



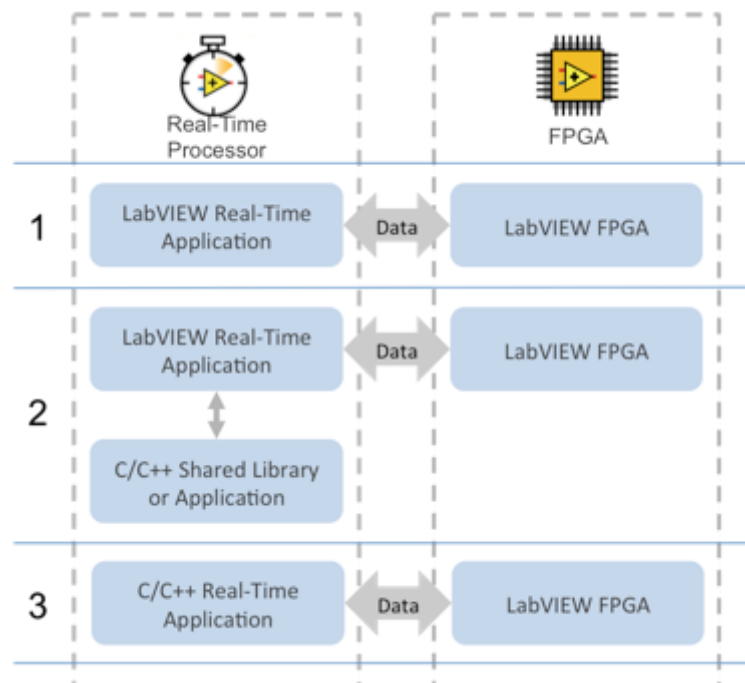
# NI LabVIEW RIO Evaluation Kit

## C++ Tutorial

# NI LabVIEW RIO Evaluation Kit Tutorial

This document contains step-by-step instructions for experiencing embedded system design using LabVIEW system design software and C++ with NI reconfigurable I/O (RIO) hardware architecture to create an embedded control and monitoring system.

It is possible to program NI hardware using LabVIEW, C/C++, or a combination of both, as shown below. This tutorial covers programming the real-time application entirely in C++ (scenario 3 below). There is a second tutorial, *NI LabVIEW RIO Evaluation Kit Tutorial.pdf*, installed by the DVD included with this kit, that walks through programming the application entirely in LabVIEW and optionally calling a C shared library from your LabVIEW Real-Time code (scenarios 1 and 2), which can be found on disk in the same folder as this LabVIEW tutorial.



The ideal next step prototyping platform is [NI CompactRIO](http://ni.com/compactrio), built with the same architecture as this kit but with modular I/O. The programming experience and APIs learned in this evaluation experience are the same, in fact, you can even reuse the code you've built! This LabVIEW RIO architecture covers a wide range of hardware products; visit [ni.com/embedded-systems](http://ni.com/embedded-systems) for more information.

**Note:** It may be easier to complete the step-by-step exercises if you **print** this document before proceeding. Otherwise, the Table of Contents contains links to the exercises for more convenient navigation of this document.

# Table of Contents

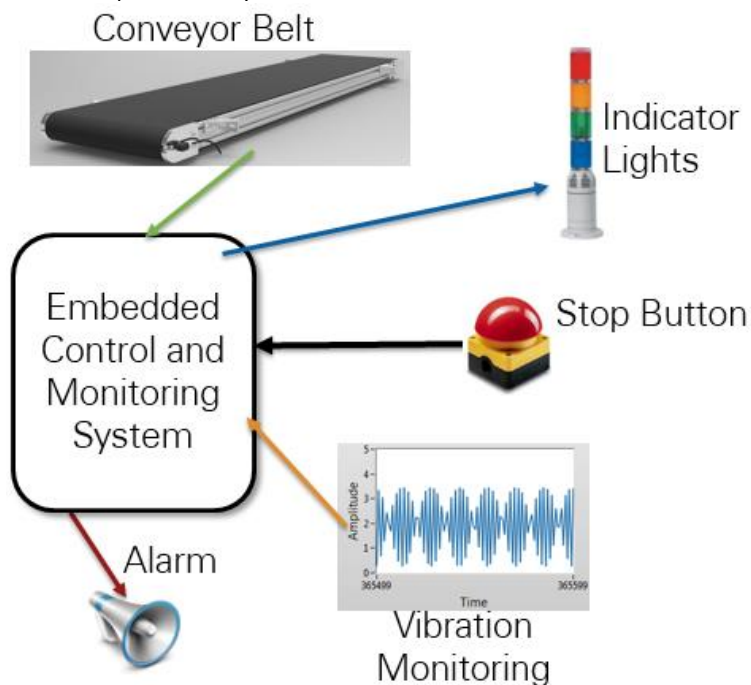
Tutorial Overview .....	4
Navigating Exercises .....	9
Using the Solutions .....	9
Troubleshooting .....	10
Activating LabVIEW .....	11
Getting Started – LabVIEW Programming Basics .....	11
Initial System Configuration .....	13
Exercise 1 – Open and Run Application .....	26
Exercise 2 – Create a Monitoring and Control FPGA Application .....	31
Test the FPGA Application .....	41
Exercise 3 – Develop Real-Time Application .....	42
Test the Real-Time Application .....	50
Exercise 4 – Run with a Windows User Interface .....	52
Run and Verify the Completed System .....	54
Exercise 5 – Startup Application .....	55
Appendix A – LabVIEW RIO Training Path .....	56
Appendix B - Changing the IP Address in the LabVIEW Project .....	60

## Tutorial Overview

In this tutorial, you will complete five exercises that demonstrate how to develop an embedded system using LabVIEW system design software and C++ with NI reconfigurable I/O (RIO) hardware which includes a real-time processor, FPGA, and I/O. Using the LabVIEW RIO Evaluation Kit, your challenge will be to prototype the embedded control and monitoring system for a part inspection machine, similar to what is shown below. This system carries the product through the inspection process and controls a variety of inspection methods



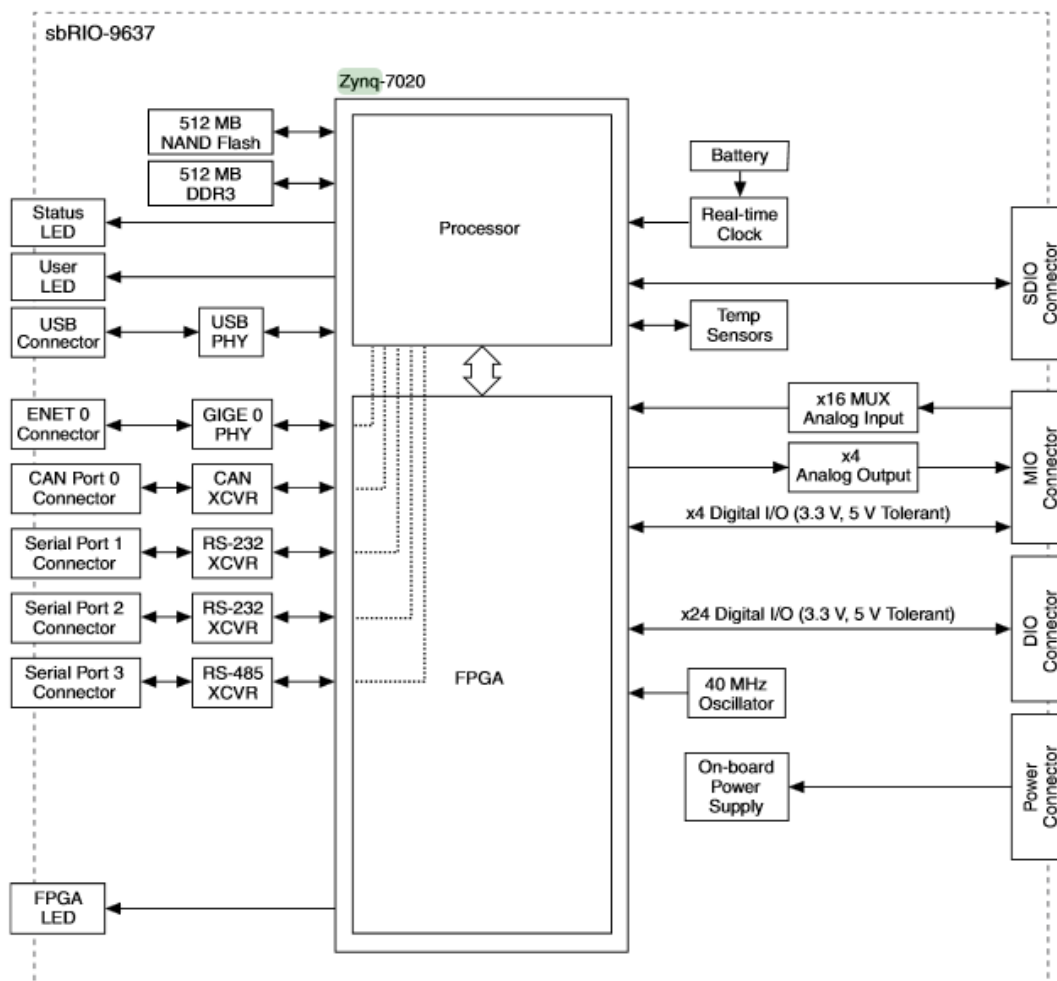
Here are the various components you will interact with in this scenario:



## System Specifications

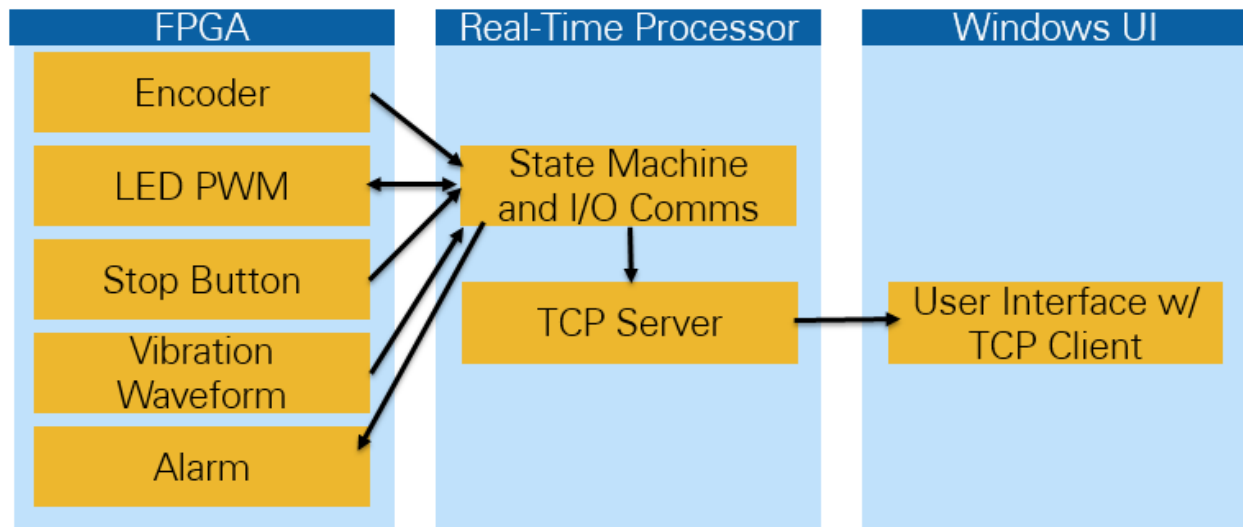
Your kit is based on a standard hardware architecture that includes a [Xilinx Zynq-7000 XC7Z020 All Programmable System on Chip](#), with an Artix-7 FPGA and a 667 MHz dual-core ARM Cortex-A9 processor running the [NI Linux Real-Time](#) (32 bit) distribution, and I/O available through the daughter card. The system you are developing in this tutorial will include both the embedded target and your Windows PC. It can be programmed using a single development tool chain, LabVIEW, or you can program the processor with C/C++; this kit includes the necessary cross-compiler and the Eclipse IDE. Since LabVIEW includes a cross-compiler, it can be used to develop applications that will run on a floating point processor, an FPGA target, and a Windows PC.

## Block Diagram



## System Diagram

Since there is some flexibility with three available targets to run code, the part inspection tasks have been mapped to processes on each of the targets (Windows User Interface, Real-Time processor, and FPGA). An FPGA (Field Programmable Gate Array) is basically software-defined hardware that can be reconfigured multiple times to create a custom circuit. All the system I/O goes through the FPGA and you can achieve very fast and reliable control and acquisition.



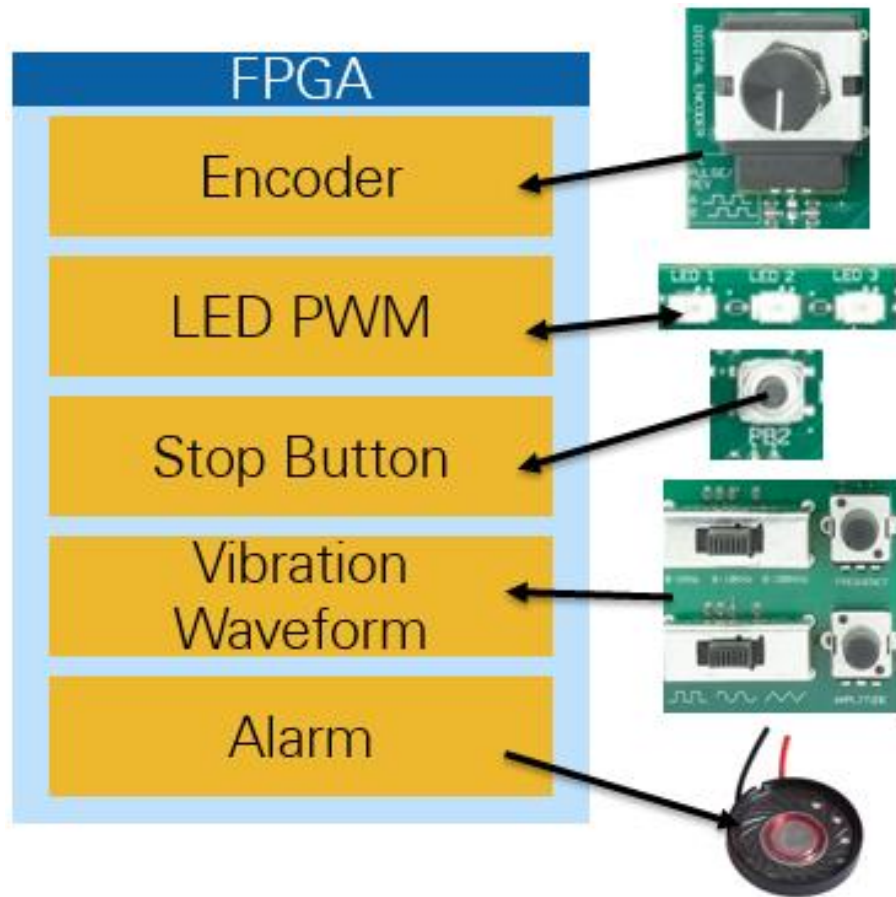
### Monitoring Tasks

1. Vibration monitoring

### Control Tasks

1. Encoder
2. Stop Button
3. Alarm
4. Indicator LEDs

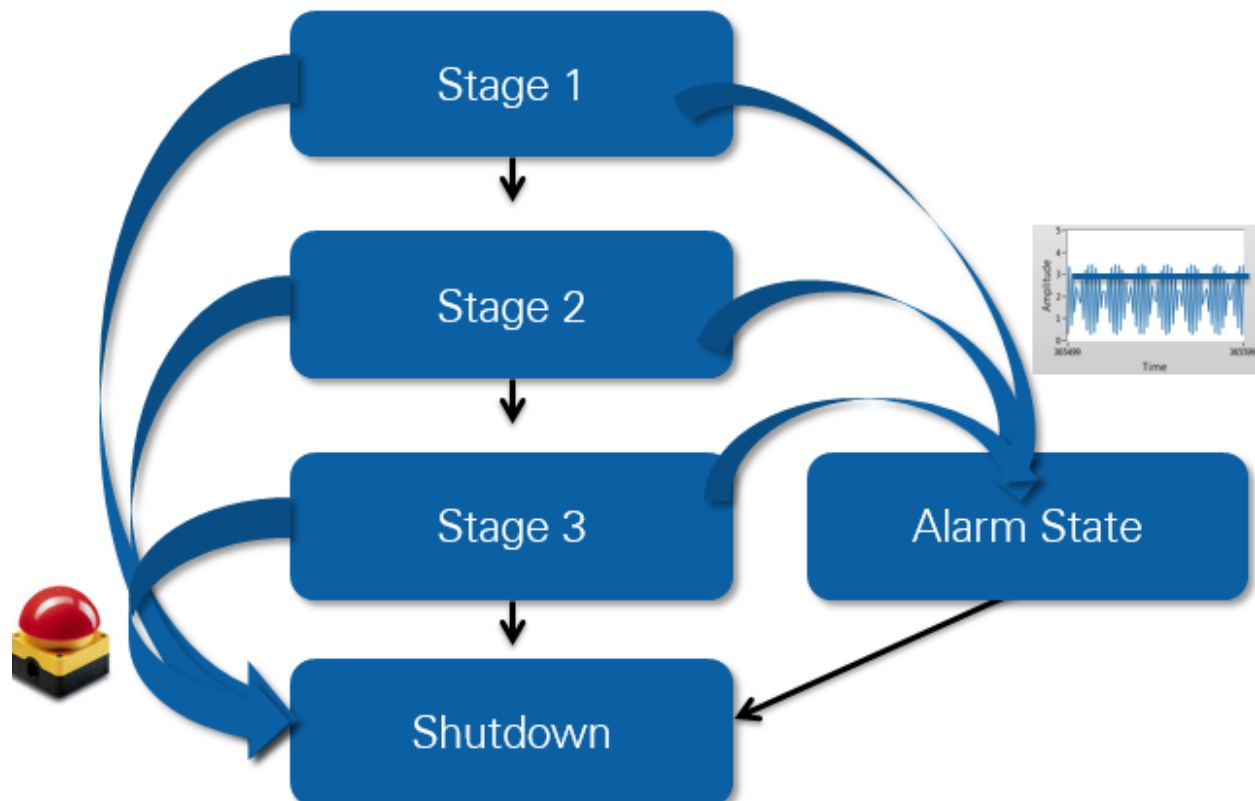
Finally, since you do not have an actual part inspection system to control and monitor here is how you will simulate various elements using the onboard I/O, connected through the FPGA of your evaluation device:



## State Diagram

Normal machine operation will progress from stage 1 » 2 » 3 then proper shutdown. If an alarm condition is tripped, when the machine's vibration waveform amplitude crosses a threshold, you will enter the alarm state and automatically progress to proper shutdown. If you press the Stop button on your board (PB2) you will also automatically progress to proper shutdown.

For example, when the amplitude of the machine's vibration waveform exceeds a preset threshold, it could mean a damaged bearing that a technician should investigate.





To build up the control and monitoring system outlined, you will design the components of the system through each of the tutorial exercises:

### **Exercise 1: Open and Run Application**

Explore and run a precompiled application that simulates precise intensity control of two lasers, pulsed at a high frequency. The FPGA implements a pair of high frequency and reliable pulse width modulation (PWM) channels. The RT controller hosts a simple user interface (UI) to control the lasers.

### **Exercise 2: Create a Monitoring and Control FPGA Application**

Create an FPGA application on your own to control and interact with your I/O for the part inspection machine. While this application is compiling, start on Exercise 3.

**Note:** For this exercise you will either need internet access to compile your LabVIEW FPGA code with the [NI LabVIEW FPGA Compile Cloud Service](#) (required if using Windows 8) or you will need to install the LabVIEW 2015 FPGA Module Xilinx Vivado 2014.4 Compile Tools to your computer from the second DVD in your kit accessories box or from [ni.com](#).

### **Exercise 3: Develop Real-Time Application**

Design a real-time application running on the processor which communicates with the FPGA to exchange data. This code is based on a State Machine architecture.

### **Exercise 4: Create a Windows User Interface**

Extend the embedded system to include a user interface running on a Windows computer communicating over TCP/IP to display the current status of your part inspection system.

### **Exercise 5: Create a Standalone Startup Application**

Now that your system is complete, learn how to deploy the system to run standalone and startup automatically at system reboot.

After completing these exercises, ask questions and explore a variety of LabVIEW Real-Time and LabVIEW FPGA examples and getting started resources built for the kit on an online community for LabVIEW RIO Evaluation Kit users at [ni.com/rioeval/nextstep](#).

For additional details on the NI Linux Real-Time distribution, and tutorials for interacting with third party packages and C/C++ code, visit [ni.com/linuxrtforum](#)

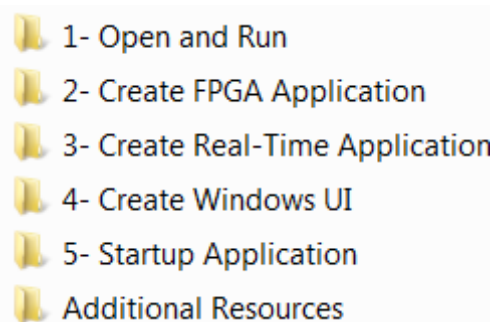
## Navigating the Exercises

The LabVIEW RIO Evaluation Kit DVD installs the exercises on your development machine. By default, these exercises install on your computer's hard drive at:

**C:\Users\Public\Documents\National Instruments\LabVIEW RIO Evaluation Kit\Tutorials**

Alternatively, you may locate these files from the Start menu shortcut at **All Programs»National Instruments»LabVIEW RIO Evaluation Kit»LabVIEW RIO Evaluation Kit Tutorials**. The original files are also located on the DVD included in the evaluation kit.

