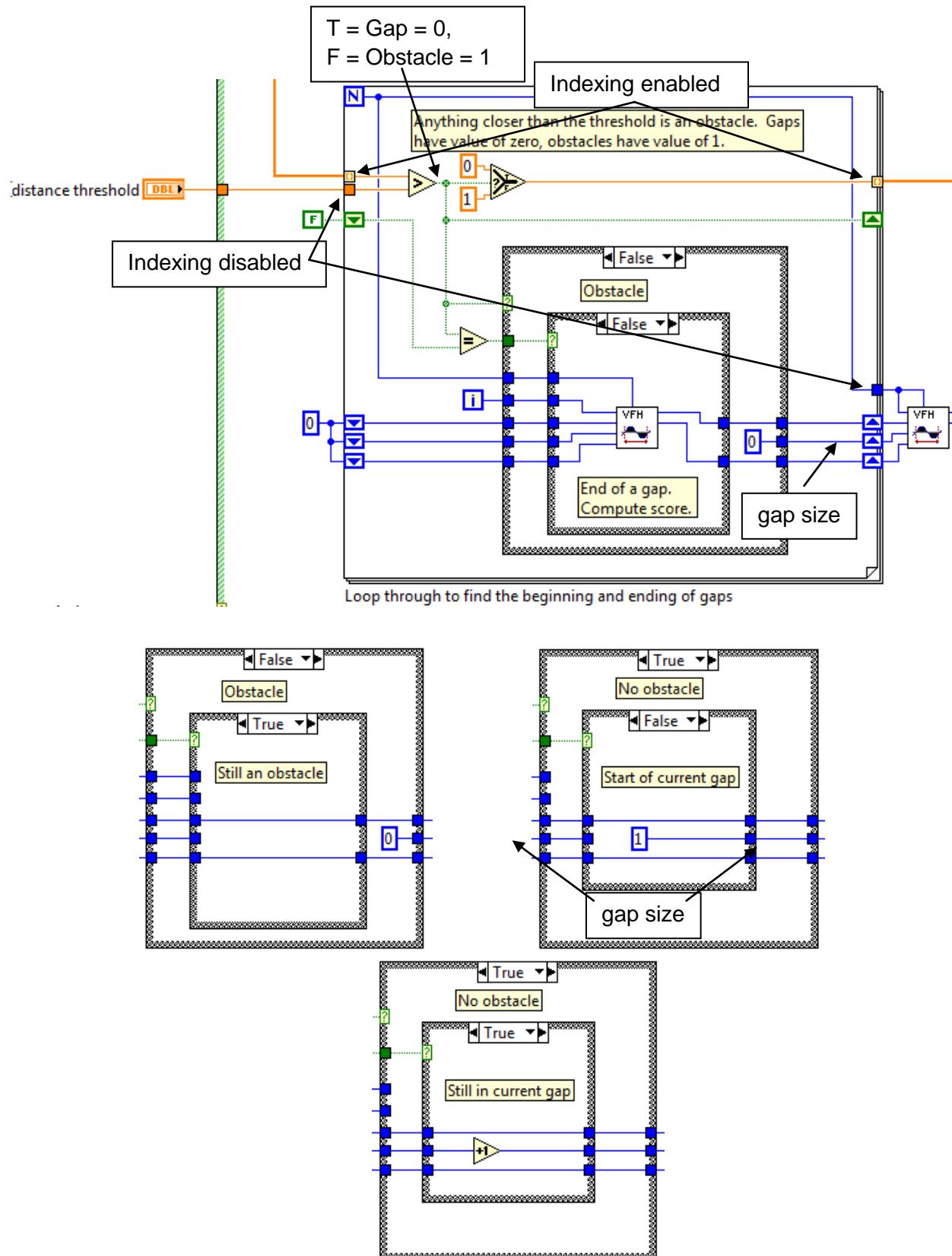
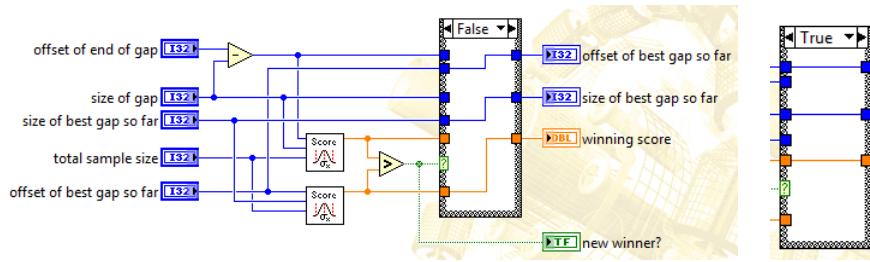


Figure 0-22. Gap feature extraction code segment of the VHF block diagram

To determine whether an individual distance value represents a gap or an obstacle, each element passed into the loop with auto indexing is compared to the distance threshold with a Greater? Function as shown in the upper portion of Figure 0-22. The Boolean output goes to a Select function that outputs 0 if the distance is greater than the threshold and 1 if it isn't. An output tunnel on the right side of the loop automatically reassembles the array. The result is that the values in the distances array are replaced by 0s and 1s where a gap is 0 and an obstacle is 1. Note that the output tunnel is configured to enable indexing which reassembles the array that was disassembled by the input tunnel. The new array has the same number of elements and is reassembled in the same order as the original array was disassembled. So, the input tunnel read all of the array values from memory and the output tunnel writes all of the array values back to memory.