This is the directory structure for each of the exercises, and you will find additional resources linking to ni.com.



## Using the Solutions

If for any reason you are not able to complete an exercise successfully feel free to open the solution and/or use it to continue on to the next exercise. You will need to modify the solution LabVIEW project and Eclipse Remote System Explorer with your device's IP address if using Ethernet. Reference **Appendix B** for more details on this process.

## Troubleshooting

If you have any questions or run into any configuration issues while exploring this evaluation kit, please review the LabVIEW RIO Evaluation Kit Frequently Asked Questions document installed at **All Programs»National Instruments»LabVIEW RIO Evaluation Kit** or online at [ni.com/rioeval/faq](http://ni.com/rioeval/faq). This document contains information on how to change your IP address or reconnect to your device, and includes answers to installation questions. If that document doesn't cover your question, please post it to the LabVIEW RIO Evaluation Kit user community ([ni.com/rioeval/nextstep](http://ni.com/rioeval/nextstep)) or contact NI Support ([ni.com/ask](http://ni.com/ask)).

## Activating LabVIEW

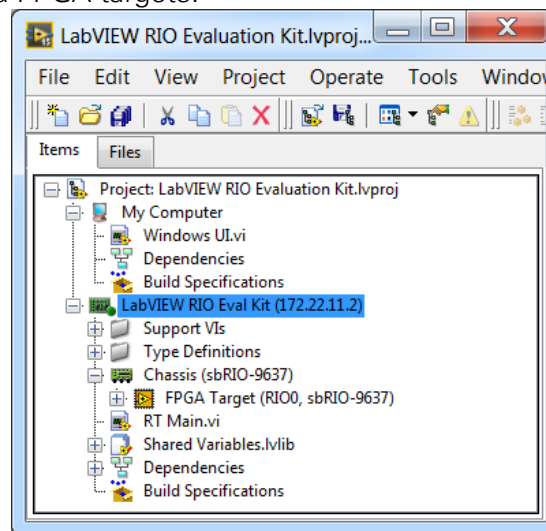
The evaluation kit installs a 90-day full evaluation version of LabVIEW; you will be prompted to activate LabVIEW when you open the environment. To continue in evaluation mode, click the “Launch LabVIEW” button.

## Getting Started – LabVIEW Programming Basics

If you are new to LabVIEW, this section will help you learn more about the LabVIEW development environment and graphical programming language. This tutorial assumes you ran the LabVIEW RIO Evaluation Setup utility upon reboot. If you have not done so, run it now. You can access the utility from your Windows Start menu, select **All Programs»National Instruments»LabVIEW RIO Evaluation Kit»LabVIEW RIO Evaluation Kit Setup Utility**.

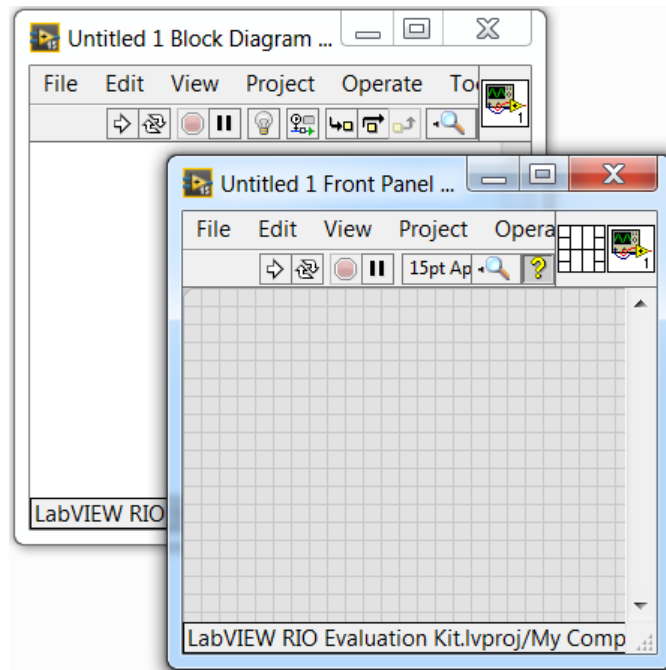
A LabVIEW application is called a “VI”, or virtual instrument, and is composed of two primary elements: a front panel and a block diagram, which you can program using the LabVIEW Functions Palette.

- **Project Explorer** – The Project Explorer window provides a system-level view of the files and VIs in your application and allows you to manage code and build specifications for desktop, real-time, and FPGA targets.



- **Front panel** – The front panel is what you use to create a LabVIEW user interface (UI). For embedded applications, such as FPGA applications, you either create subfunctions, or subVIs, where controls and indicators are used to pass data within the target application or you use the front panel to define sockets/registers that are exposed to other elements of your system (such as the real-time processor) with read/write access.

**Note:** If you close the front panel, it will also close the block diagram, so be sure to minimize it instead if you wish to use the block diagram.



- **Controls Palette** – The Controls palette contains user interface components for creating front panels. To create your user interface, drag the components onto the front panel and wire them on the block diagram to the data you want to display or control.
- **Block diagram** – The block diagram is where you program LabVIEW applications using a combination of graphical and textual notations. To program the block diagram, right-click anywhere on the diagram (blank white window) to bring up the Functions palette. Objects on the front panel window appear as terminals on the block diagram. Terminals are entry and exit ports that exchange information between the front panel and block diagram. Terminals are analogous to parameters and constants in text-based programming languages.
- **Functions palette** – The Functions palette contains primitive blocks and functions for creating FPGA, RT, and Windows applications. Drag the components onto the block diagram and wire them together by left-clicking on a terminal and dragging the wire to your destination, completing this wire segment with another left-click to create a dataflow program.

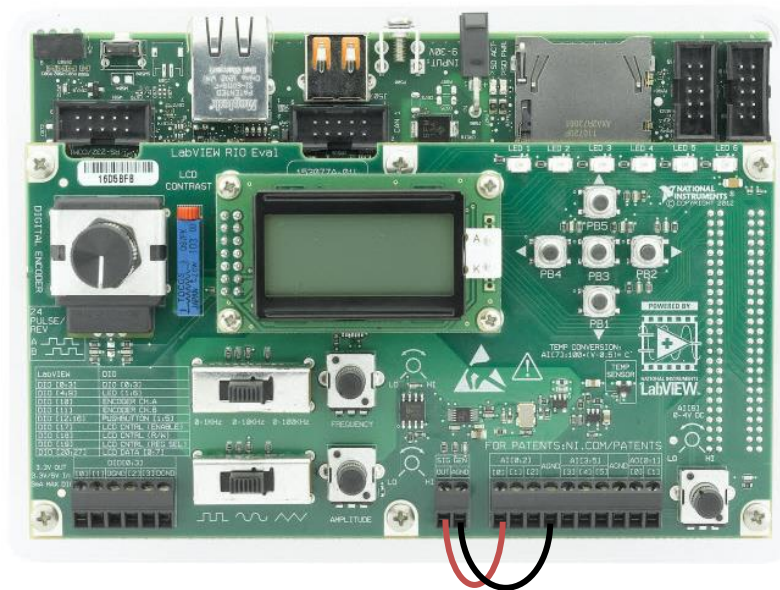
**Using the functions palette** – In this tutorial, **bold** text denotes an item to select from the Functions palette. To access the Function palette, right-click anywhere on the LabVIEW block diagram. You can also “pin” the functions palette (in the upper left corner of its window) so that it is always present on the block diagram.

**More on Using LabVIEW** - To learn more about the LabVIEW graphical programming environment including syntax, deployment, debugging, and more, we *strongly recommend* you reference <http://www.ni.com/gettingstarted/labviewbasics/>.

# Initial System Configuration

## LabVIEW RIO Evaluation Hardware

1. If you haven't already, complete the **Setup Wizard** located at **Start»All Programs»National Instruments»LabVIEW RIO Evaluation Kit»LabVIEW RIO Evaluation Kit Setup Utility**
2. Use the included NI screw driver and wire to connect the Signal Generator OUT and AGND terminals to the AI0 and AGND (ground) terminals as shown below.



3. In preparation for application development verify that the function generator is set to **0-10KHz** (center) and **Sine Wave** generation (center) settings.



4. Connect the Speaker, red wire to AO0 and black wire to AGND as shown below.

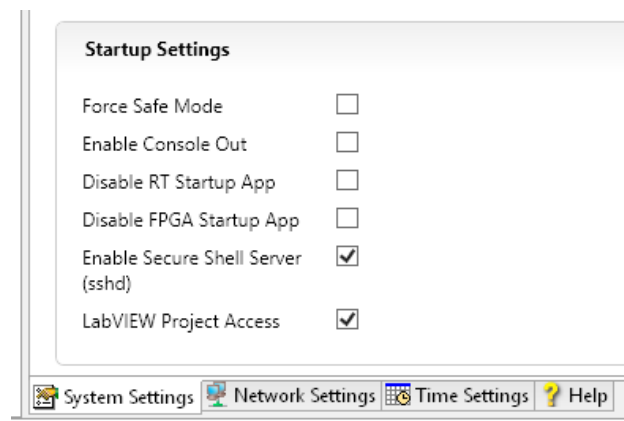


## Eclipse Project Setup

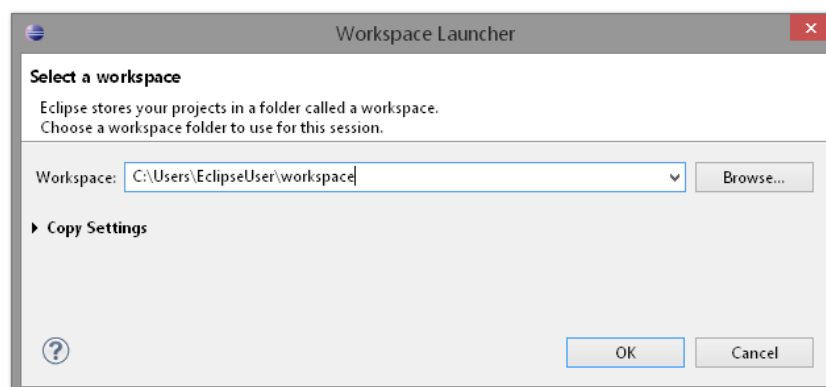
1. Connect the included USB or Ethernet cable directly from your LabVIEW RIO Evaluation Kit to the appropriate port on your computer or to your network router (Ethernet).
2. Open **NI MAX (Measurement and Automation Explorer)** and navigate to the LabVIEW RIO Evaluation Kit under the **Remote Systems** section.



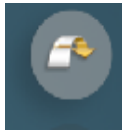
3. Select the System Settings tab on the right and check the box next to **Enable Secure Shell Server (sshd)** in **Systems Settings** if it is not already connected.



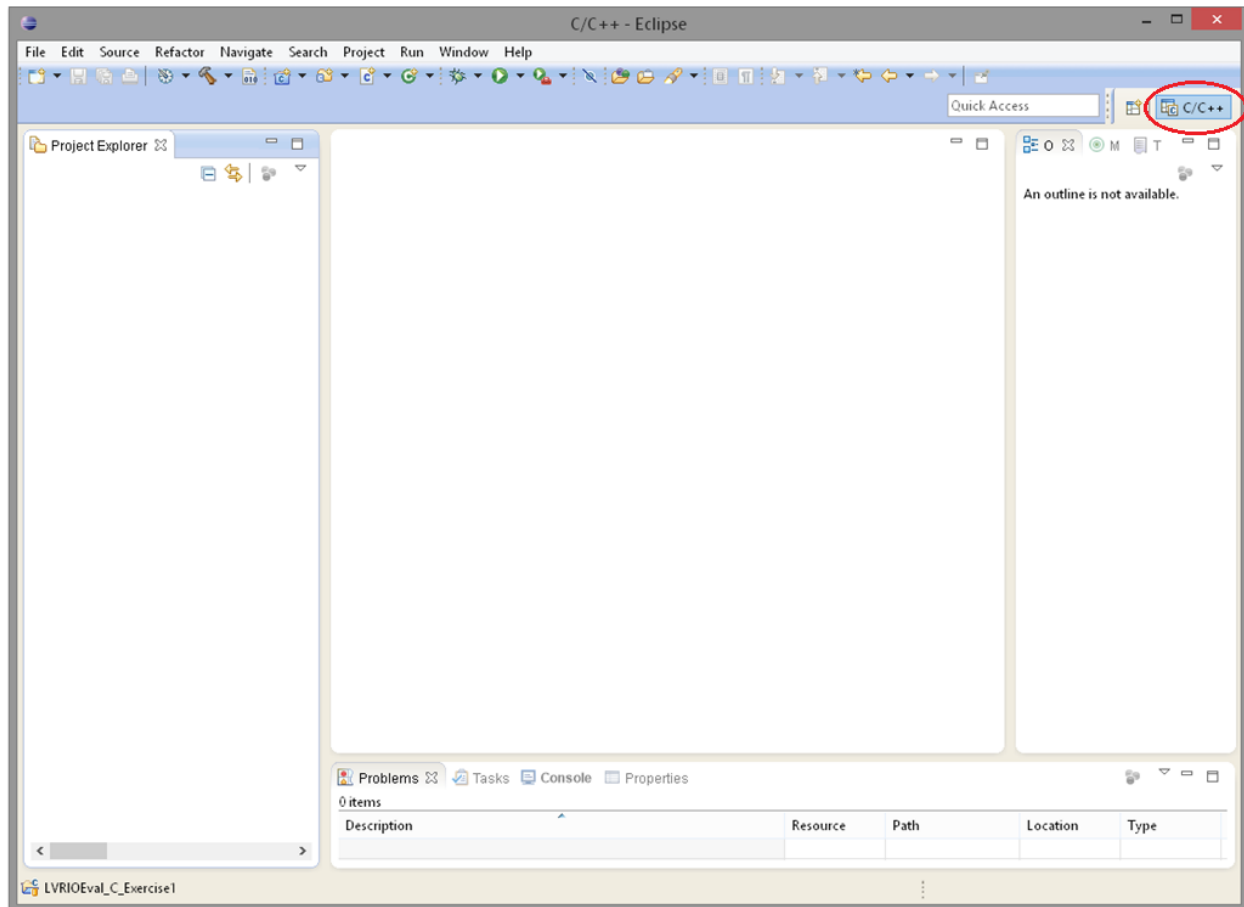
4. Click **Save** at the top to apply these changes to the target.
5. Launch **C & C++ Development Tools for NI Linux Real-Time, Eclipse Edition 2014**
6. When prompted, select a folder in which to store Eclipse projects and click OK. You can create a new folder on your desktop.



7. In the Eclipse welcome screen, select the Workbench icon on the far right to open the workbench view.

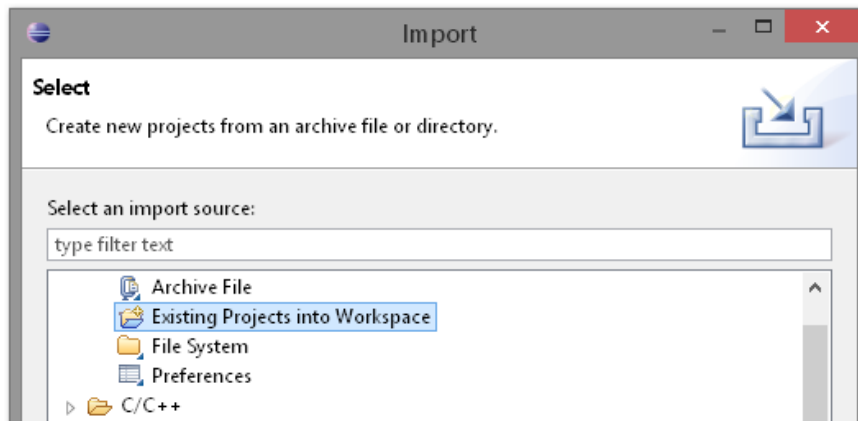


You are now in the C/C++ perspective.

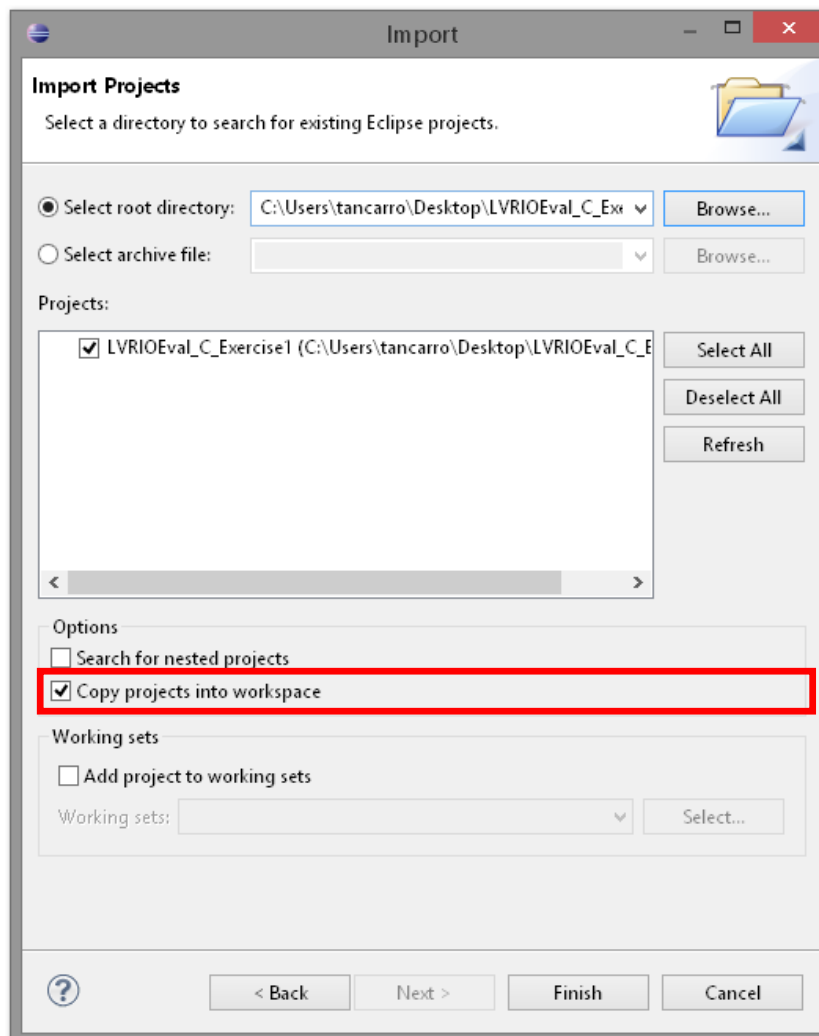




8. To import the Exercise 1 project, go to **File » Import** and select **General » Existing Projects into Workspace** and hit Next.



9. Browse to the "LVRIOEval\_C\_Exercise1" folder from the attached zip file and click OK.
10. Ensure "Copy Projects into workspace" is enabled.



11. Click Finish.

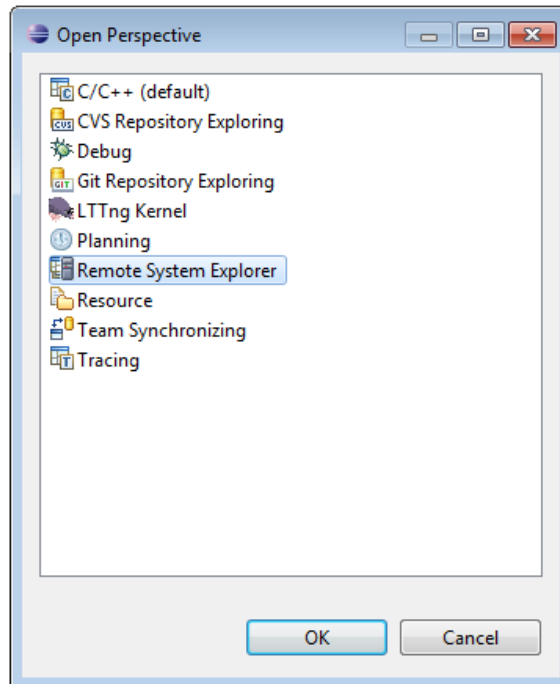


12. Repeat Steps 7 to 10 for "LVRIOEval\_C\_Exercise3" project folder.

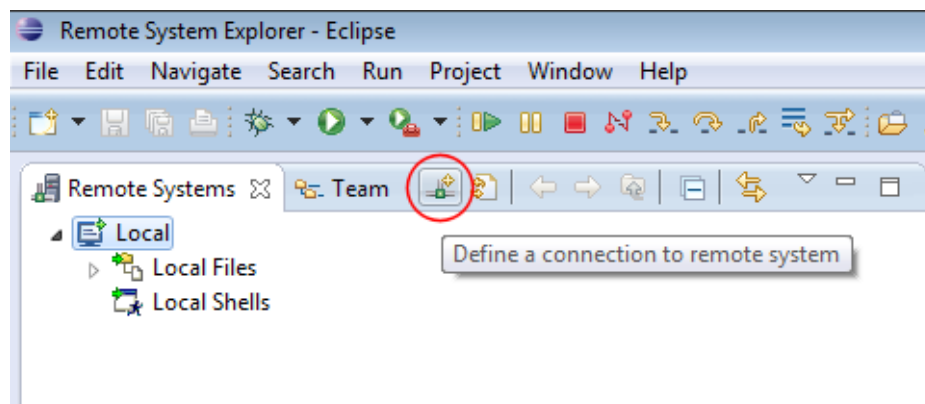
**Note:** Exercise 2 does not use Eclipse so there is no project to import.

### Connect to the LabVIEW RIO Evaluation Kit

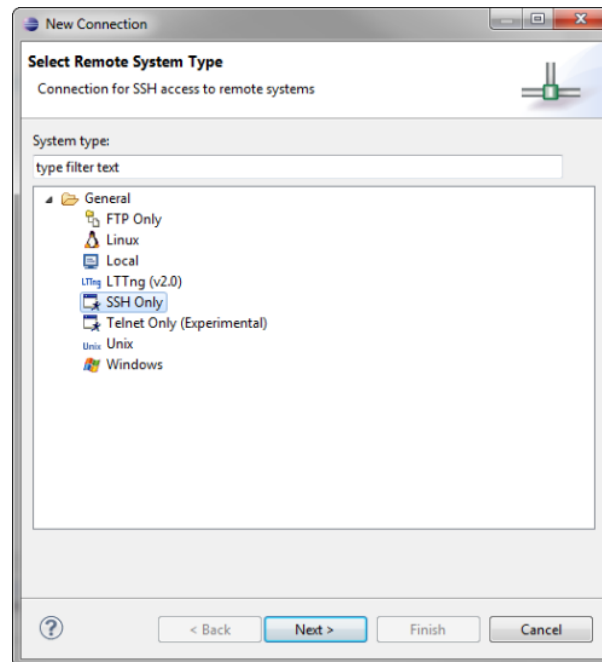
1. Select **Window»Open Perspective»Other** to open the Open Perspective dialog box.
2. Select **Remote System Explorer**.



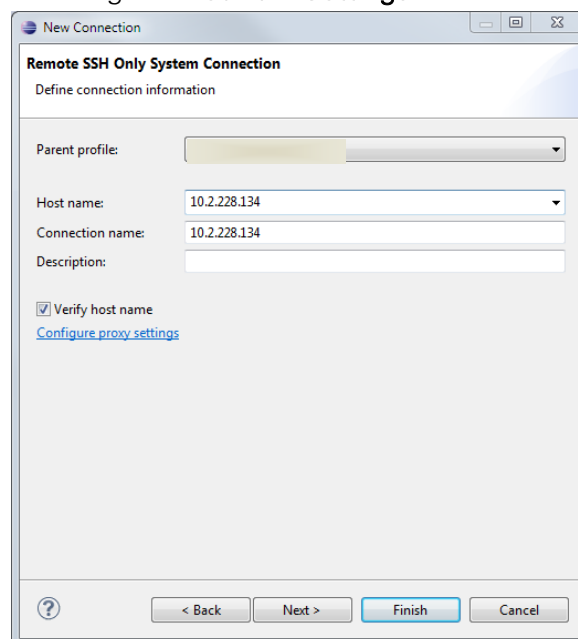
3. Click **OK** to add the Remote System Explorer perspective to the workbench.
4. Click the **Define a connection** to remote system button, circled in the following image, to open the New Connection wizard.



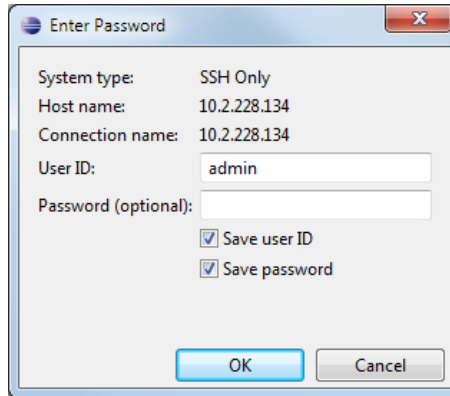
5. Select **SSH Only** under the **General** folder.



6. Click **Next** to open the **Remote SSH Only System Connection** page.
7. Enter the IP address of your NI Linux Real-Time target in the **Host name** text box. If you are connected over USB it is 172.22.11.2 by default. You can use the *LabVIEW RIO Eval Kit Setup Utility* or *NI MAX* to identify your target's IP address under the **Remote Systems** section and selecting the **Network Settings** tab if connected over Ethernet.

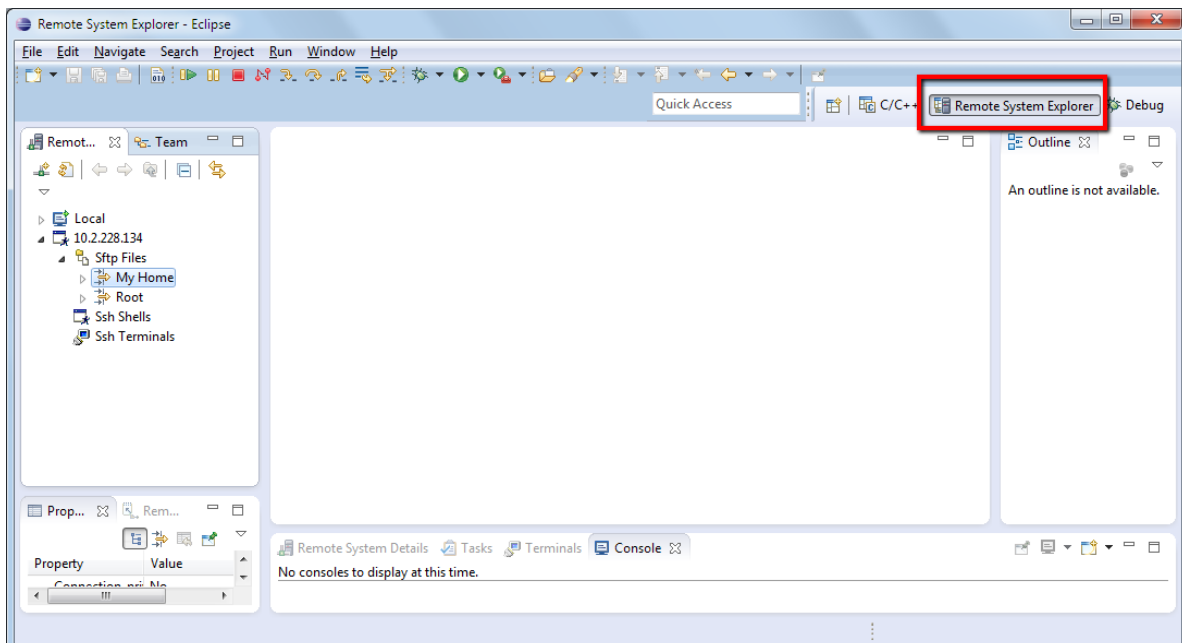


8. Click **Finish**.
9. Make sure you are connected to the target, right-click the target and select **Connect**
10. When prompted, enter the user name and password assigned to your target and click **OK**. The default user name for the LabVIEW RIO Evaluation Kit hardware is "admin" and the default password is a *blank* password, as shown below.



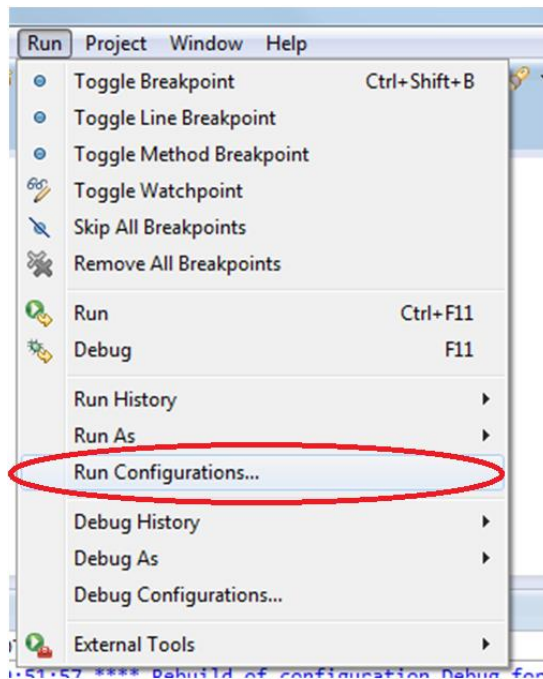
**Note:** The default User ID may show as your Windows login name. Make sure to change this to "admin".

11. You can now toggle between the C/C++ and Remote System Explorer using the Perspective buttons in the top right corner.



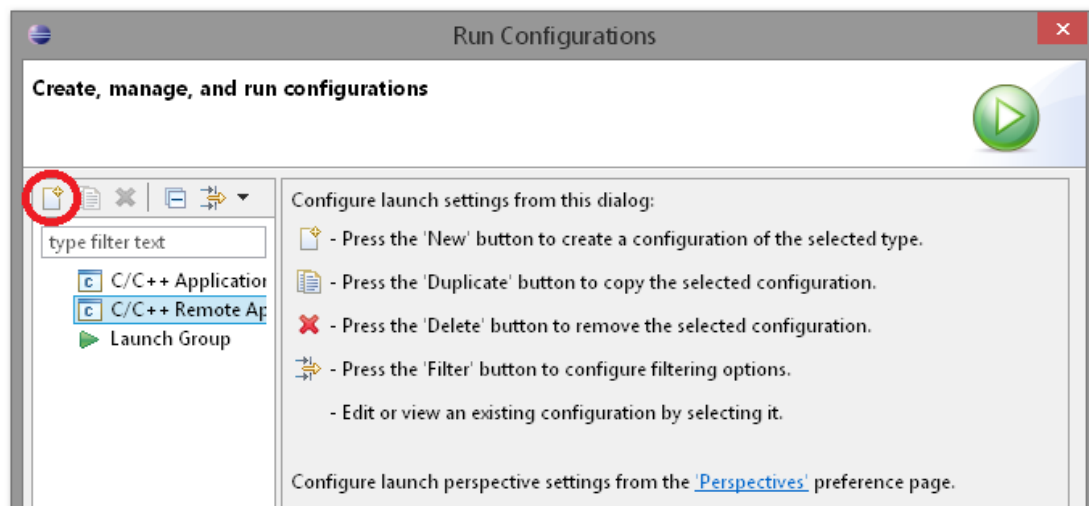
12. Switch to the **C/C++** Perspective.

13. Select **Run»Run Configurations** to open the Run Configurations dialog box.



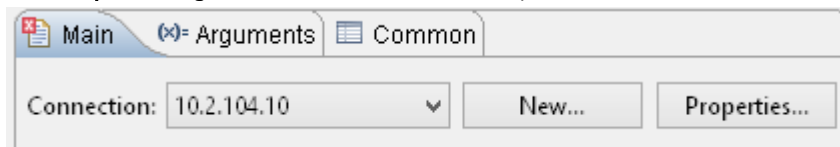
14. Select **C/C++ Remote Application** in the left pane.

15. Click the **New launch configuration** button, circled in the following image, to specify settings for running an executable on your target.



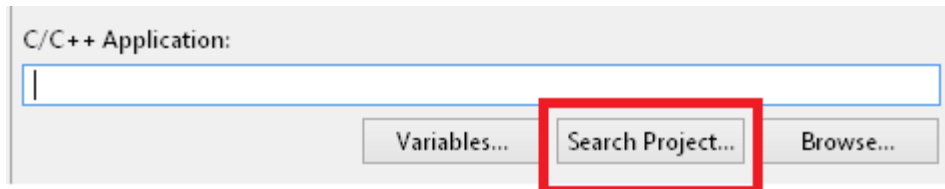
16. Change the name from New\_configuration to **LVRIOEval\_DeployConfig**

17. Select **your target** from the **Connection** pull-down menu.

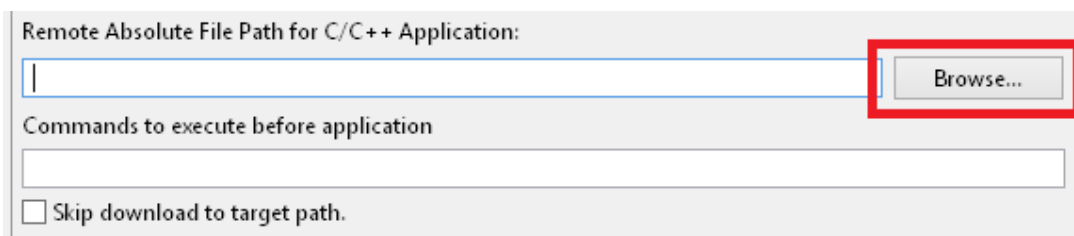


18. Click **Browse** next to **Project:** and select LVRIOEval\_C\_Exercise1.

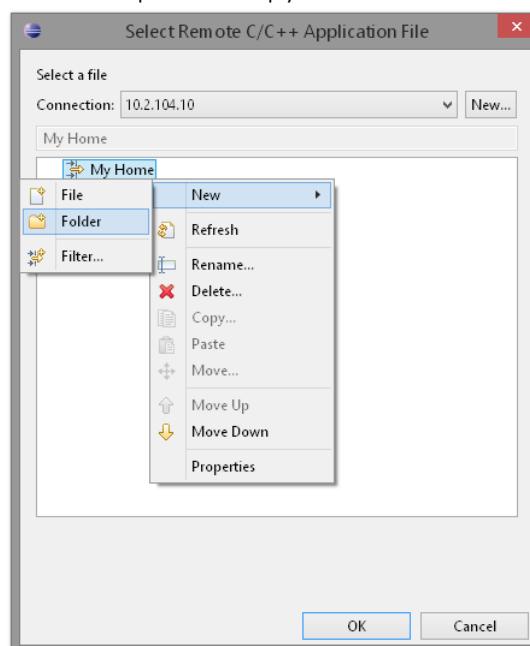
19. Click **Search Project** next to **C/C++ Applications** and select LVRIOEval\_C\_Exercise1.



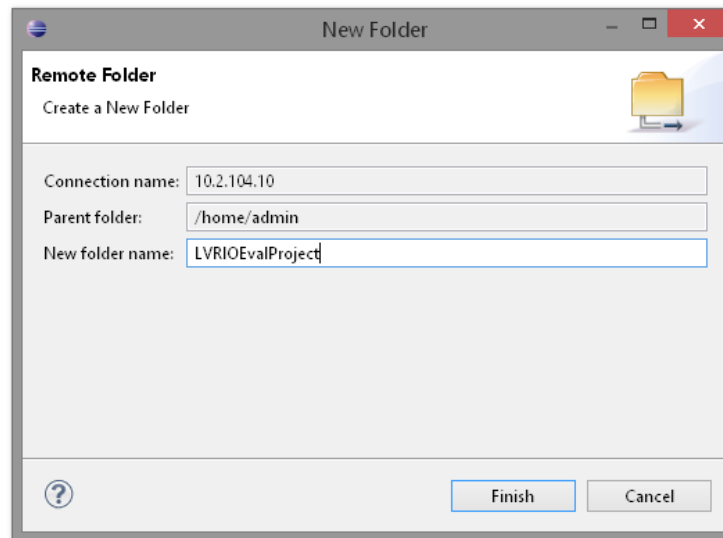
20. Click the **Browse** button beside the **Remote Absolute File Path** for C/C++ Applications text box to open the Select Remote C/C++ Application File dialog box.



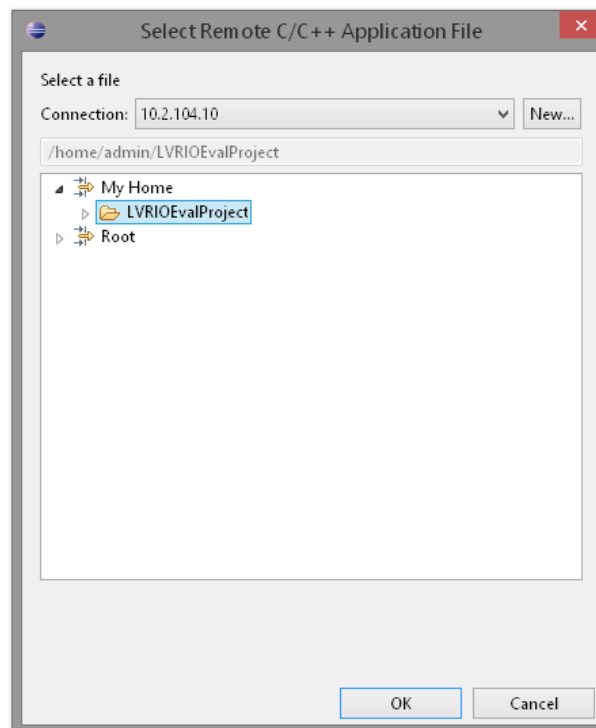
21. Right-click the **My Home** directory in the listbox and select **New»Folder** to create a folder on the target in which to place a copy of the executable.



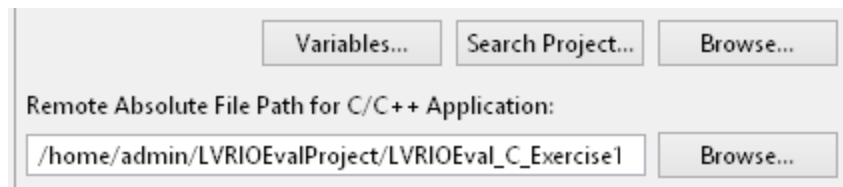
22. Enter a name such as “LVRIOEvalProject” and click **Finish**.



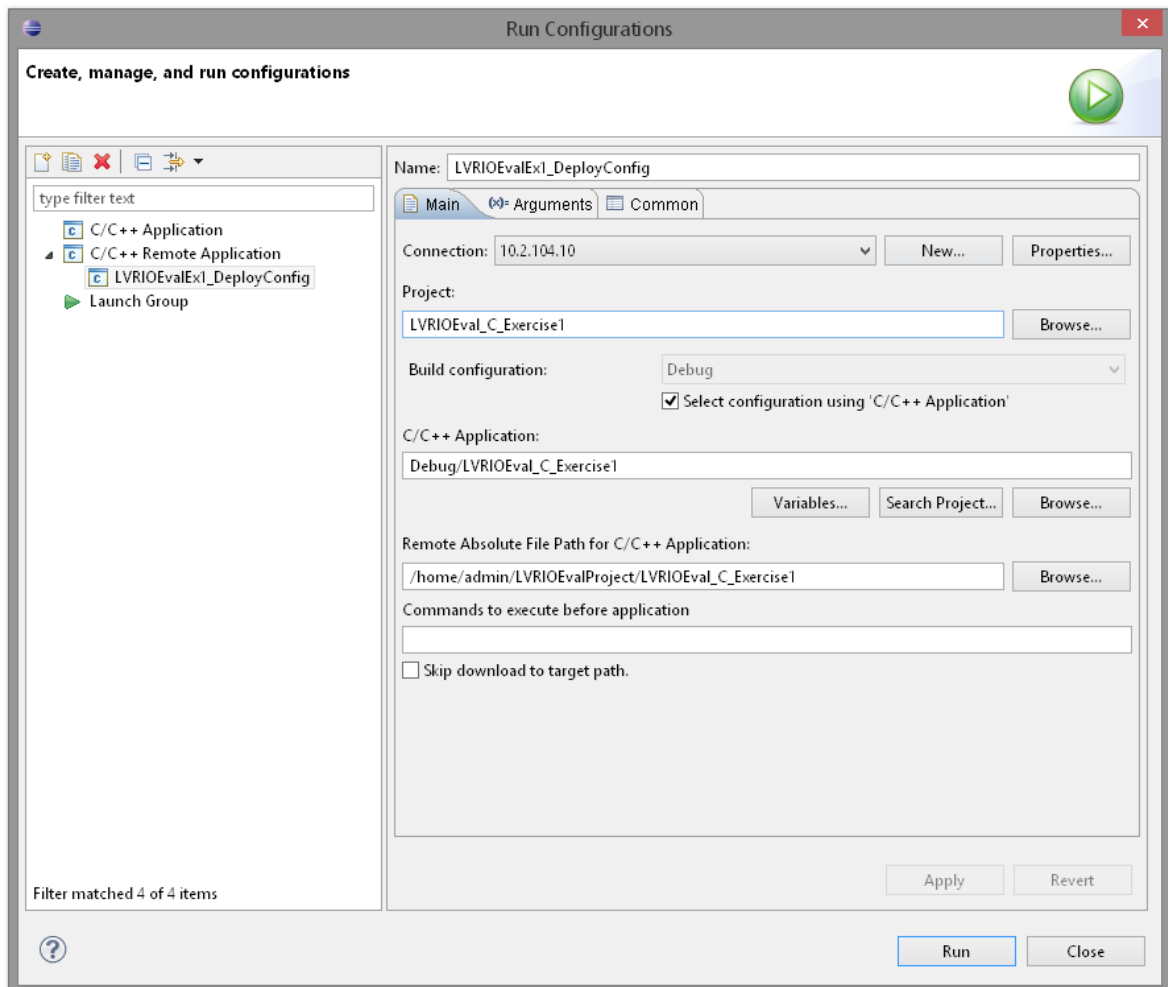
23. Click the new folder and Press **OK**.