The nested Case structures in Figure 0-22 call the Simple VFH Evaluate Current Gap VI whose block diagram is shown in Figure 0-23. It determines the beginning and end of gaps and the gap size. The outer Case structure selector input is the output of the gap comparison. If a gap was identified, the True case executes. The True case contains a nested Case structure whose selector input is a comparison with the state of the previous distance stored in a shift register. If the state was False, a new gap has started. If it is True, the current distance is in the same gap as the previous distance. If a new gap is identified, a value of 1 is written to the Size of Gap meaning it is a new gap with size = 1. If this distance continues the gap, the size is incremented by one. If a new gap is identified, the value of the loop index, I, is written to the offset of best gap so far. Note that the loop index is the same as the array index, so the offset is essentially the position of the value in the array.

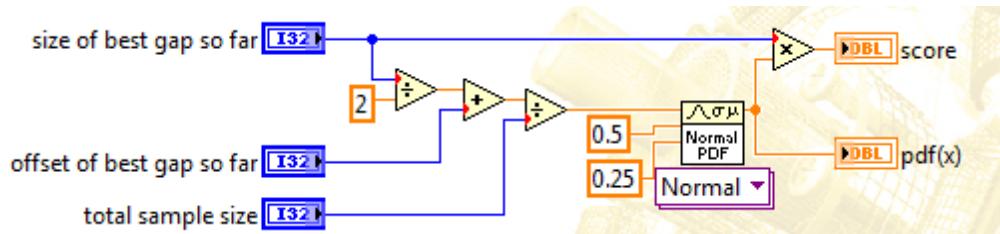


**Figure 0-23. Simple VFH Evaluate Current Gap VI block diagram**

If an obstacle was identified, the False case executes. The False case contains a nested Case structure whose selector input is a comparison with the state of the previous distance stored in a shift register. If the state was True, the obstacle continues. If it is False, the end of a gap has been identified and the gap is compared with other gaps by calculating a score. If this gap's score is the highest so far, its offset and size are output.

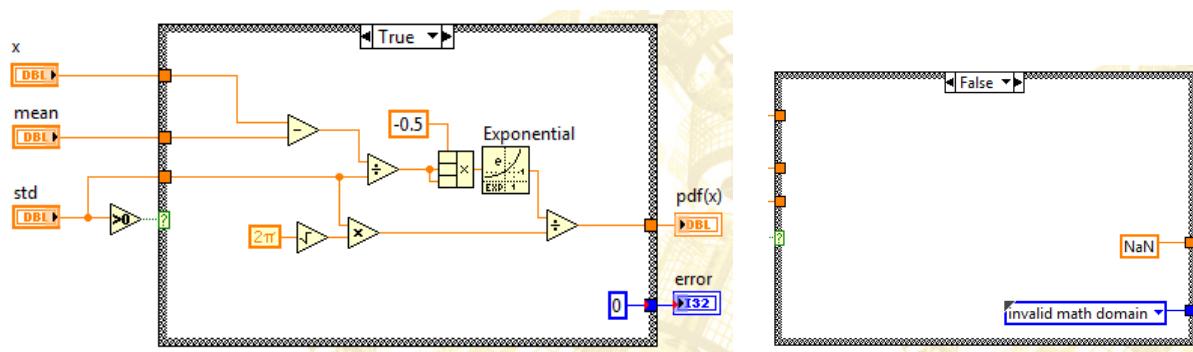
The evaluation is done by calling the Simple VFH Score VI shown in Figure 0-24. The score is compared with that of the best gap so far. The Greater? Comparison output is input to a Case structure selector. If the score of the current gap is better than previous gaps, the True case

executes and the offset, size, and score of the current gap is output. If the comparison is False, the previous values of offset, size, and score of the best gap so far are output.



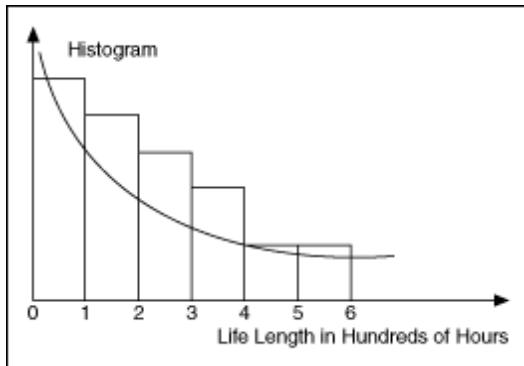
**Figure 0-24. Simple VFH Score VI block diagram**

The Simple VFH Score VI inputs  $x$ , mean, and std to the normal probability density function code shown in Figure 0-25 that calculates the score.  $x$  specifies the quantile of the continuous random variate,  $X$ . Mean specifies the location or mean parameter of the variate. Std specifies the scale or standard deviation parameter of the variate and must be greater than 0.  $\text{pdf}(x)$  is the probability density function at  $x$ .



**Figure 0-25. Normal PDF VI block diagram**

Siegwart et al (2011) describes the probability density function in chapter 4. There is also a brief description and example in LabVIEW Help. Continuous random variables can take on any value in an interval of real numbers. For example, an experiment measures the life expectancy  $x$  of 50 batteries of a certain type. The batteries selected for the experiment come from a larger population of the same type of battery. Figure 0-26 shows the histogram for the observed data.



**Figure 0-26. Probability density function example**

The figure shows that most of the values for  $x$  are between zero and 100 hours. The values drop off smoothly as  $x$  increases. The value of  $x$  can equal any value between zero and the largest observed value, making  $x$  a continuous random variable. You can approximate the histogram with an exponentially decaying curve (note the use of the exponential function in the Figure 0-25 block diagram). If you want to know the probability that a randomly selected battery will last longer than 400 hours, you can approximate the probability value by the area under the curve to the right of the value 4. The function that models the histogram of the random variable is the probability density function.

The gap output code segment also outputs a cluster of the angle to gap and the size of gap. The angle to gap is determined by sending its index to the index array function connected to the direction angles array. The index is the offset +  $\frac{1}{2}$  size. The size of gap output is calculated by multiplying the size value and the difference between two adjacent angle values extracted from the first two array values by the index array function.

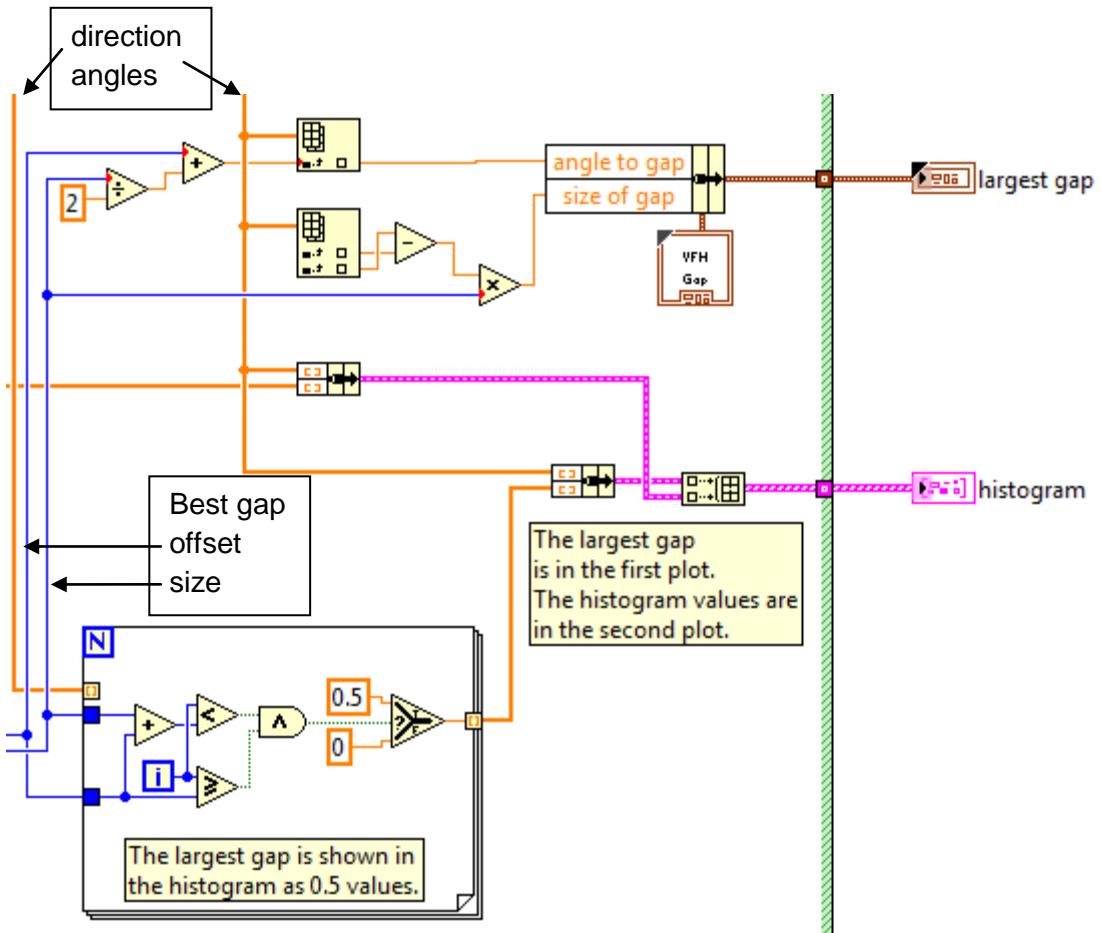
Draw a flow chart of the gap feature extraction code. Use the following symbols to represent different operations:

- Oval - program terminations,
- Diamond – selection (Case structure and select function)
- Parallelogram - input and output
- Rectangle - a process
- Small circle - beginning and end of a subset of the entire VI
- Arrows - flow of control between elements

Flowchart drawing-symbol tools are available in Microsoft Office Word and other programs.