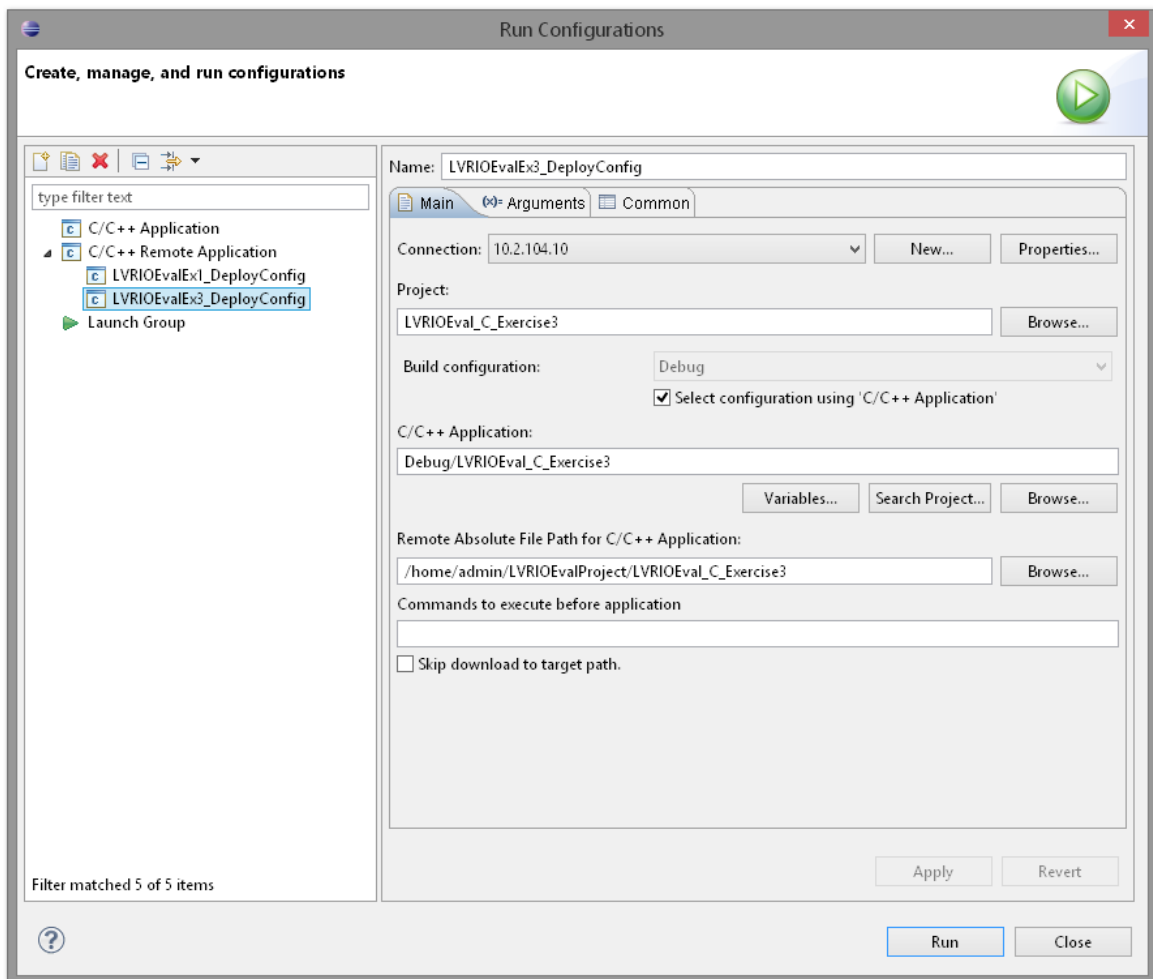
24. Append your project name to the file path populated in the **Remote Absolute File Path** for **C/C++ Applications** text box.



25. Click **Apply**. The Run Configuration should now look similar to the following:



26. Repeat steps 13 to 23 creating another Run Configuration for the **Exercise 3** project. The Exercise 3 files can also share the same remote folder structure as Exercise 1 but should use a different file name. The Run Configuration should look similar to the following.



27. Click **Apply** and **Close** to finalize your changes.

Both Exercise 1 and Exercise 3 rely on FPGA bitfiles created using LabVIEW FPGA. The projects expect bitfiles in the following locations on the target,

- Exercise 1: /home/admin/LVRIOEvalProject/NiFpga\_LEDPWMFPGA.lvbitx
- Exercise 3: /home/admin/LVRIOEvalProject/NiFpga\_MachineInspectFPGAC.lvbitx

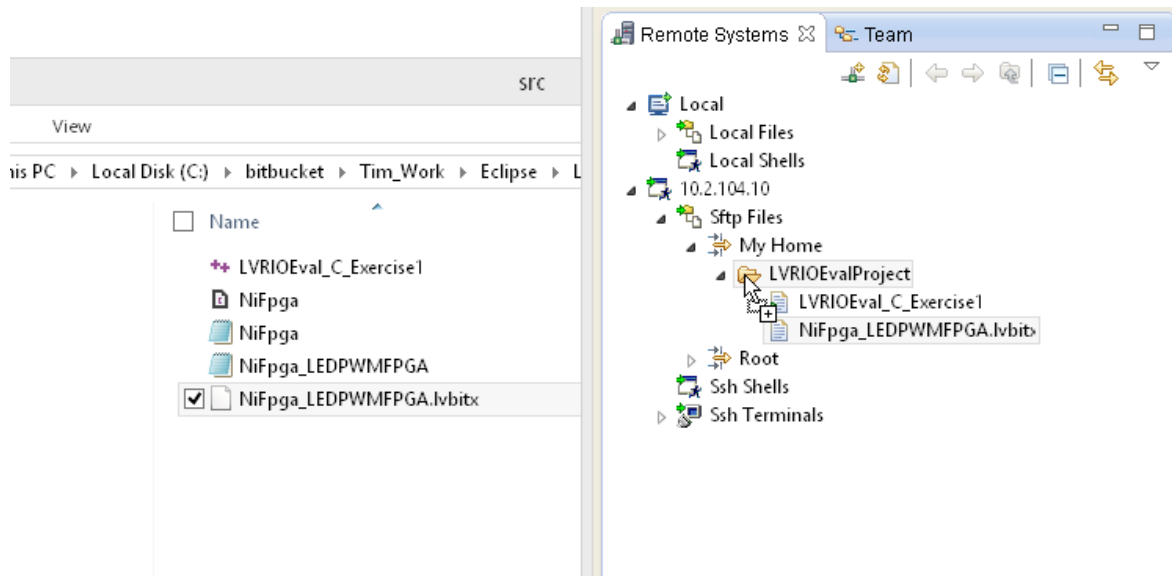
The FPGA bitfiles are located on the host in each of the respective project folders,

- .../LVRIOEval\_C\_Exercise1/src/NiFpga\_LEDPWMFPGA.lvbitx
- .../LVRIOEval\_C\_Exercise3/src/NiFpga\_MachineInspectFPGAClvbitx

Be sure to replace any existing bitfile already found in this directory, otherwise you will get **Error -63101**.



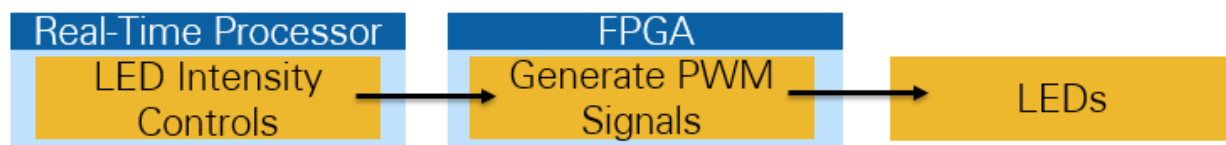
28. Copy both FPGA bitfiles to the directory on the target by drag and dropping the lvbitx file in the Eclipse *Remote Systems Explorer*. The following image shows an example of transferring the Exercise 1 bitfile to the remote target using Eclipse to drag and drop.



## Exercise 1 | Open and Run Application

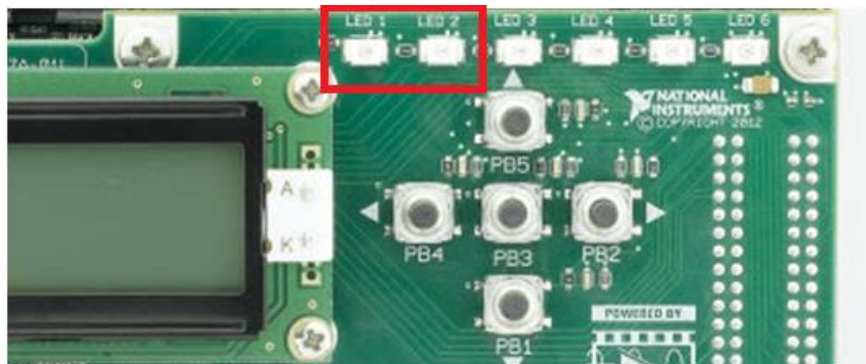
### Summary

In this exercise you will learn how to navigate the Eclipse Project included with your evaluation kit, then open and run a pre-compiled application with code running on both the FPGA and RT Processor. This application simulates precise intensity control of two lasers (LEDs), which need to be pulsed at a high frequency to prevent damage of the part under inspection. The FPGA is used to implement a pair of high frequency and reliable pulse width modulation (PWM) channels.

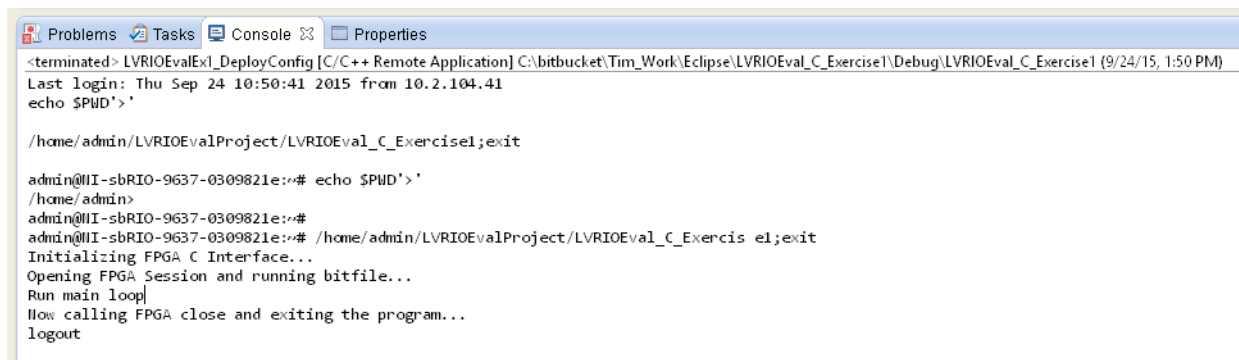


### Explore the Application

1. From Eclipse, select **Run»Run Configurations** to open the Run Configurations dialog box.
2. Select the "LVRIOWalEx1\_DeployConfig" on the left side and click **Run**.
3. The Console tab will show the progress of the application deployment and execution on the remote target.
4. LEDs 1 and 2 should pulse on and off periodically.



- The *Console* tab below the Editor window will also provide useful deployment, execution, and debugging information

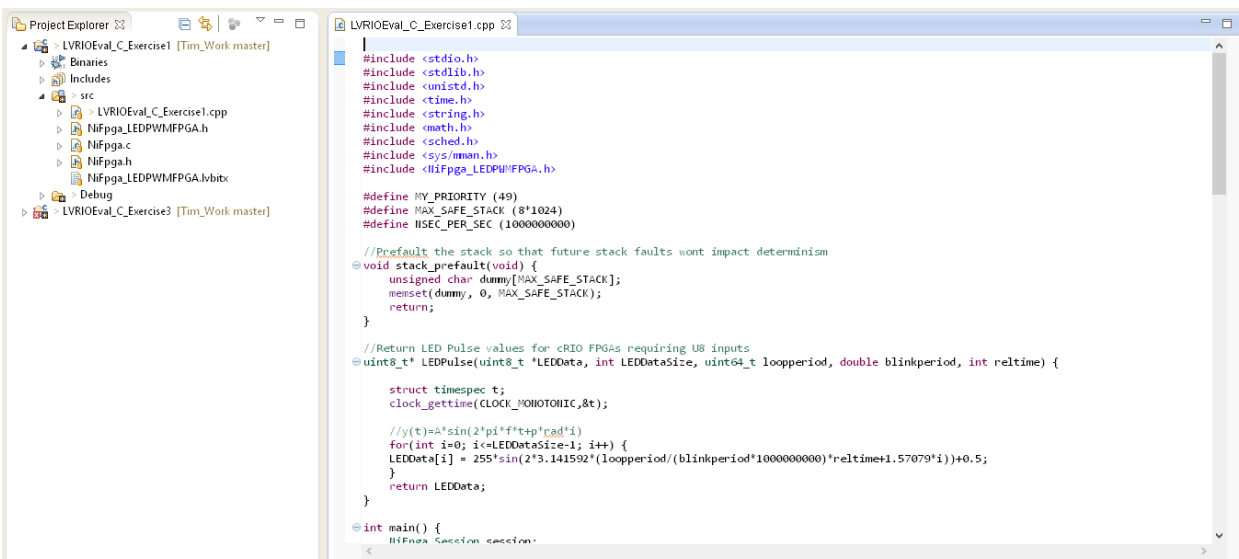


```
<terminated> LVRIOEvalExl_DeployConfig [C/C++ Remote Application] C:\bitbucket\Tim_Work\Eclipse\LVRIOEval_C_Exercise1\Debug\LVRIOEval_C_Exercise1 (9/24/15, 1:50 PM)
Last login: Thu Sep 24 10:50:41 2015 from 10.2.104.41
echo $PWD>'

/home/admin/LVRIOEvalProject/LVRIOEval_C_Exercise1;exit

admin@III-sbRIO-9637-0309821e:~$ echo $PWD>'
/home/admin
admin@III-sbRIO-9637-0309821e:~$
admin@III-sbRIO-9637-0309821e:~$ /home/admin/LVRIOEvalProject/LVRIOEval_C_Exercise1;exit
Initializing FPGA C Interface...
Opening FPGA Session and running bitfile...
Run main loop
Now calling FPGA close and exiting the program...
logout
```

- Expand the LVRIOEval\_C\_Exercise1 project on the left side and double click “LVRIOEval\_C\_Exercise1.cpp” under the “src” folder. This will open the source file for the application.



```
LVRIOEval_C_Exercise1.cpp
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include <math.h>
#include <sched.h>
#include <sys/mman.h>
#include <NiFpga_LEDPWMFPGA.h>

#define MY_PRIORITY (49)
#define MAX_SAFE_STACK (8*1024)
#define HSEC_PER_SEC (1000000000)

//Prefault the stack so that future stack faults wont impact determinism
void stack_prefault(void) {
    unsigned char dummy[MAX_SAFE_STACK];
    memset(dummy, 0, MAX_SAFE_STACK);
    return;
}

//Return LED Pulse values for cRIO FPGAs requiring U8 inputs
uint8_t* LEDPulse(uint8_t *LEDData, int LEDDataSize, uint64_t loopperiod, double blinkperiod, int reltime) {

    struct timespec t;
    clock_gettime(CLOCK_MONOTONIC,&t);

    //y(t)=A*sin(2*pi*f*t+tp*cos(i)
    for(int i=0; i<LEDDataSize-1; i++) {
        LEDData[i] = 255*sin(2*3.141592*(loopperiod/(blinkperiod*1000000000)*reltime+1.57079*i))+0.5;
    }
    return LEDData;
}

int main() {
    //FPGA Session session
```

- Navigate to **Window»Preferences»General»Editors»Text Editors** and check the “Show line numbers” option.
- Press **Apply** and **OK**.
- Here is a brief overview of how this application functions
  - (Lines 37-55) Configure the application to execute in real-time manner. This includes changing priority, performing a stack fault, and locking all of the processes memory.
  - (Lines 61-71) Initialize the RIO Driver interface, Open and run the NiFpga\_LEDPWMFPGA.lvbitx on RIO0 of the sbRIO-9637.

- c. (Lines 73-80) Define variables that define the behavior of the main loop.
- d. (Lines 83-112) Main execution loop. Writes LED intensity values to the FPGA using the `NiFpga_WriteArrayU8()` function provided by the FPGA Interface C API. Timing for this loop is controlled by checking time against `CLOCK_MONOTONIC` and waiting until 15000000 ns (15ms) have elapsed.
- e. (Lines 114-121) Close out FPGA References and exit the main application.

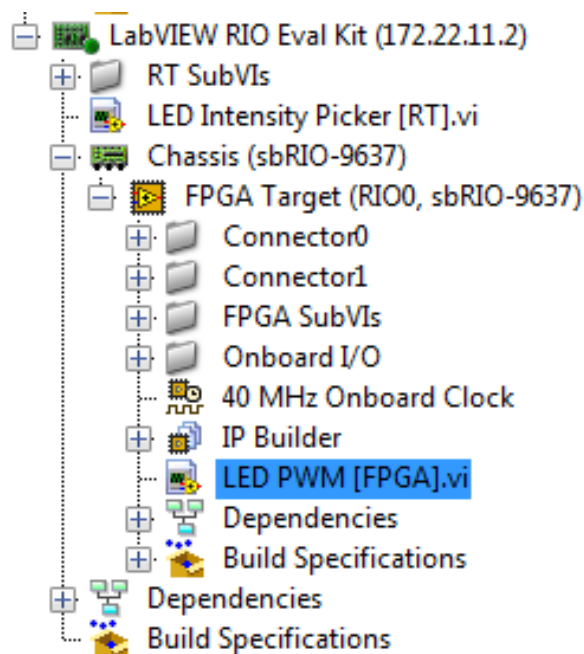
10. Adjust the values of **PulseLEDs**, **BlinkPeriod**, and **LED#IntensityData** variables.

11. Run the **LVRIOEvalEx1\_DeployConfig** Run Configuration again and observe your changes.

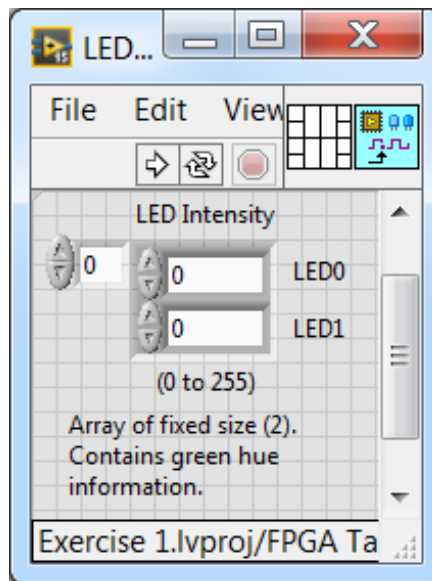
### Review the LabVIEW FPGA code

12. Open up the Exercise 1 LabVIEW project file by navigating to `.\1- Open and Run \Exercise 1.lvproj`.

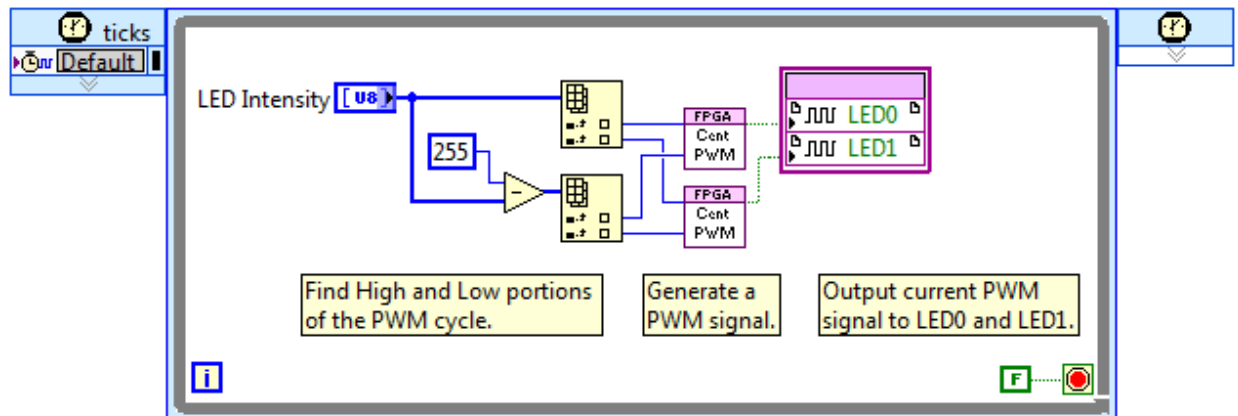
13. In the project, expand out the *LabVIEW RIO Eval Kit* target, Chassis, and then the FPGA Target. Open **LED PWM [FPGA].vi** under the FPGA Target.



14. Note that the front panel of an FPGA application is simple as it is not intended to be used as a User Interface (UI) but instead the *LED Intensity* array control represents an FPGA register that is accessible for communication between the FPGA and the real-time processor, and in this case receives the intensity values coming from the LED Intensity Picker [RT] vi read/write control.



15. Open the block diagram by pressing <Ctrl-E>.

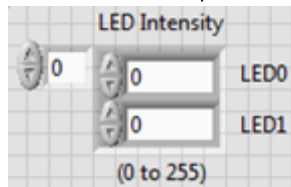


This LabVIEW FPGA code implements a pair of simple PWM channels. They are controlled independently, but are synchronized using LabVIEW Real-Time. Because they are implemented in hardware on the FPGA, the PWM channels are highly reliable and execute with virtually no latency.

Placing the channels in a Timed Loop on the FPGA implements the code within one tick of the default hardware clock (40 MHz), ensuring they execute at a high frequency. The PWM channels change the intensities of the onboard LEDs on your kit labeled **LED0** and **LED1**.