After determining the offset and size of the largest gap, the VFH code outputs the information and creates a histogram in the code shown in Figure 0-27. The angles array is wired to a For loop tunnel to set the number of iterations with auto indexing. The size and the offset of the gap with the best score are input to the loop and summed. If the sum is less than the loop index,  $i$ , which is also the array index, or the offset, the Less? Function outputs True to the logical AND

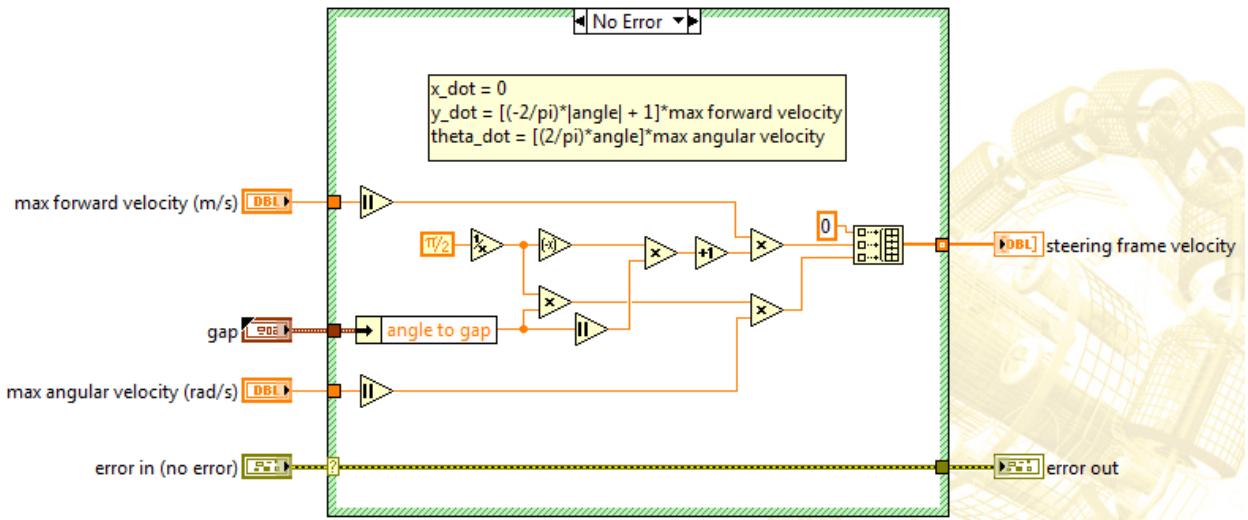
function. If the loop index is also greater than or equal to the offset, 0.5 is written to the output array. Otherwise 0 is written to the output array. This sends an array of 0.5 and 0 values to the histogram along with the array of direction angles. The result is to show the location of the largest gap in the histogram. The histogram also plots the array of 0's and 1's for gaps and obstacles respectively versus the direction angles array.



**Figure 0-27. Gap data output section of the VFH block diagram**

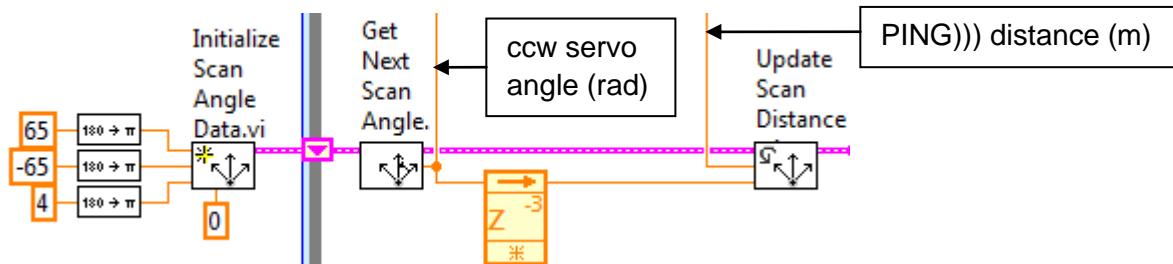
The Simple VFH VI also outputs an angle and a distance to the closest obstacle. The closest obstacle is determined by wiring the distances array to the Array Max & Min function and writing the Min output to the distance to obstacle. The angle is determined by wiring the index of the Min value to an Index Array function connected to the direction angles array.

The output from the Simple VFH is sent to the Drive Towards Gap VI as shown in Figure 0-18. The Drive Towards Gap code is shown in Figure 0-28 where units are converted and the steering frame velocity array is assembled as explained in the previous experiment.



**Figure 0-28. Drive Towards Gap VI block diagram**

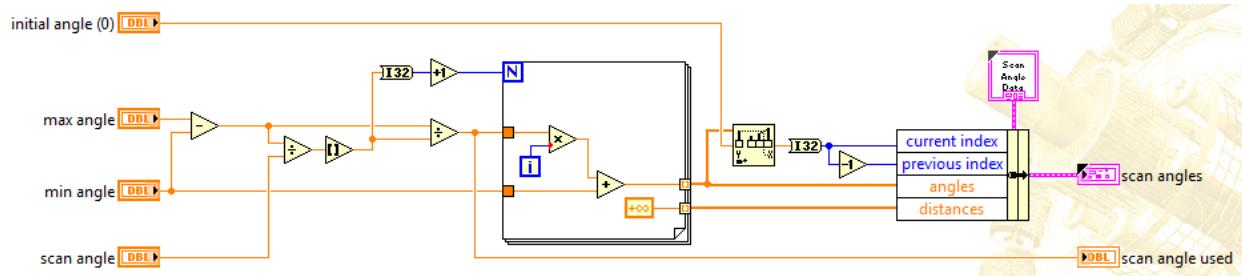
The process of extracting gap features requires input data of two 1D arrays: distances and direction angles arrays that are created in the code segment from the Roaming VI shown in Figure 0-29. The Initialize VI initializes the scanned obstacles data structure, the Get Next Scan Angle VI calculates the next scan angle for PING))), and the Update Scan Distance adds the sensor distance reading to the scanned obstacles data.



**Figure 0-29. Code segment from the Roaming VI that generates the distances and direction angles arrays**

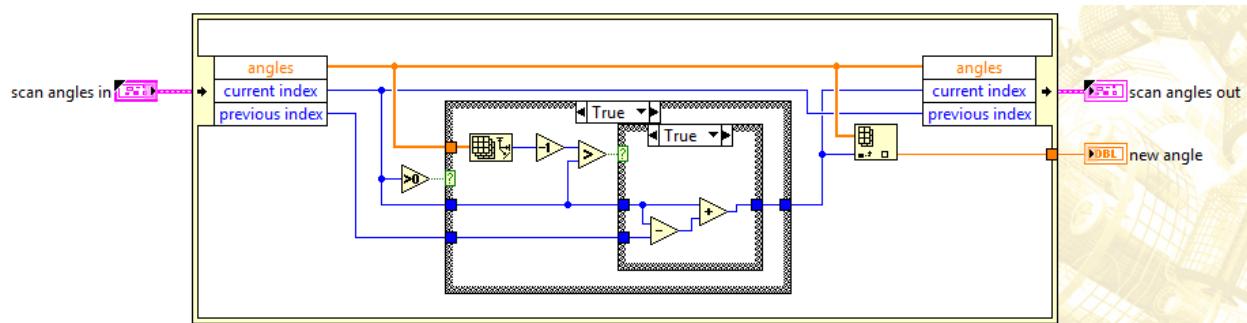
The Initialize VI shown in Figure 0-30 creates a data structure for the angle and distance information by bundling two l32 scalars: current index and previous index, with two arrays: angles and distances into a cluster named scan angles. The values for max angle, min angle, and scan angle: 65, -65, and 4 respectively are set by constants on the Roaming VI block diagram. The number of values in the arrays are calculated by subtracting the min angle from the max angle and dividing by the scan angle, taking the absolute value, converting to a 32-bit integer, and incrementing by 1. The calculated number of values in the arrays is written to N on the For loop to set the number of iterations. Code inside the loop writes initial scan angle and distance values to the arrays. Scan angle values are calculated by multiplying the loop iteration

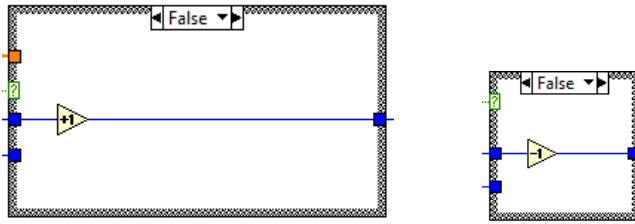
value, i, by the number of angle values + the min angle. The positive infinity constant is written to all of the distance array elements.



**Figure 0-30. Initialize Scan Angle Data VI block diagram**

After initialization in front of the loop, the Get Next Scan Angle VI shown in Figure 0-31 executes iteratively in the loop. It uses the In Place Element structure to control how the LabVIEW compiler performs to increase memory and VI efficiency. Without this structure, LabVIEW creates an additional space in memory, so it has a memory space for the previous values and a second memory space for new values. This can be problematic when working with arrays and clusters with large numbers of elements on hardware like the sbRIO that has limited memory. With this structure you can eliminate the requirement to copy data values and maintain those values in memory, in essence using the same data space in memory by writing over previous values. The structure can perform several operations. Right-click the border and select the node that matches the operation you want to perform, in this instance unbundle the cluster values on the left (input) side and rebundle them on the right (output) side. Inside the structure, the current index value is written to the previous index. The node contains nested Case structures. If the current index is >0, and the size of the angles array is greater than the current index both True cases execute, and the difference between the current and previous indices is calculated and added to the current index. This value is written to current index to update it. This value is also input to an index array function wired to the angles array to output the new angle value. If the current index is not >0, the outer False case and the inner True case executes, and the current index is incremented by 1. If the current index is >0, and the size of the angles array is not greater than the current index, the outer True case and the inner False cases execute, and the previous index is decremented by 1. Note that the values in the angles array do not change.