16. The **LED Intensity** control is the resource being written to from the C application you ran earlier.

LV LED Intensity Control:

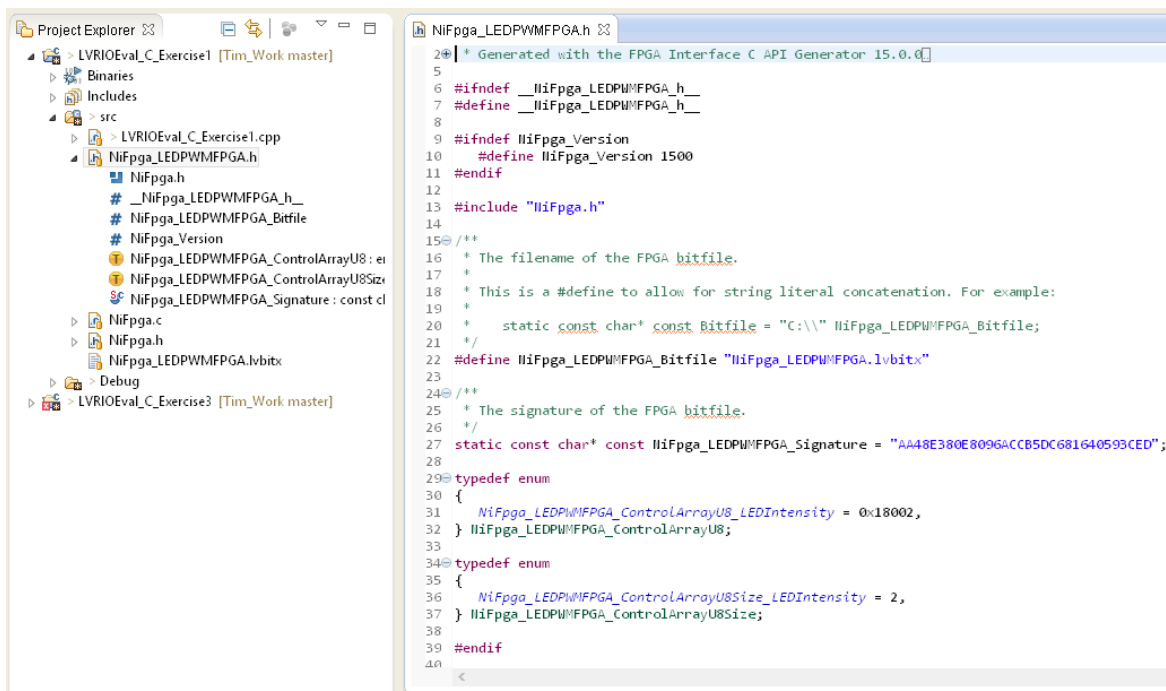


FPGA Interface C API call to the LED Intensity Control:

```
//Send LED Intensity values to the FPGA
NiFpga_MergeStatus(&status, NiFpga_WriteArrayU8(session,
NiFpga_LEDPWMFPGA_ControlArrayU8_LEDIntensity,
LEDIntensityData,
NiFpga_LEDPWMFPGA_ControlArrayU8Size_LEDIntensity));
```

17. In addition to providing a calling API for the NI RIO driver, the FPGA Interface C API also provides a tool called the **FPGA Interface C API Generator**. This tool will take your LabVIEW FPGA bitfile and produce a unique C header file that reveals the register locations for the controls and indicators in your bitfile.

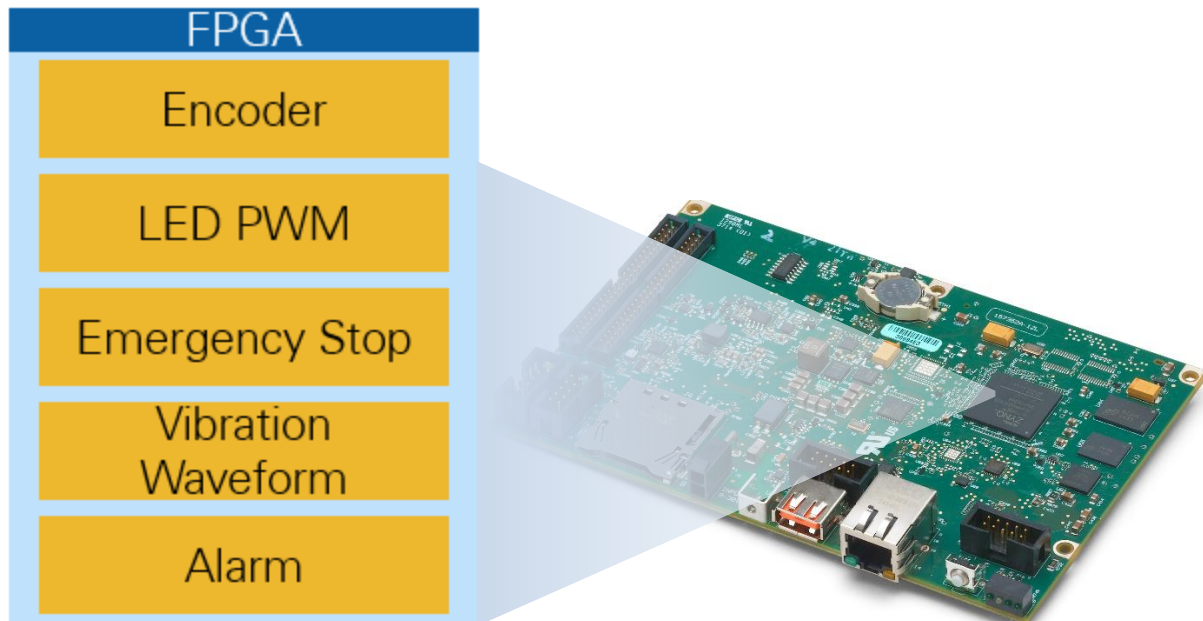
The FPGA Interface C API Generator was used to create the header file used in this exercise. You can review this header file in the Eclipse project by double clicking on **LVRIOEval\_C\_Exercise1»src»NiFpga\_LEDPWMFPGA.h**.



## Exercise 2 | Create a Monitoring and Control FPGA Application

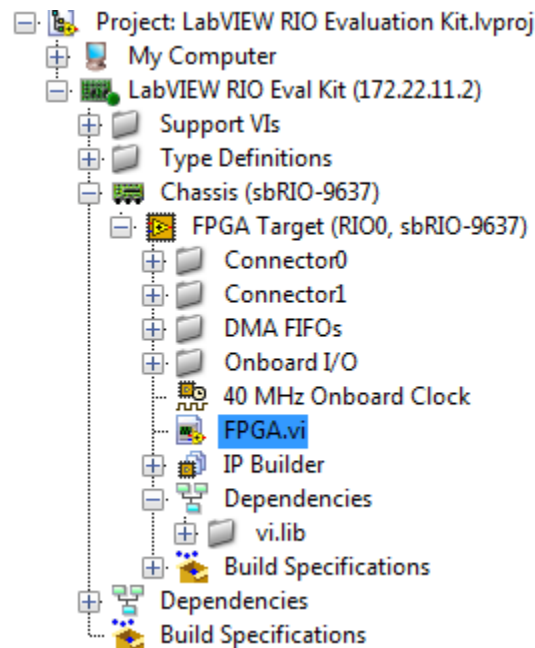
### Summary

Learn how to create and control I/O on your kit's daughter board through the FPGA. The FPGA code controls the conveyor belt and measures sensor signals. You will complete the logic then compile it to run on the FPGA. You will interact with the following I/O on the board: Quadrature Encoder, push buttons, LEDs, amplitude and frequency knobs.

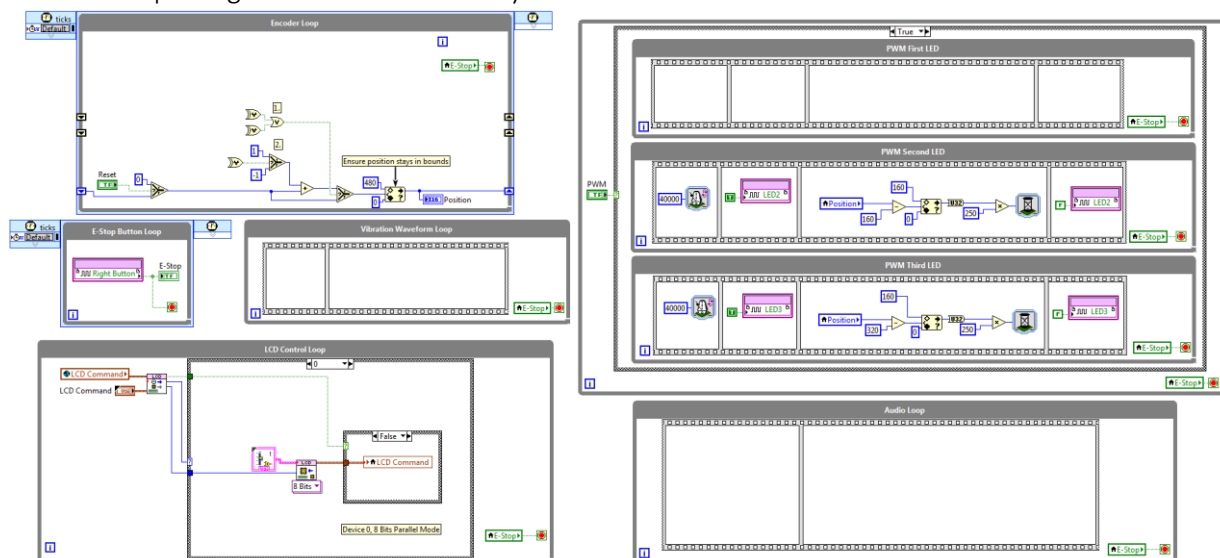


## Implementation

1. Open LabVIEW RIO Evaluation Kit.lvproj from .\2- Create FPGA Application directory.
2. Expand the LabVIEW RIO Eval Kit target > Chassis > FPGA Target in the project and double click to open the **FPGA.vi**

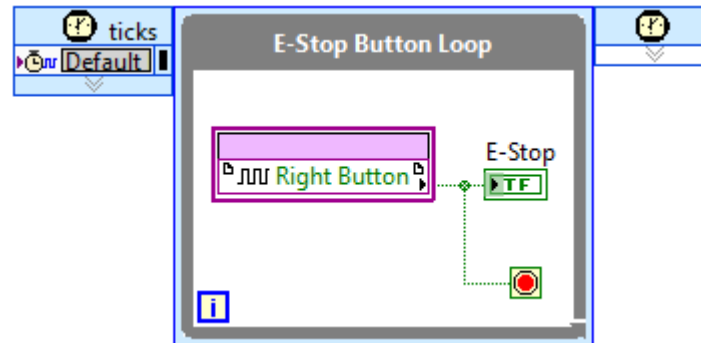


3. Open the block diagram by pressing <Ctrl-E>. Notice that the outline of several loops have already been created for you. This is mainly infrastructure logic, and you will be completing the core functionality.

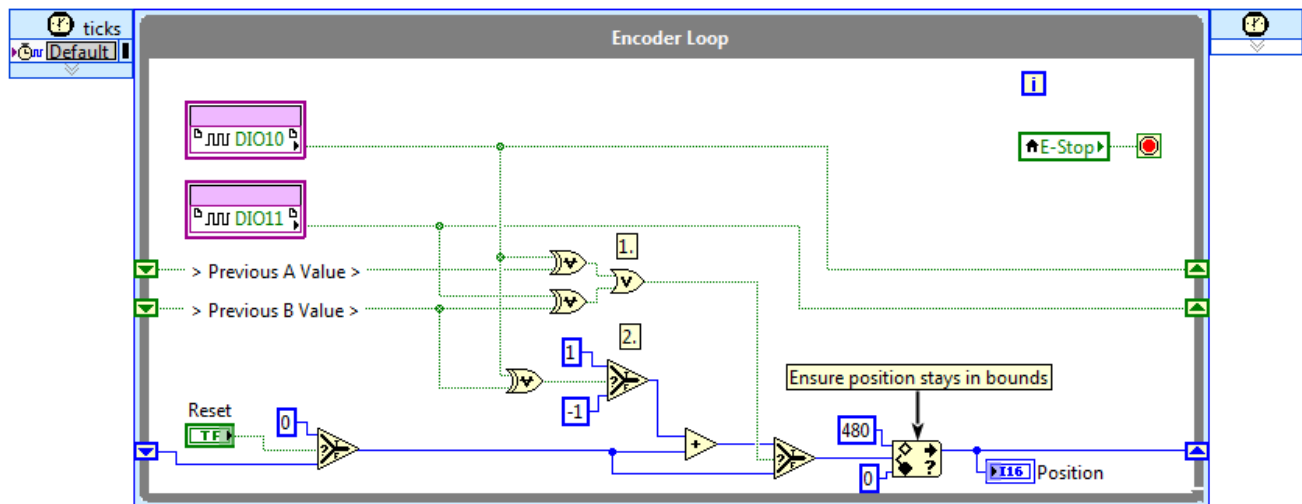




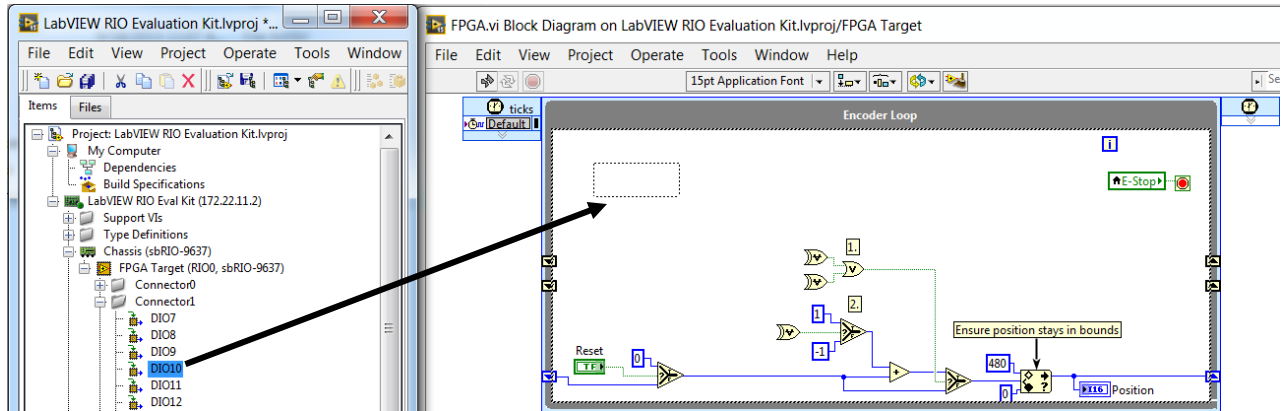
4. First, view the *E-Stop Button Loop* as shown below. If the Right 'Stop' button (PB2) is pressed on the eval board, it will set a flag to stop all the other FPGA loops.



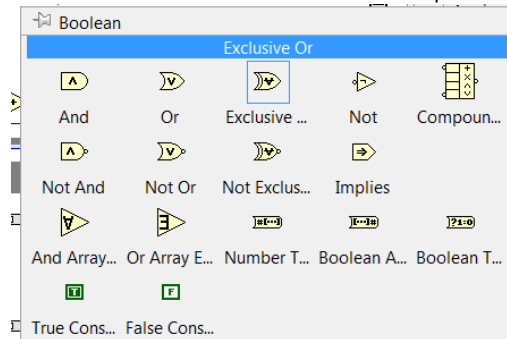
- **I/O Node**—Underneath *FPGA Target* in your project, open the *Connector1* folder and find *Right Button* (DIO 13, digital pin). This I/O node was dragged into the loop from the project and represents the right push button (PB2) on your board. You can rename I/O to something meaningful by selecting the item and pressing **F2** on your keyboard.
  - **E-Stop**—This is the global stop button for the application, a local variable has been created to stop all the other FPGA loops based off of this boolean value.
  - **Timed Loop**— a Timed Loop on the FPGA implements the code within one tick of the default hardware clock (40 MHz)
5. Complete the *Encoder Loop* as shown below. This loop will constantly read the value of the quadrature encoder.



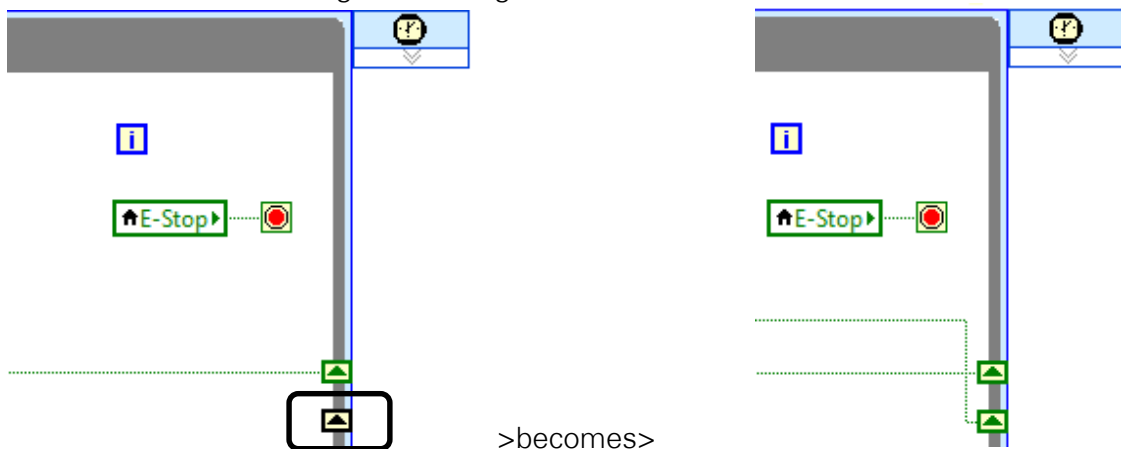
- **I/O Nodes**—Underneath *FPGA Target* in your project, open the *Connector1* folder. Drag I/O items DIO10 and DIO11 onto the block diagram. These digital lines are connected to the encoder.



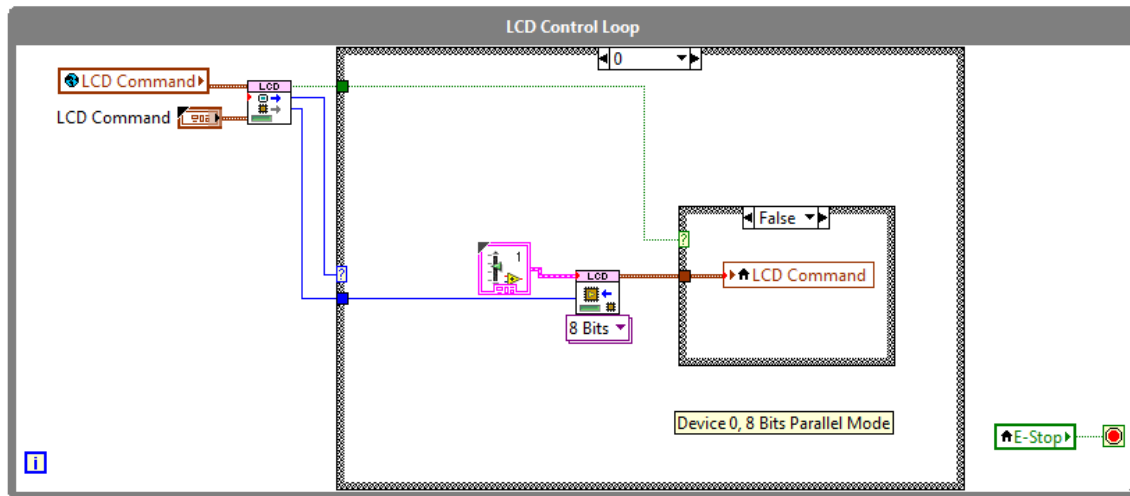
- **Boolean Logic**—Right-click to bring up the *Functions* palette and find the *Exclusive Or*, and *Or* functions in the *Boolean* subpalette.



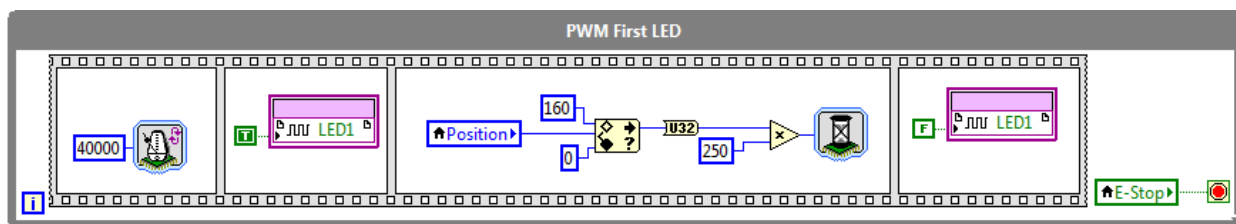
- **Shift Register**—Wire the boolean output value from **DIO 10** to the upper shift register and **DIO 11** into the lower shift register on the right edge of your loop, as shown below. These shift registers will carry those values between loop iterations to compare the position.
- Wire the remaining boolean logic



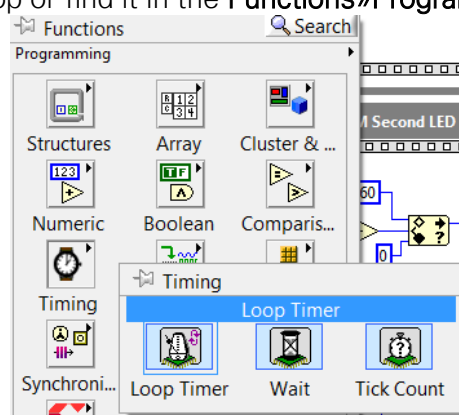
6. The *LCD Control Loop* serves as the driver that controls the LCD through the 16 DIO pins connecting it. This is only being used in the LabVIEW Real-Time tutorial.



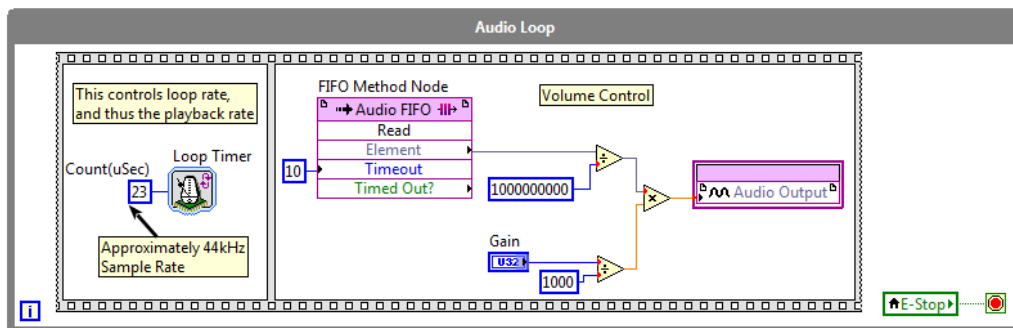
7. Complete the *PWM First LED* loop as shown below. This loop will be nearly identical to the two loops below it. You can highlight and copy the logic from the other loops, but take note to update the integer constant values. The goal of this loop is to turn an LED on with increasing brightness and off using PWM based on the position of the encoder. LED1 LED2, and LED3 on your board are connected to the digital lines DIO4, DIO5, and DIO6 respectively.