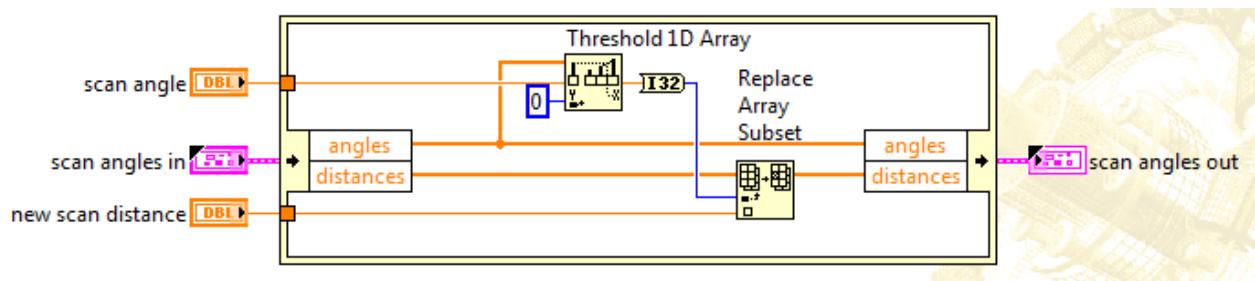




**Figure 0-31. Get Next Scan Angle VI block diagram**

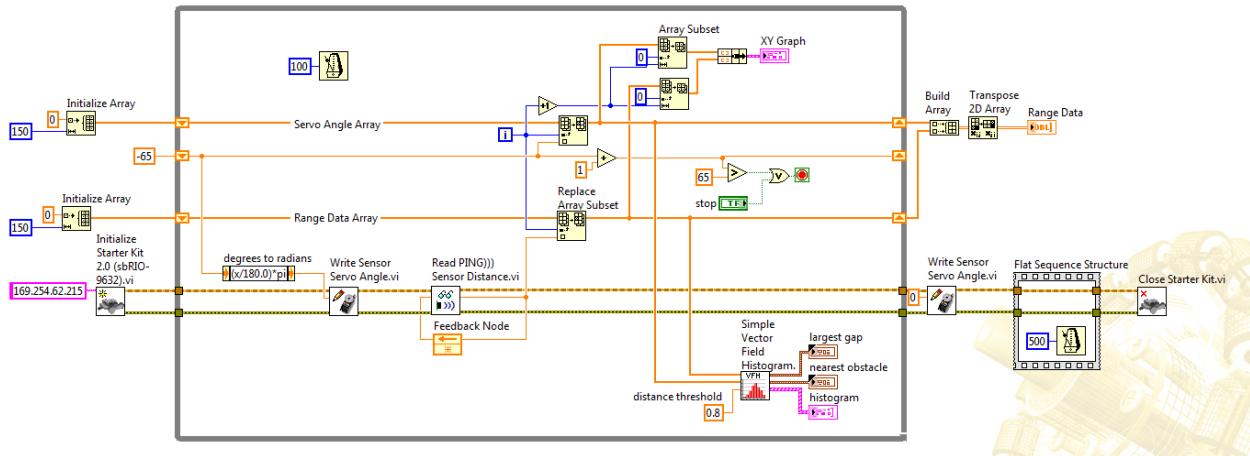
Draw a flow chart for the Get Next Scan Angle VI code.

The modified scan angle data structure output from the Get Next Scan Angle VI is written to the Write Sensor Servo Angle VI which rotates the servo and PING))) to a new position with the code shown in Figure 0-16. After it executes, the next range data is acquired by PING))) and written to the distances array in the scan angles data structure with the code shown in Figure 0-32. The Threshold 1D Array function compares threshold  $y$  to the values in the array until it finds a pair of consecutive elements such that  $y$  is greater than the value of the first element and less than or equal to the value of the second element. LabVIEW calculates a fractional index,  $x$  for the position of  $y$  between the two values. Suppose the array elements are 2.3, 5.2, 7.8, 7.9, 10.0, the start index is 0, and the threshold  $y$  is 6.5. The output is 1.5 because 6.5 is halfway between 5.2 (index 1) and 7.8 (index 2). The output is changed to an integer and is used to set the index at which the new scan distance is written over the previous value in the distances array.

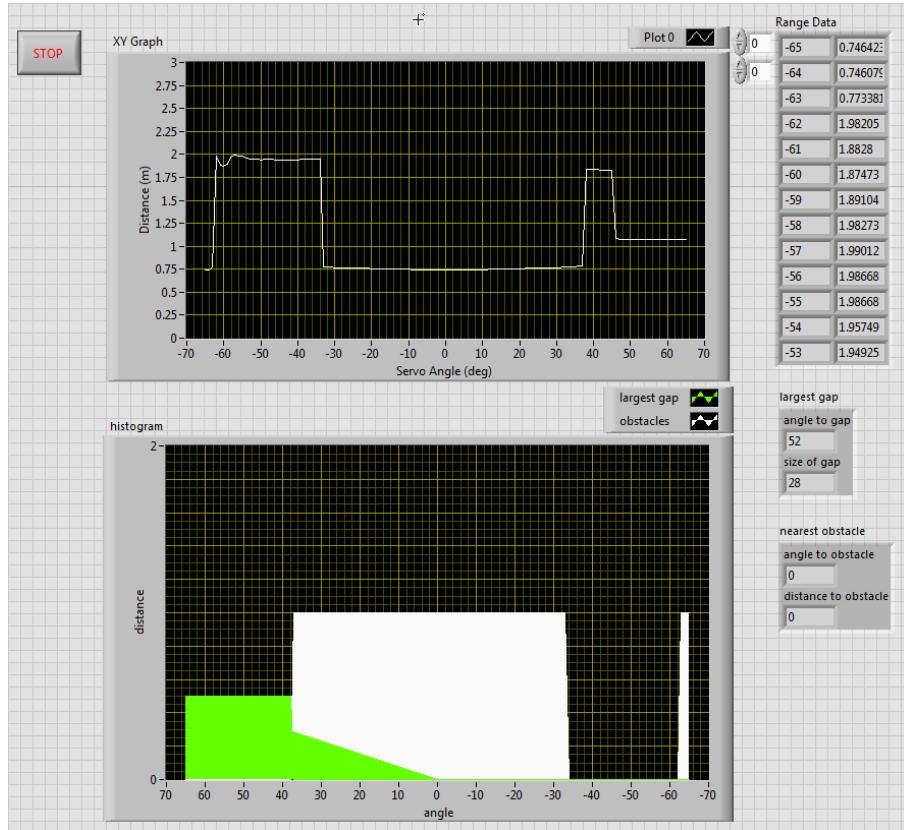


**Figure 0-32. Update Scan Distance VI block diagram**

Add the Simple VFH VI (Functions>>Robotics>>Obstacles palette) to the Acquire and Display Range Data VI developed previously as shown in Figure 0-33. Create an environment similar to that used previously to test the Acquire & Display Range Data VI. Run the VI with DaNI in a stationary position while PING))) scans the environment to obtain results similar to Figure 0-34. Compare the output of the XY graph with the VFH histogram and explain the differences. Change the environment with different obstacles in different locations and distances. Continue the comparison between the histogram and xy graph. What happens if you increase  $\Delta\theta$  from  $1^\circ$  to  $4^\circ$ ?



**Figure 0-33. Acquire & Display Range Data VI with VFH VI block diagram**



**Figure 0-34. Range data comparison with VFH histogram**

Modify the environment so there are at least two gaps. Add functionality to the VI above so it displays a list of gaps and their scores. Manually calculate the score of one of the gaps and compare it with the results from the VI.





# Experiment 6 – Localization

## Instructor's Notes

This experiment requires that the previous experiment be completed. Similar experimental area and tools used in the previous experiment are used here.

## Goal

Determine DaNI's location in the environment with odometric localization (dead reckoning).

Build an occupancy grid map.

## Required Components

Linear distance measuring tool like a ruler, meter stick, or tape measure.

Angle measuring tool like a protractor.

## Background

Students should study Siegwart et al (2011) chapter 5. This experiment requires that the previous experiments have been completed.

## Experiment 6-1 Odometric Localization (Dead Reckoning)

Figure 0-1 introduced local and global reference frames. Previous experiments measured distances to obstacles and extracted line features relative to DaNI in the local reference frame. The roaming VI also used relative positions. However, sometimes you need to know determine global positions of DaNI and obstacles (occupied space) and open space.

A previous experiment characterized the wheel encoders and explained the FPGA code that uses their feedback data in a PID algorithm to control wheel velocity. You determined the wheel

circumference and evaluated the errors in orientation and distance in driving from point A to point B. In this experiment, you will develop code to send DaNI in a path that returns to the starting point using the encoder data. Siegwart et al (2011) calls this practice odometric position estimation, other roboticists term it dead reckoning.

Create a VI that will drive a closed path from point A to B, C, D and back to A in a 1 m square path. Record the error in orientation and distance at each point. Evaluate the propagation of the error.

Repeat 3 times, stopping after each circuit so the position resets to (0,0) before the next circuit and report the maximum and average errors.

Repeat 3 times without stopping and don't reset to (0,0) after each circuit and report the propagation of errors.

Siegwart et al (2011) explains that the errors are from three sources:

1. Distance (sum of wheel movements),
2. Turn error (difference of the wheel movements when turning), and
3. Drift error (difference in wheel error causes angular orientation error =  $d \sin \Delta\theta$  which is the largest of the three errors). The text presents the following analysis. Use the data from the above experiment to evaluate the variables in the following.

$$\Delta\theta = \frac{\Delta s_r - \Delta s_l}{b}, \quad (5.4)$$

$$\Delta s = \frac{\Delta s_r + \Delta s_l}{2}, \quad (5.5)$$

where

$(\Delta x; \Delta y; \Delta\theta)$  = path traveled in the last sampling interval;

$\Delta s_r; \Delta s_l$  = traveled distances for the right and left wheel respectively;

$b$  = distance between the two wheels of differential-drive robot.