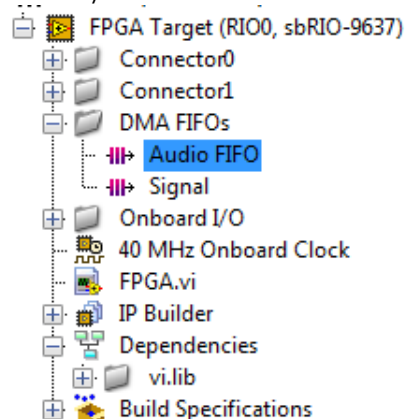
- **I/O Nodes**—Drag in two instances of LED1 from the project into frame two and four of the sequence structure as shown. I/O Nodes can be changed to write by right clicking on the middle of the node and selecting **Change to Write**.
- **Loop Timer**—This function controls how fast the loop executes. In this case, the loop executes once every 40,000 clock cycles (ticks). You can copy this from the *PWM Second LED* loop or find it in the **Functions»Programming»Timing** palette.



- **Position local variable**—create a local variable from the **Position** indicator in the *Encoder Loop* by Right-clicking on the indicator and selecting **Create»Local Variable**. Drag it from the Encoder Loop down to the PWM First LED loop and wire as shown.
  - **In Range and Coerce**—This logic prevents the loop from having timing violations when the position of the encoder continues to increase. Use the **Quick Drop** menu <Ctrl-Spacebar> to find and insert this VI if you did not copy it from another loop.
  - **Wait Timer**—The wait timer delays the execution of the loop for the specified number of clock cycles and is also found at **Functions»Programming»Timing**.
  - Complete the remaining integer logic as shown above.
  - **E-Stop Local Variable**—Note the local variable for the E-Stop indicator is wired to the stop terminal.
8. Complete the *Audio Loop* as shown below. This loop will read waveform data from the Audio\_Left FIFO being sent from your real-time VI and output audio to the speaker connected to the AO0 terminal.

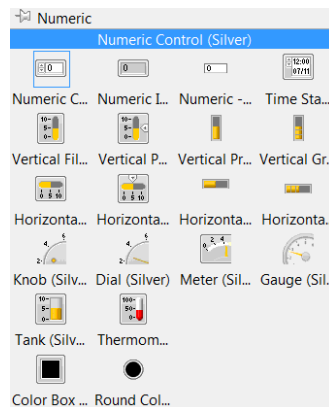


- **Loop Timer**—Find it in the **Functions»Programming»Timing** palette. Double-click the VI and set *Counter Units* to **uSec**. Place and wire a **Numeric Constant** to the **Count (uSec)** input terminal. A loop rate of 23 uSec is approximately 44kHz, which is typical for audio applications.
- **I/O Node**—The Audio Output node can be found in the project *Connector0* folder. Drag it into the block diagram in the second frame like you did earlier.
- **FIFO Read**—Open the *DMA FIFOs* folder beneath the *FPGA target* in your project and drag **Audio\_FIFO** onto the block diagram. This adds a FIFO read method. This function will allow the FPGA VI to read waveform data streamed from the Real-Time application you will create in the next exercise.

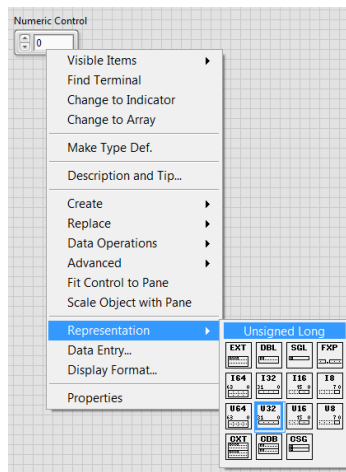


Direct Memory Access (DMA) FIFOs directly accesses memory to transfer data from FPGA target VIs to host VIs and vice versa, allocating memory on both the host computer and the FPGA target, yet acts as a single FIFO. Right-click the *Timeout* input and **Create»Constant** with the value as **10**, which specifies the number of clock ticks that the method waits for available space in the FIFO if the FIFO is full.

- **E-Stop Local Variable**—Note the local variable for the E-Stop indicator is wired to the stop terminal.
- **Gain**—The waveform data from the FIFO is multiplied by the **Gain** as a form of volume control. Press <Ctrl-E> to switch to the Front Panel and right click to bring up the *Controls* palette. Place down a control from **Controls»Silver»Numeric»Numeric Control**. Rename the control **Gain**.

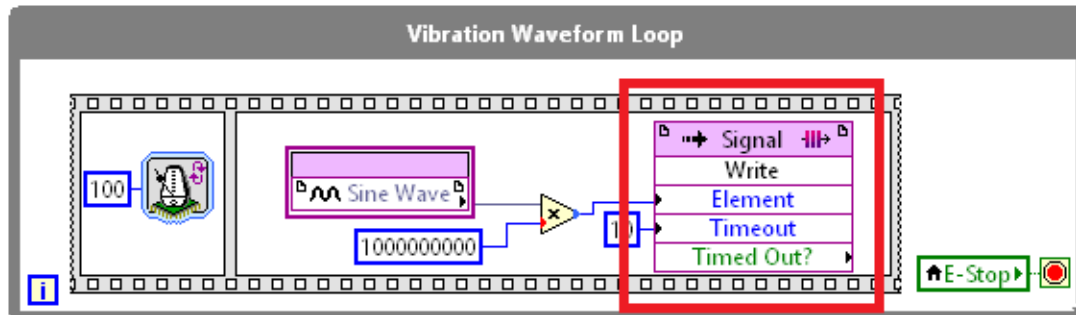


Right-click on the control and select **Representation** then choose **U32 (Unsigned Long)**.

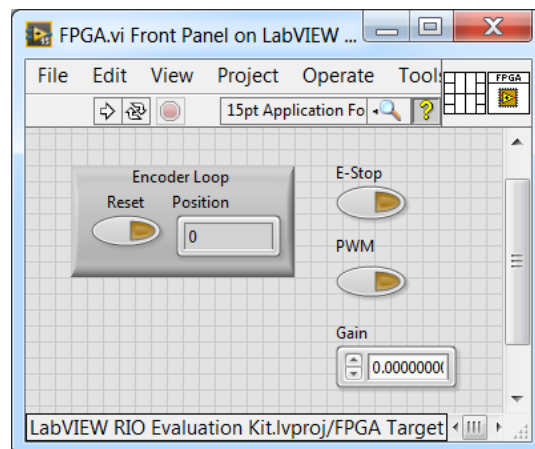


- Wire each of the **Element** and **Gain** outputs to the top input of a **Divide** node (**Functions»Programming»Numeric**). Then right-click **Create»Constant** on the bottom input of the **Divide** nodes. Right-click and change the **Representation** to **U32** for these constants as well.
- Enter in **1,000,000,000** and **1,000** for the constants respectively for scaling.
- Wire those two outputs into a **Multiply** node (**Functions»Programming»Numeric**) and then into the **Audio Output** I/O node.

9. Complete the *Vibration Waveform Loop* as shown below. This loop is taking a simulated vibration measurement, generated by the sine wave function generator on your board, at a given frequency and amplitude that you change with the knobs on your board.



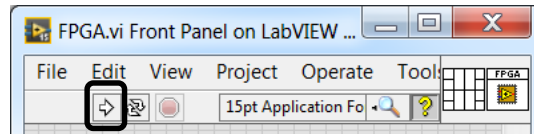
- **Loop Timer**— Find it in the **Functions»Programming»Timing** palette. Double-click the VI and set *Counter Units* to **uSec**. Right-click on the input terminal and select **Create»Constant**. Type in a loop rate of 100 uSec.
  - **I/O Node**—The **Sine Wave** I/O node (AI0) can be found in the project in the *Connector0* folder. Drag it into the block diagram.
  - Wire the *Sine Wave* output into a **Multiply** node, then right-click **Create»Constant** on the bottom input of the **Multiply** node entering **1,000,000,000** for scaling.
  - **FIFO Write**— Open the *DMA FIFOs* folder beneath the *FPGA target* in your project and drag **Signal** onto the block diagram. This adds a FIFO write method. This function will allow the FPGA to stream waveform data to the RT VI. Wire the output of the **Multiply** function into *Element* input. Right-click the *Timeout* input and **Create»Constant** with the value as **10**.
  - **E-Stop Local Variable**—Note the local variable for the E-Stop indicator is wired to the stop terminal.
10. Arrange your front panel controls and indicators as shown below and ensure the spelling and capitalization are correct, as we will be referencing them by name in Exercise 3.



11. Save the VI as `Machine Inspect [FPGA]C.vi`.

## Start LabVIEW FPGA Compilation Process

1. Save the FPGA VI and click the **Run** button to start the compilation process.

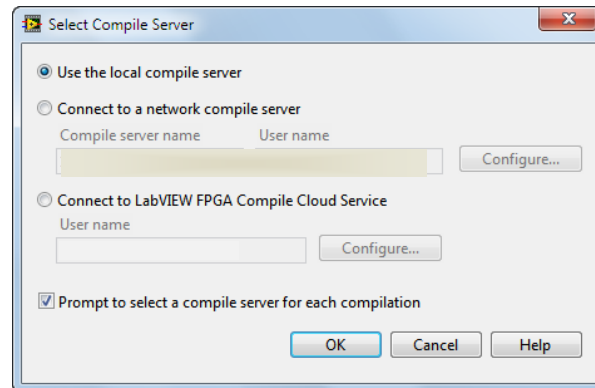


In contrast to LabVIEW applications running on a Windows OS, an FPGA-based VI is compiled down to a bitfile, which is loaded onto the FPGA chip configuring its logic cells and I/O blocks to implement the requested logic. This is a two-step process. First LabVIEW generates intermediate files and then it transfers them to the Xilinx compilation tools for the final compile stage. To learn more about simulating your LabVIEW FPGA code before compiling, please read [Testing and Debugging LabVIEW FPGA Code](#).

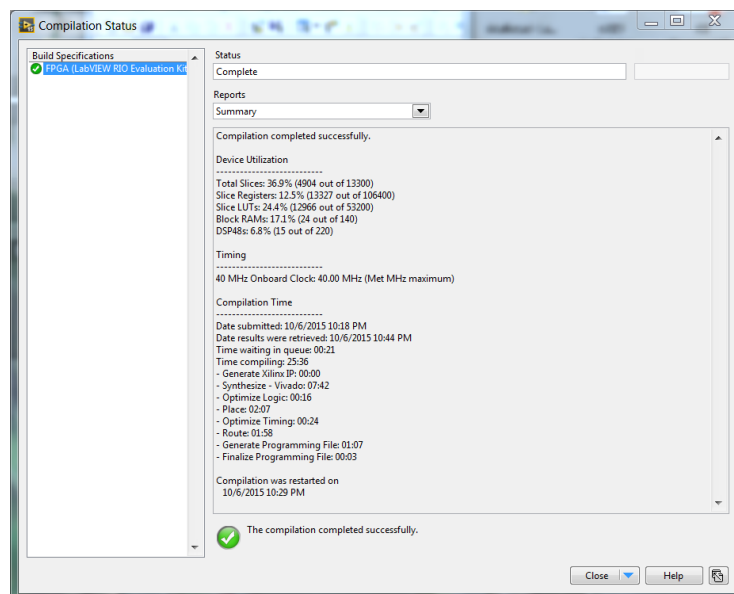
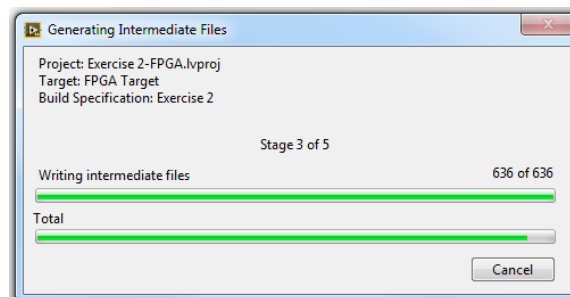
2. In the *Select Compile Server* dialog box that appears, there are three options to compile your code. We will not consider the network compile server in this evaluation experience.
  - a. The recommended option is the [NI LabVIEW FPGA Compile Cloud Service](#), which utilizes high-performance servers that compile 30-50% faster than when installed on your development machine and can execute multiple parallel compilations. Click on the link above or navigate to [ni.com/trycompilecloud](http://ni.com/trycompilecloud) and enter your preferred email address. NI will send you information to quickly setup your NI Cloud Services Portal account.

To enter your credentials in LabVIEW, select the **Connect to LabVIEW FPGA Compile Cloud Service** option. Press the **Configure...** button to enter the NI Cloud Services Portal login credentials you just created. You will have full access to this service during your 90 day evaluation and long term when you purchase LabVIEW you will have access included for free with your active Standard Service Program (SSP) membership.

- b. The other option is to install the *LabVIEW FPGA Xilinx Vivado 2014.4 Compilation tools* locally to your computer from the second DVD that was included in your kit or download from [ni.com](http://ni.com) [here](#). Note you will need around 5 GB of free space on your hard drive to install these tools and it is required if you do not have internet access. Once installation is complete, select **Use the local compile server** to use this option.



- Click **OK** and the intermediate file generation process will automatically start, where LabVIEW translates your graphical implementation to the native Hardware Description Language for an FPGA. Once complete, LabVIEW will kick off the Xilinx compilation tools. In total, the process should take about 20 minutes to finish compilation.



**Note:** If a communication error occurs when the local compile server starts, manually start the Compile Worker by navigating in the Windows start menu to **All Programs»National Instruments»FPGA Compile Tools»FPGA Compile Worker**. Then, once it starts up, click the **Run** button on your LabVIEW FPGA VI again to re-establish communication.



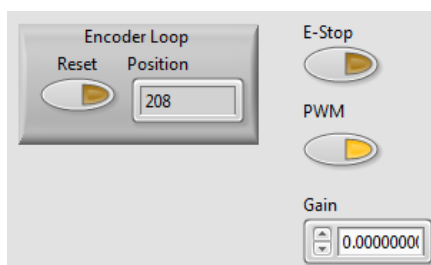
## Test the FPGA Application

When the FPGA VI finishes compiling, you can run the following tests on the application. Once Compilation is complete, in the *Compilation Status* window you can see the device utilization, or various FPGA resources that your code uses on the chip. Unlike a processor, it is safe to use nearly 100% of the FPGA resources, since it is implemented hardware.

1. Double click on the **FPGA.vi** from the Exercise 2 project that should still be open in the background.

**Note:** If you did not correctly complete and compile the Exercise 2 FPGA application, you can reference the solution in the `.\Solutions\2- Create FPGA Application` folder. Run the **FPGA.vi** directly but note that you may need to recompile the *FPGA.vi* for your specific target. Reference the *Using the Solutions* section at the front of the manual for more details on using the solution.

2. Click the **Run** arrow if the VI is not already running and complete the following tests:
  - ✓ Click the **PWM** control to True on the VI front panel. LEDs 1-3 should light up with increasing intensity as you turn the encoder to the right.



- ✓ Press the Stop button (PB2) on your board and ensure the E-Stop indicator on the FPGA VI lights up.



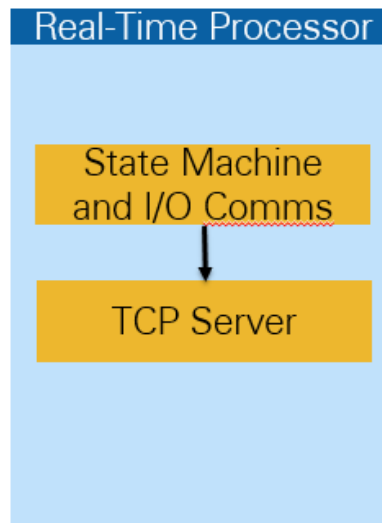
**Note:** Since the real-time application contains the logic to write to the LCD and exchange data with the audio and signal FIFOs, they will not yet update until you complete exercise 3. The next testing section for the real-time application will exercise this functionality.

3. Save LabVIEW RIO Evaluation Kit.lvproj.

## Exercise 3 | Develop Real-Time Application

### Summary

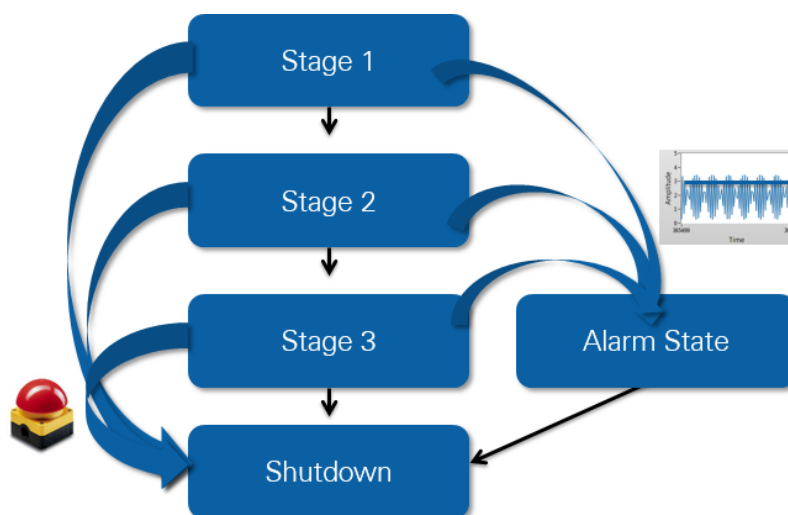
Create a multithreaded C++ application to exchange data with the FPGA VI and host a TCP server to share data across the network.



### State Diagram

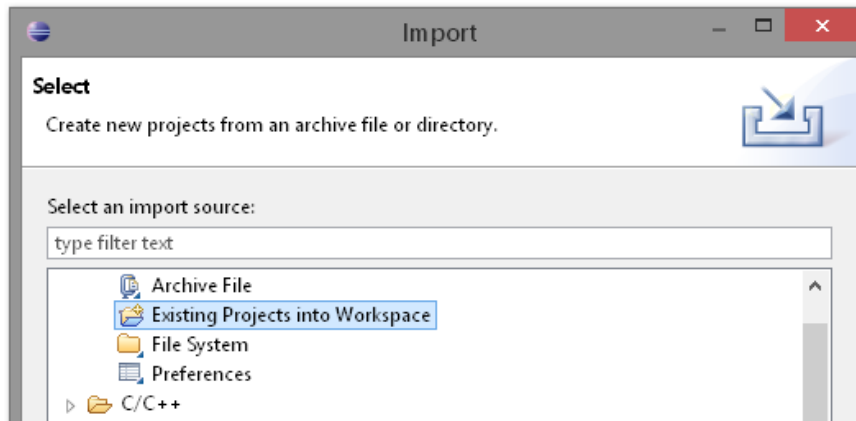
Normal application operation will progress from stage 1 » 2 » 3 then proper shutdown. If the alarm condition is tripped, when the vibration waveform amplitude crosses the threshold, potentially indicating a damaged bearing that a technician should investigate, the application will enter the alarm state and automatically progress to proper shutdown. If you press the Stop button on your board (PB2) you will automatically progress to proper shutdown.

Example: Stage 1-Align part » Stage 2-Weld part » Stage 3-Scan to inspect welded joint

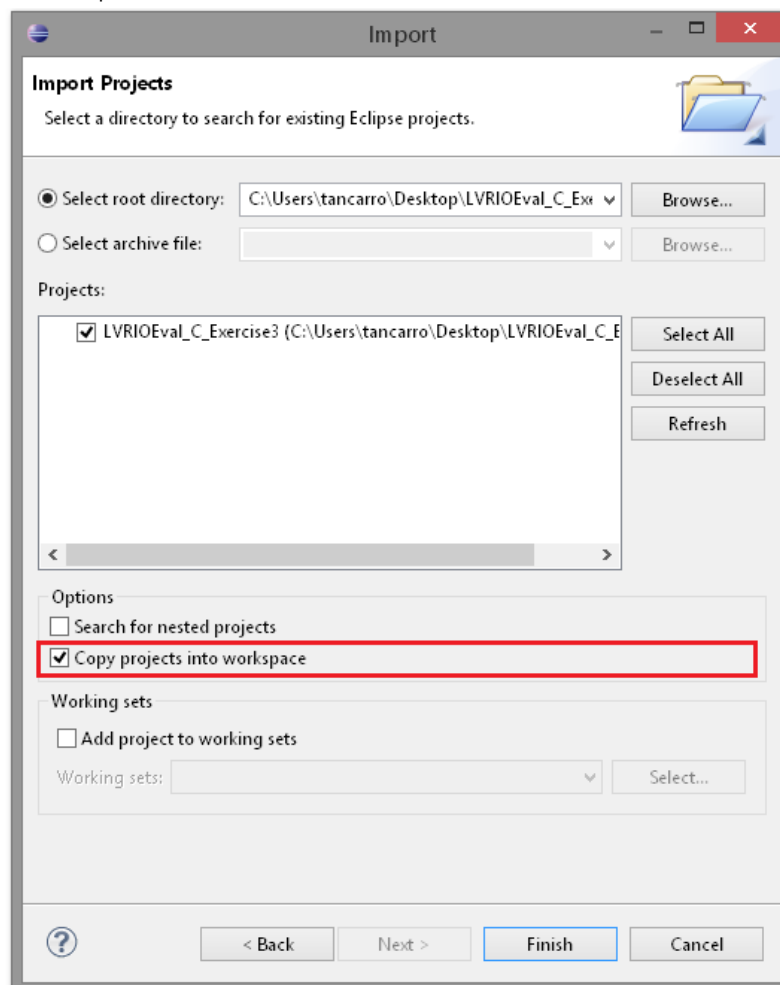


## Implementation

1. Import the Exercise 3 project in Eclipse, go to **File >> Import** and select **General >> Existing Projects into Workspace** and hit Next.