Thus we get the updated position  $p'$ :

$$p' = \begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = p + \begin{bmatrix} \Delta s \cos(\theta + \Delta\theta/2) \\ \Delta s \sin(\theta + \Delta\theta/2) \\ \Delta\theta \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} \Delta s \cos(\theta + \Delta\theta/2) \\ \Delta s \sin(\theta + \Delta\theta/2) \\ \Delta\theta \end{bmatrix}. \quad (5.6)$$

$$p' = f(x, y, \theta, \Delta s_r, \Delta s_l) = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} \frac{\Delta s_r + \Delta s_l}{2} \cos\left(\theta + \frac{\Delta s_r - \Delta s_l}{2b}\right) \\ \frac{\Delta s_r + \Delta s_l}{2} \sin\left(\theta + \frac{\Delta s_r - \Delta s_l}{2b}\right) \\ \frac{\Delta s_r - \Delta s_l}{b} \end{bmatrix}.$$

## Experiment 6-2 Localize with Range Data

If the robot has an environmental map stored in its computer, it can use it to keep position uncertainty from growing unbounded when using odometric localization. The robot can localize itself in relation to the map.

Create a VI that includes reading a file containing map data. The map data might be distances to sides of boxes placed strategically so DaNI can use the range data (distances and angles) to calculate its position at the start point of the circuit in the previous experiment. How many range data measurements are necessary to determine position? In essence DaNI is going to use the range data to the obstacles to measure its error automatically after creating the circuit, report its

new coordinates, and calculate the error from (0,0). Test the VI by driving the circuit, automatically calculating the error and comparing the result to the actual error. How does the error in ultrasonic range data affect the result?

What change would you have to make in the above map, obstacle environment, and VI to report orientation in addition to position after completing the circuit?

## Experiment 6-3 Occupancy grid map

Create a VI that will build an occupancy grid map of an area. The area should be designed to contain some obstacles and to have some free space. The area could be approximately 2 m x 2 m but a larger area or a rectangular area with one direction over 3 m would be better.

Represent the occupancy map as a set of 0s and 1s in a file as a matrix or table. Graph the occupancy grid results in an XY graph on the front panel. Choose a large cell size like 0.25 m so if the area is 2 m x 2 m, the map will have 64 cells in an 8 x 8 matrix. Write 0s in all of the cells in the map file to begin. Place DaNI within the area and take a series of range data sets with DaNI positioned over the same point but rotating 45 deg after each data set until DaNI has rotated 315 deg cw to cover the entire area around the robot. This will produce 7 data sets. Assume the point where DaNI is located is at global coordinate (0,0). Determine the global x and y coordinates of all obstacles. Determine which cells contain the obstacle coordinates. Write a 1 to cells that are occupied. Leave 0 values in any cells that can't be seen because obstacles block the ultrasound meaning 0 represents both free and unmapped cells.

# **Optional Projects and Competitions**

## **Obstacle avoidance, Localization and Mapping**

Set up a space with four walls and two stationary obstacles. Create a VI that will roam around the space and report the position of the walls and obstacles in an occupancy grid map where the grids are 0.5 m square. The occupancy grid result will be written to a file on the sbRIO during/after roaming. After roaming, connect to a host via Ethernet cable, communicate the file to the host, and display the map in an XY graph on the host. You get three tries and you are allowed to improve your VI after each try. Make it a competition by reporting time to complete the map and the level of accuracy of the map.

## **Obstacle avoidance, Localization, Mapping, and Object Recognition**

Set up a space with four walls and two stationary obstacles. The obstacles have significantly different geometries for example a chair and a box, or a chair and a doll. Identify one of the two objects as a target. Create a VI that will roam around the space and report the position of the target obstacle. Write the position to a file on the sbRIO during/after roaming. After roaming, connect to a host via Ethernet cable, communicate the file to the host, and display the obstacle position in an XY graph on the host. Make it a competition by reporting time to complete the map and the level of accuracy of the map. You get three tries and you are allowed to improve your VI after each try. Make it a competition by reporting time to locate the target and the accuracy of the location.

## **Obstacle Avoidance, Mapping, and Navigation**

Set up a maze that has one entrance and one exit with wooden or cardboard walls. Include at least one dead end passage, one cw turn, and one ccw turn. Create a VI that will navigate the maze with purely reactionary control. Report the time to navigate the maze. You get three tries and you are allowed to improve your VI after each try. Make it a competition by reporting time to complete the maze and the total distance travelled.

Increase the difficulty by creating a map of the maze that is communicated to and displayed to a host computer at the end of navigation.

## **Hardware Enhancement**

Each student or team is given a budget for additional sensor(s) purchase to improve performance in one of the above projects. Students should evaluate alternative sensors to decide which will improve performance and can be implemented on DaNI. Sensors could be from sparkfun, from one of the FRC vendors, from another source, or from inventory at the institution.