2. Browse to the "LVRIOEval\_C\_Exercise3" folder from the attached zip file and click OK.
3. Ensure "Copy Projects into workspace" is enabled if the project directory is not already in your workspace.



4. Click **Finish**
5. Open **LVRIOEval\_C\_Exercise3.cpp** and notice that the infrastructure is already completed for you; a switch based finite state machine, threaded TCPServer function, RT execution and priority settings.

**Note:** Comment sections with the “(CODE)” header are sections you will implement in later steps. The Stage1 state for example,

```

101         switch(state)
102         {
103         case Stage1:
104             /* (CODE)
105              * FPGA Read EStop Boolean and update the following variable
106              *   niFpga_Bool estop;
107              */
108
109             /* (CODE)
110              * FPGA Read Encoder Position I16 and update the following variable
111              *   int16_t positionCurrent = 0;
112              */
113
114             /* (CODE)
115              * FPGA Write PWM Boolean using a constant value of true
116              */
117
118             //Check if the vibration signal and go to alarm state if alarm threshold is reached
119             if(amplitude >= alarmThres)
120                 state = Alarm;
121             else
122                 //Check if the Encoder Position has reached the threshold and, if so, transition to next stage
123                 if(positionCurrent >= positionThres[0]) {
124                     state = Stage2;
125                 }
126             else
127                 state = Stage1;
128
129             break;

```

6. Also take this time to review the **LVRIOEval\_C\_Exercise3.h** header file. This header file contains all of the “#include” statements, data and data types, and functions that will be used in the main application. Defining these in the header file improves readability of the main source code and promotes reusability of functions and types.
7. The first part you will program is the FPGA Session Function **NiFpga\_Open()** immediately following the **NiFpga\_Initialize()**.

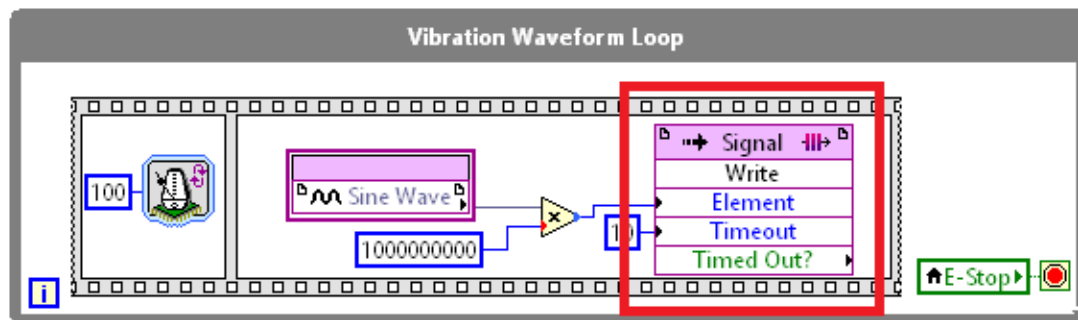
**Note:** It is encouraged that you open the **FPGA Interface C API Help** documentation as a reference when getting started. The “API Reference” section in particular is helpful when getting started.

Open this document (type in the name from the Start menu to locate it) and use it as a reference when programming FPGA accessing functions.

8. The **NiFpga\_Open()** call should look similar to the following and when successfully called will update the "session" variable with a valid FPGA session handle that you can reference later to communicate with the FPGA.

```
NiFpga_MergeStatus(&status, NiFpga_Open("/home/admin/LVRIOEvalProject/NiFpga_MachineInspectFPGAC.lvbitx",
NiFpga_MachineInspectFPGAC_Signature,
"RIO0",
0,
&session));
```

9. Notice that the **NiFpga\_Open()** call references the FPGA bitfile using a static path on the target. During Step 28 of the previous **Initial System Configuration** section you should have distributed the "NiFpga\_MachineInspectFPGAC.lvbitx" to the target. If not, please do so at this time.
10. The next code segment is the Vibration Signal I32 FIFO Read at the beginning of the state machine loop. The Signal FIFO is provided data from the "Sine Wave" input in the Vibration Waveform Loop.



The call signature for FIFO Reads (specifically I32 data type in this case) is the following,

### NiFpga\_ReadFifoI32

```
NiFpga_Status NiFpga_ReadFifoI32(NiFpga_Session session, uint32_t fifo, int32_t*
data, size_t numberOfElements, uint32_t timeout, size_t* elementsRemaining)
```

Reads from a target-to-host FIFO of signed 32-bit integers.

Use the "vibSignalSize", "vibSignal", and "vibFIFOremaining" variables to construct this call. You can use a timeout value of 5000ms. The result should look similar to the following.

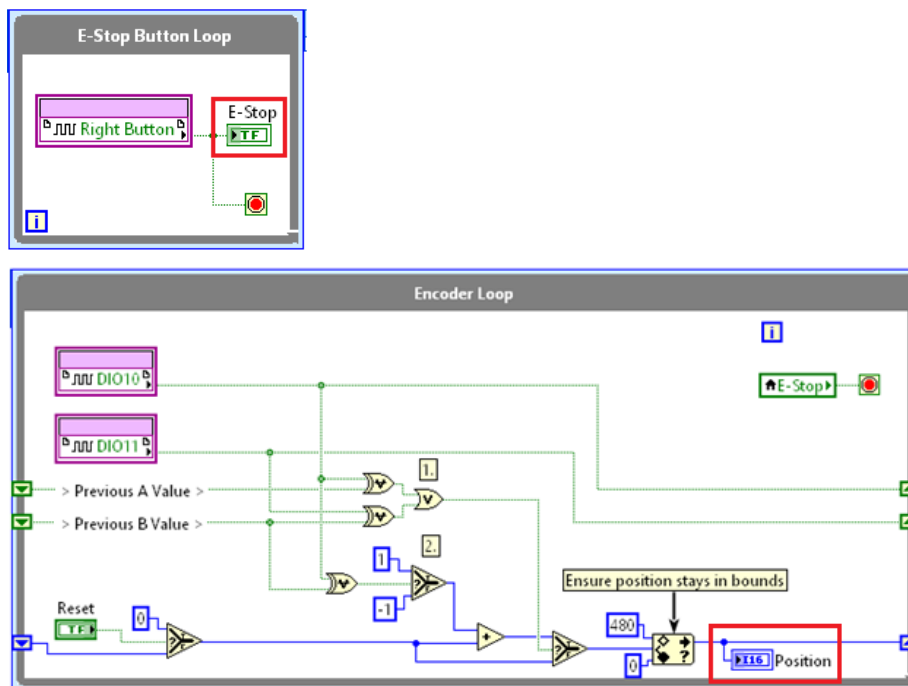
```
NiFpga_MergeStatus(&status, NiFpga_ReadFifoI32(session,
NiFpga_MachineInspectFPGAC_TargetToHostFifoI32_Signal,
vibSignal,
vibSignalSize,
5000,
&vibFIFOremaining));
```

11. This signal is then converted to a double array and passed to the **calc\_amp()** function. This function returns the amplitude value for the signal which you will use to update the **amplitude** variable. That variable is then compared against the **alarmThres** variable in each of the three Stages.

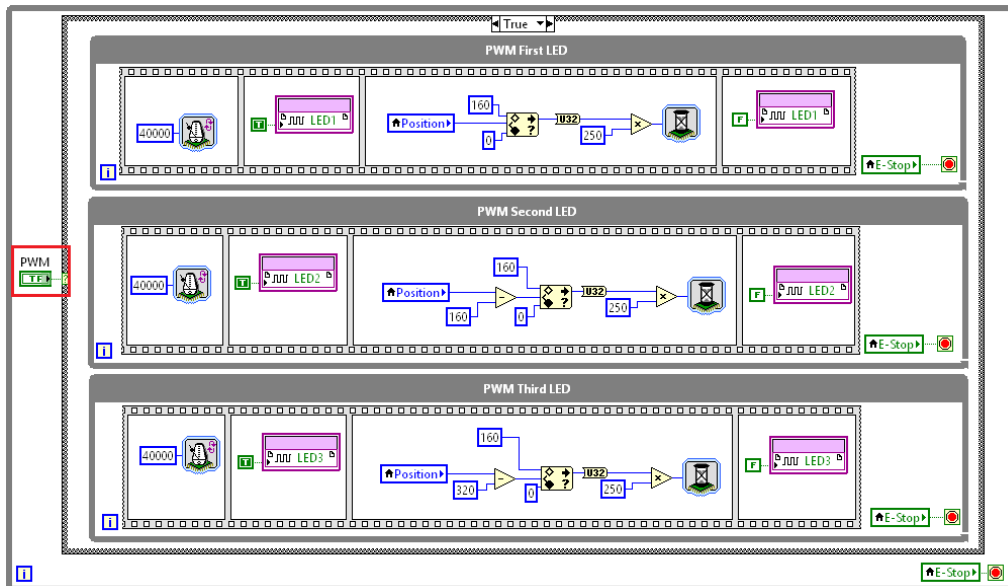
```
//Convert I32 Vibration Signal array into a double array
for(int i = 0; i < vibSignalSize; i++) {
    vibSignald[i] = vibSignal[i]/10000000.0;
}

//Determine amplitude of the vibration signal using the "calc_amp" function defined in LVRIOEvalEx3.h
amplitude = calc_amp(vibSignald, vibSignalSize);
```

12. Each Stage starts with a Boolean E-Stop and I16 Position read from FPGA. These outputs are produced in the FPGA code within the E-Stop Button Loop and Encoder Loop.



13. Each stage also writes to the Boolean PWM value which determines the output settings for the LEDs 1 through 3.



Here are the call signatures for each,

#### NiFpga\_ReadBool

```
NiFpga_Status NiFpga_ReadBool(NiFpga_Session session, uint32_t indicator, NiFpga_Bool* value)
```

Reads a boolean value from a given indicator or control.

#### NiFpga\_ReadI16

```
NiFpga_Status NiFpga_ReadI16(NiFpga_Session session, uint32_t indicator, int16_t* value)
```

Reads a signed 16-bit integer value from a given indicator or control.

#### NiFpga\_WriteBool

```
NiFpga_Status NiFpga_WriteBool(NiFpga_Session session, uint32_t control, NiFpga_Bool value)
```

Writes a boolean value to a given control or indicator.

In the "Stage1" case update the **estop** and **positionCurrent** variables with the read functions and write a constant true value for the Boolean write.

The result should look similar to the following,

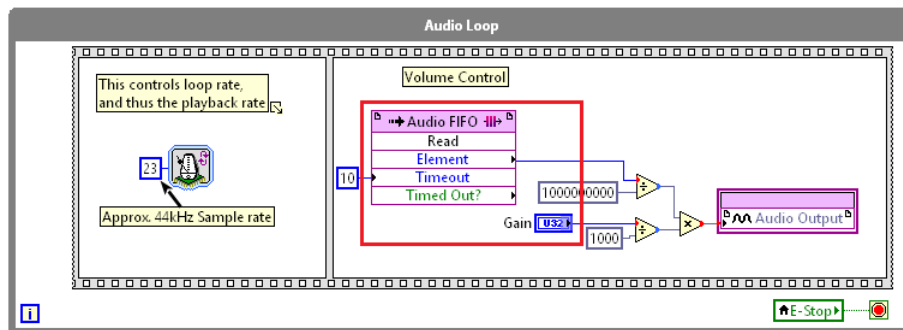
**case Stage1:**

```
NiFpga_MergeStatus(&status, NiFpga_ReadBool(session,
                                             NiFpga_MachineInspectFPGAC_IndicatorBool_Estop,
                                             &estop));

NiFpga_MergeStatus(&status, NiFpga_ReadI16(session,
                                             NiFpga_MachineInspectFPGAC_IndicatorI16_Position,
                                             &positionCurrent));

NiFpga_MergeStatus(&status, NiFpga_WriteBool(session,
                                             NiFpga_MachineInspectFPGAC_ControlBool_PWM,
                                             true));
```

14. Copy the same function calls created in Step 11 to the "Stage2" and "Stage3" cases. All three stages are set to read the current position of the encoder, evaluate when the stage is complete, and then transition to the next stage.
15. Next is the "Alarm" case. In the event that an Alarm is triggered (**amplitude > alarmThres** or "Stage3" is completed) an alarm sound will occur and the code transitions to the "Shutdown" case. The digitized sound is already implemented in the "LVRIOEval\_C\_Exercise3.h" file. You will need to send this signal to the FPGA and set the Gain (volume) appropriately. This is completed using the FPGA Write U32 function for the Gain and FPGA FIFO Write I32.



The signatures are:

### NiFpga\_WriteU32

```
NiFpga_Status NiFpga_WriteU32(NiFpga_Session session, uint32_t control, uint32_t value)
```

Writes an unsigned 32-bit integer value to a given control or indicator.

### NiFpga\_WriteFifoI32

```
NiFpga_Status NiFpga_WriteFifoI32(NiFpga_Session session, uint32_t fifo, const int32_t* data, size_t  
numberOfElements, uint32_t timeout, size_t* emptyElementsRemaining)
```

Writes to a host-to-target FIFO of signed 32-bit integers.



16. Use a Gain value of 5000 for the U32 Write. Use variables **boingsound**, **boingsoundsize**, **audioFIFOremaining**, and a 5000 ms timeout for the FIFO I32 Write. The result should look the following for the “Alarm” case.

```
case Alarm:
    printf("Alarm!\n");

    NIIFpga_MergeStatus(&status, NIIFpga_WriteU32(session,
        NIIFpga_MachineInspectFPGAC_ControlU32_Gain,
        5000));

    NIIFpga_MergeStatus(&status, NIIFpga_WriteFifoI32(session,
        NIIFpga_MachineInspectFPGAC_HostToTargetFifoI32_AudioFIFO,
        boingsound,
        boingsoundsize,
        5000,
        &audioFIFOremaining));

    usleep(2000000);

    state = Shutdown;
    break;
```

**Note:** `usleep()` for 2 seconds is used to allow time for the sound to play.

17. The last portion of code to create is the “Shutdown” case. This case will reset the encoder position, turn off the PWM output, and stop the state machine. This will require two FPGA calls, Write Boolean PWM and Write I16 Position. The call signatures are,

#### **NIIFpga\_WriteBool**

`NIIFpga_Status NIIFpga_WriteBool(NIIFpga_Session session, uint32_t control, NIIFpga_Bool value)`

Writes a boolean value to a given control or indicator.

#### **NIIFpga\_WriteI16**

`NIIFpga_Status NIIFpga_WriteI16(NIIFpga_Session session, uint32_t control, int16_t value)`

Writes a signed 16-bit integer value to a given control or indicator.

Use the **positionCurrent** and **smstop** variables to complete these calls. The resulting “Shutdown” case should look similar to the following,

```
case Shutdown:
    positionCurrent = 0;
    smstop = true;

    NIIFpga_MergeStatus(&status, NIIFpga_WriteBool(session,
        NIIFpga_MachineInspectFPGAC_ControlBool_PWM,
        false));

    NIIFpga_MergeStatus(&status, NIIFpga_WriteI16(session,
        NIIFpga_MachineInspectFPGAC_IndicatorI16_Position,
        positionCurrent));


    break;
```

With the “Shutdown” case complete the application is ready to be compiled and executed on the LabVIEW RIO Evaluation Kit.

## Test the Real-Time Application

### *Application Testing*

1. Run the **LVRIOEvalEx3\_DeployConfig** Run Configuration you created previously in Section 1 Step 26.
2. As the program deploys and begins to execute you should see execution outputs ("printf()" statements in the code) in the Console Tab at the bottom of Eclipse.

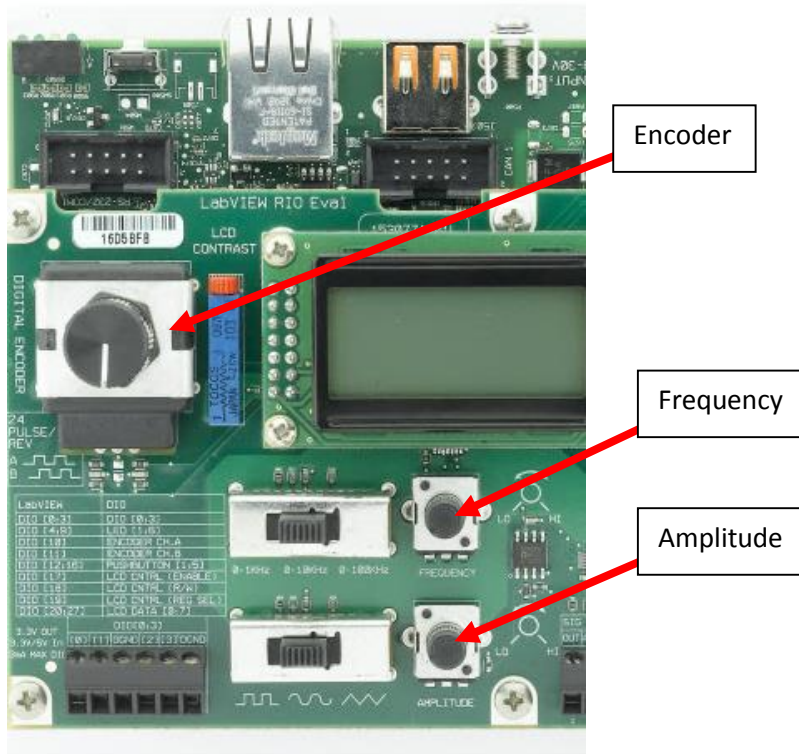


```
<terminated> LVRIOEvalEx3_DeployConfig [C/C++ Remote Application]
admin@sbRIO-9637:~# echo $PWD'>'
/home/admin>
admin@sbRIO-9637:~#
admin@sbRIO-9637:~# /home/admin/LVRIOEvalProject/LVRIOEvalEx3_DeployConfig
Opening FPGA Session and running bitfile...
Starting main state machine loop...
```

**Note:** If you receive an error during deployment check to make sure that the IP address is correct and try again. If the target gets into a state where you can no longer deploy to it, try restarting the LabVIEW RIO Evaluation Kit, by pressing the button next to the Ethernet port, and trying again.

3. Wait for the "Starting main state machine loop..." message to appear on the console before proceeding to the the next step.

4. Turning the Encoder to the right should progress through the three stages and turning the frequency knob will change your waveform, and turning the amplitude knob will allow you to test your alarm logic.



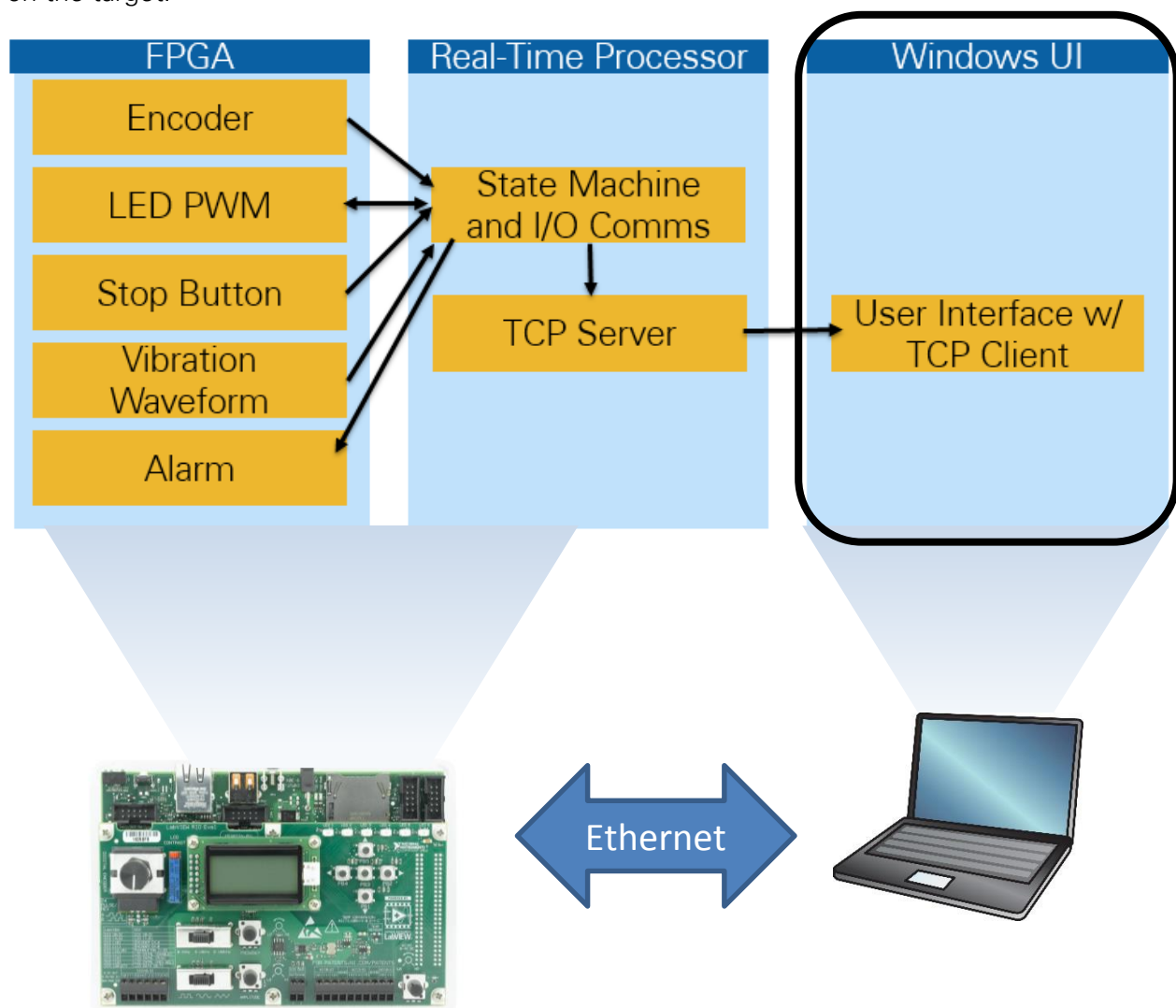
5. When you are finished testing the real-time application, press the E-Stop button or run through all three stages to exit the application normally.
6. Save all of the project files by selecting **File»Save All** in the Project Explorer window.

## Exercise 4 | Run with a Windows User Interface

### Summary

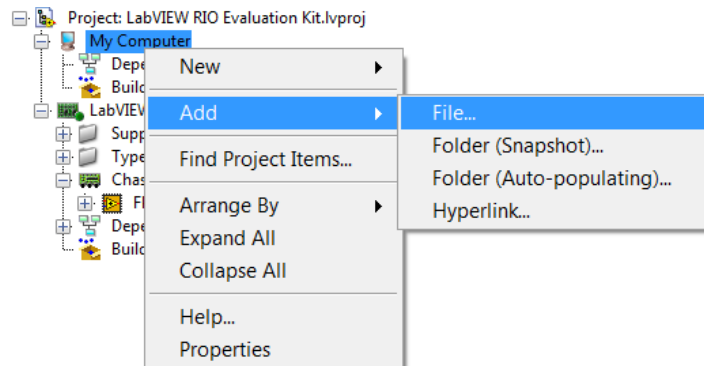
In this exercise you will finish the overall embedded system by connecting to a user interface running on your Windows development machine over TCP/IP. The LabVIEW user interface VI has already been completed for you, and you reviewed the TCPServer function running on the kit in `LVRIOEval_C_Exercise3.cpp`.

The evaluation kit does not have an integrated GPU or display port, so it is meant for true headless operation. The final UI can be as simple as the LCD, to a LabVIEW VI running on a Windows PC (Exercise 4), to a thin client reading data in a browser from a web service running on the target.

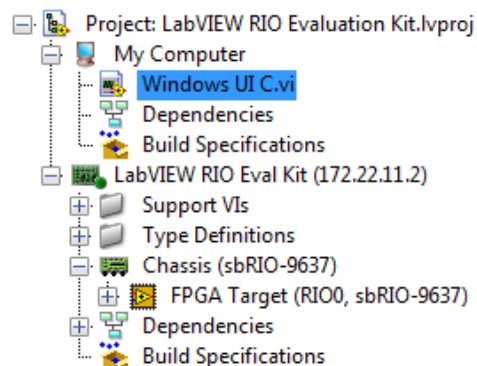


## Implementation

1. Open **LabVIEW RIO Evaluation Kit.lvproj** from the `.\2- Create FPGA Application` directory that you worked on in Exercise 2.
2. To add the Windows User Interface code, in the LabVIEW Project Explorer window, right-click on *My Computer* and select **Add»File...** Navigate to the `.\4- Create Windows UI` folder and select **Windows UI C.vi**.

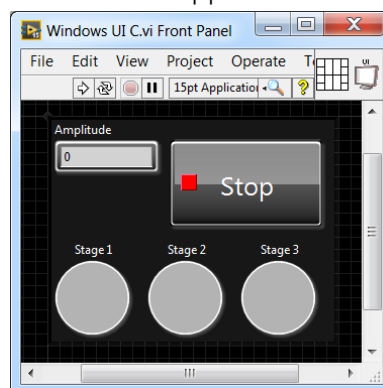


3. Your project hierarchy should now look like this:

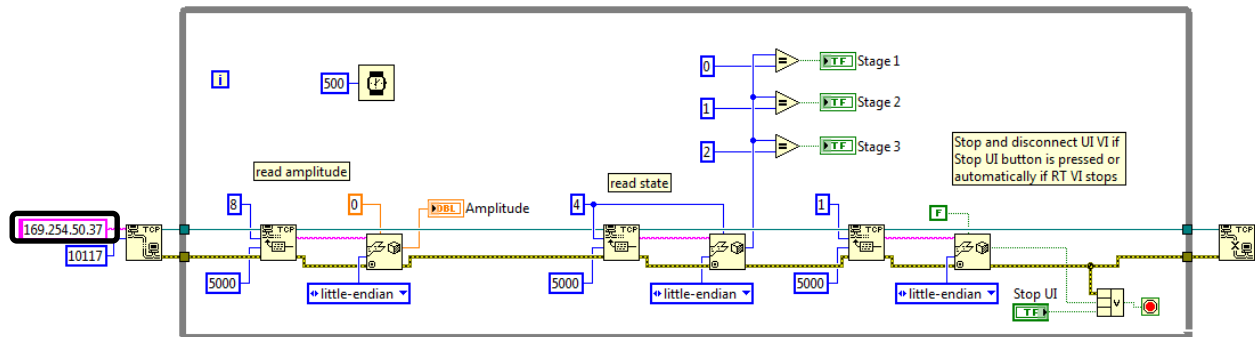


## Explore the Windows Application User Interface

4. Open **Windows UI C.vi** and notice the Front Panel User Interface is simple, but has been customized. It will show the current stage and amplitude value, and provides a stop button to disconnect the UI from the application running on your target.



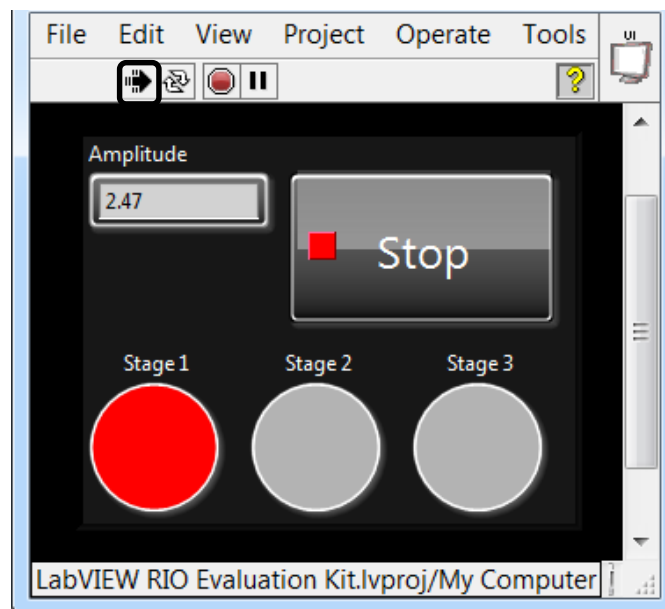
5. Press <Ctrl-E> to open the block diagram and review the code.



- The connection is made by pointing the VI to your target's IP Address and opening a connection on port 10117.
- Update that IP address with your device's IP address, as shown above.
- Then the VI reads Amplitude, State, and Stop data from the TCP server running on the target respectively. The data must be unflattened from a string in each instance.
- There is some logic to light the appropriate LED based on what Stage number the target is executing.

## Run and Verify the Completed System

1. First run the Eclipse Run Configuration in **LVRIOEvalEx3\_DeployConfig** you created previously in Exercise 3.
2. Then, click the **Run** arrow on the **Windows UI C.vi**, otherwise the client will not be able to find the server. This code was designed for only one client.



- ✓ As you operate the part inspection machine to progress through the three stages, note how the Amplitude and LED indicators on the Windows UI front panel update.
- ✓ Check that when the RT application stops, the Windows UI VI should as well.
- ✓ While the RT application is running, press the Stop button on the Windows UI front panel and ensure your RT application keeps running. Restart your Windows UI VI and ensure it reconnects with updated data.

## Exercise 5 | Startup Application

### Summary

Now that you have completed the development of your embedded system, you may want to deploy the application to run at startup. Startup scripts tell the OS on the target which applications to load during the startup process, in what order, and at what default run level. This exercise will have you add a new startup script on NI Linux Real-Time.

### Implementation

This tutorial is hosted online the NI Community NI Linux Real-Time Forum, [Installing Startup Scripts on NI Linux Real-Time](#). This [community](#) is the recommended next step to find tutorials for NI Linux Real-Time, look for supported packages on the [NI Repository](#), and ask questions on our active [discussion forum](#).

- Remember your application is on the target in the `/home/admin/LVRIOEvalProject` directory.
- You can create the startup script using the editor of your choice. You also can create the startup script on your host computer, and then transfer it to your RT target. To save time, the online tutorial includes a sample startup script [here on ni.com](#).

Once you have tested and confirmed the startup script functions and installed successfully, allow a minute for the RTOS to boot up. Once it does successfully reboot, run through the system tests from the *Run and Verify the Completed System* section of Exercise 4.

**Congratulations!** You have now completed the development of a LabVIEW RIO-based embedded system with three targets, User Interface on your Windows PC, State Machine running on the NI Linux Real-Time distribution on the processor, and I/O control and monitoring on the FPGA.

## Next Steps

This evaluation tutorial was an introduction to the LabVIEW RIO architecture. Before you purchase NI LabVIEW and a RIO hardware device to start programming your application, please review the following:

### 1. LabVIEW RIO Architecture Training Path

Since this was a brief introduction to the LabVIEW Real-Time and LabVIEW FPGA modules it is highly recommended that you better understand what further knowledge you need to gain before you start creating your own system.

**Appendix A** has a guide to help you identify a training path to gain the appropriate skill level for the task you are trying to complete using LabVIEW.

### 2. RIO Hardware Form Factors

In this evaluation kit you used a board-level form factor of the RIO hardware platform. This however is just one form factor of many different families of RIO hardware products that can all be similarly programmed with the LabVIEW FPGA and LabVIEW Real-Time modules.



Learn more about the other families by visiting [ni.com/embedded-systems](http://ni.com/embedded-systems).

### 3. Online Community with Challenge Exercises and Resources

To find getting started resources, more advanced tutorials specifically for the LabVIEW RIO Evaluation Kit, user applications, discussion forums, and to learn more about the LabVIEW RIO architecture, visit [ni.com/rioeval/nextstep](http://ni.com/rioeval/nextstep) and [ni.com/linuxrtforum](http://ni.com/linuxrtforum).



# Appendix A | LabVIEW RIO Training Path

## Maximize Your RIO Investment

## Develop Faster and Reduce Maintenance Costs

For developing embedded control and monitoring systems, the combination of NI LabVIEW software and NI CompactRIO, R Series, or NI Single-Board RIO hardware offers powerful benefits including the following:

- Precision and accuracy - Precise, high-speed timing and control combined with accurate measurements
- Flexibility – Hundreds of I/O modules for sensors, actuators and networks that with LabVIEW can connect quickly to control and processing algorithms and system models
- Productivity – LabVIEW system design software for programming processors, FPGAs, I/O and communications
- Quality & ruggedness – High-quality hardware and software for deploying reliable embedded systems that last

However, there is still a learning curve to effectively take advantage of these benefits, and your application or job in part determines the size of that curve. Every project is different. To be successful, you should determine up front what you need to learn to deliver a system that meets or exceeds requirements while also minimizing development time. If the requirements for your next project differ significantly from your current one, assess what additional concepts you should learn to successfully complete it. For example, you may be currently developing a functional prototype and just want a system that works, but if the design is approved you will likely want something that is built to last and minimizes long-term maintenance costs. Consider the different capabilities needed for each stage of developing an application based on CompactRIO, R Series, or NI Single-Board RIO, and take advantage of resources that can help you efficiently learn those necessary skills.

## Core Capabilities Required for all CompactRIO and NI Single-Board RIO Users

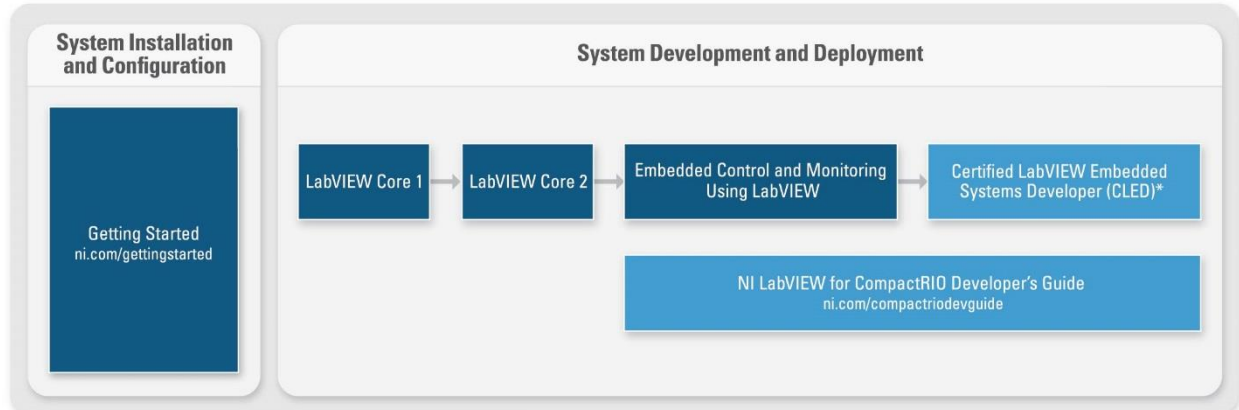
To begin with, everyone who uses LabVIEW and CompactRIO, R Series, or NI Single-Board RIO should have the ability to

- Install and configure RIO hardware and LabVIEW software
- Create a diagram or architecture for your system
- Navigate the LabVIEW environment
- Apply key LabVIEW structures (While Loops, clusters, arrays, and so on)
- Develop basic, functional applications in LabVIEW
- Apply common design patterns (state machine, producer/consumer, and so on)
- Understand the difference between Windows and real-time operating systems
- Implement communication between processes
- Deploy an application

To help you learn these abilities, National Instruments recommends the following resources:

- Getting Started With NI Products ([ni.com/gettingstarted](http://ni.com/gettingstarted))
- LabVIEW Core 1 - Core 3 and Embedded Control and Monitoring using NI LabVIEW classroom training courses ([ni.com/training](http://ni.com/training))
- LabVIEW Core 1, LabVIEW Core 2, LabVIEW Real-Time, and LabVIEW FPGA Self-Paced Online Training, free if you have an active service contract ([ni.com/self-paced-training](http://ni.com/self-paced-training))
- LabVIEW for CompactRIO Developer's Guide ([ni.com/compactriodevguide](http://ni.com/compactriodevguide))

## CompactRIO/Single-Board RIO Recommended Training Path



\* A CLD or higher is required before attempting the CLED exam

## Need More Help?

Contact a National Instruments Training & Certification Specialist at [ni.com/contact](http://ni.com/contact) for additional guidance on the level of skill you need for your application.

## No Time to Learn?

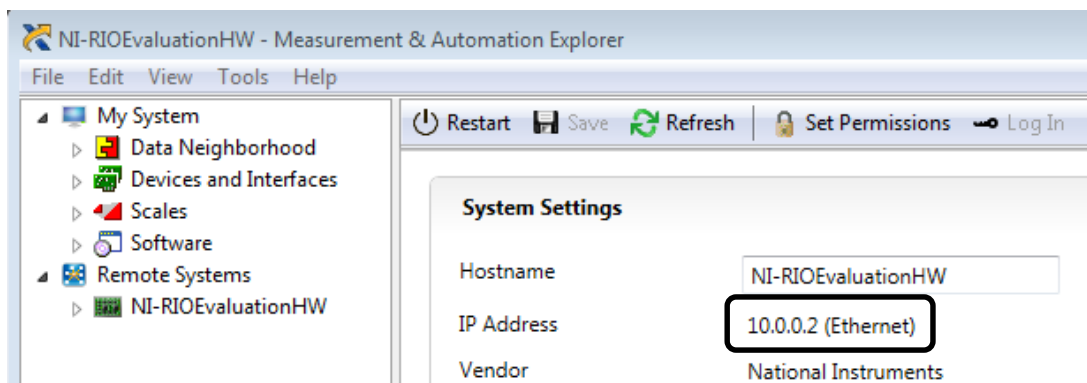
Many National Instruments Alliance Partners have already invested in the level of proficiency required for your application. If you have a CompactRIO or NI Single-Board RIO project that requires a greater skill level than you currently have and you are unable to gain the required level in the time allotted for your project, NI can temporarily augment your expertise by connecting you with an Alliance Partner that can provide consulting services while you get up to speed. Find an Alliance Partner in your area at [ni.com/alliance](http://ni.com/alliance).

## Appendix B | Changing the IP Address in the LabVIEW Project

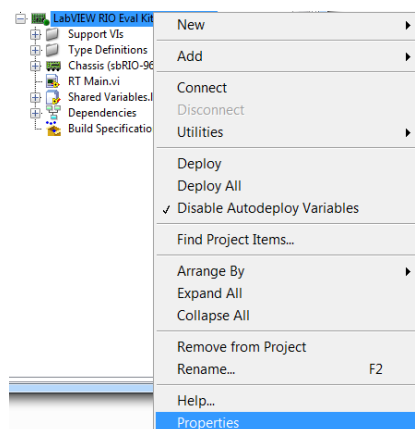
Your LabVIEW RIO Eval Kit is identified by its IP address. For each exercise, confirm that the IP address in the project matches the IP address of your RIO device. The National Instruments LabVIEW RIO Evaluation Setup utility should have prompted you to write down the target's IP address, but you also can locate the device through the following steps.

If you are connected over the USB Host-to-Host cable, your IP Address is **172.22.11.2**.

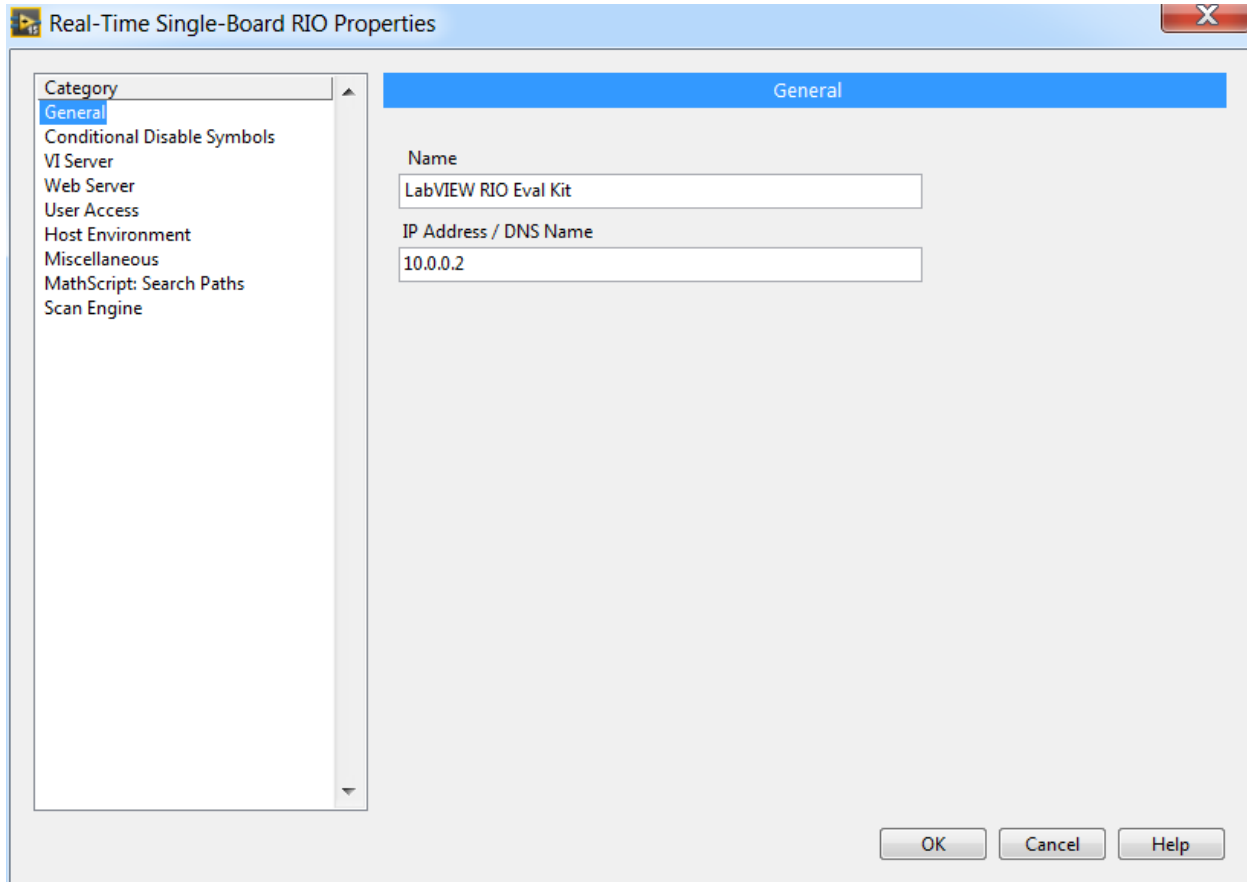
1. If you are connected over Ethernet, determine the IP address of your device by opening NI MAX (Measurement & Automation Explorer) located at **Start»All Programs»National Instruments»NI MAX**.
2. Click the triangle next to Remote Systems.
3. Click on your device in the Remote Systems tree and on the right hand side note the IP address that appears in the System Settings tab.



4. Change the IP address of the target in the LabVIEW Project Explorer window to match the IP address of your evaluation board.
  - a. Right-click the *LabVIEW RIO Eval Kit* target in the LabVIEW Project Explorer window and select **Properties** from the menu to display the General properties page.



- b. In the **IP Address / DNS Name** box, enter the IP address you wrote down from the National Instruments LabVIEW RIO Evaluation Kit Setup utility or just now from Measurement & Automation Explorer and click **OK**.



- c. Right-click on the *LabVIEW RIO Eval Kit* target in the Project Explorer window and select **Connect** to verify connection to the evaluation device.

If you cannot find your target in NI MAX, use the Remote System Discovery Utility or contact [NI Support](#).

