LabVIEW Real-Time Course Manual

November 2001 Edition

Part Number 323226A-01

© 2001 National Instruments Corporation. All Rights Reserved.
Copyright

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

Trademarks

DataSocket™, FieldPoint™, LabVIEW™, National Instruments™, NI-CAN™, ni.com™, NI-DAQ™, NI-Motion™, and NI-VISA™ are trademarks of National Instruments Corporation.

Product and company names mentioned herein are trademarks or trade names of their respective companies.
# Table of Contents

Preface ................................................................................................................................. 1
Lesson 1 LabVIEW Real-Time Basics .................................................................................. 1-1
Exercise 1-1 ......................................................................................................................... 1-23
Exercise 1-2 ......................................................................................................................... 1-23
Exercise 1-3 ......................................................................................................................... 1-24
Lesson 2 Communication with RT Series Hardware ............................................................ 2-1
Exercise 2-1 ......................................................................................................................... 2-42
Exercise 2-2 ......................................................................................................................... 2-46
Exercise 2-3 ......................................................................................................................... 2-47
Exercise 2-4 ......................................................................................................................... 2-48
Exercise 2-5 ......................................................................................................................... 2-49
Exercise 2-6 ......................................................................................................................... 2-49
Lesson 3 Multithreading and Priorities ............................................................................... 3-1
Exercise 3-1 ......................................................................................................................... 3-23
Exercise 3-2 ......................................................................................................................... 3-23
Exercise 3-3 ......................................................................................................................... 3-26
Exercise 3-4 ......................................................................................................................... 3-28
Lesson 4 Timing and Sleep ............................................................................................... 4-1
Exercise 4-1 ......................................................................................................................... 4-27
Exercise 4-2 ......................................................................................................................... 4-28
Exercise 4-3 ......................................................................................................................... 4-29
Lesson 5 Shared Resources ............................................................................................... 5-1
Exercise 5-1 ......................................................................................................................... 5-24
Exercise 5-2 ......................................................................................................................... 5-27
Exercise 5-3 ......................................................................................................................... 5-30
Lesson 6 Other Features of LabVIEW Real-Time ............................................................... 6-1
Exercise 6-1 ......................................................................................................................... 6-24
Exercise 6-2 ......................................................................................................................... 6-28
Exercise 6-3 ......................................................................................................................... 6-29
Exercise 6-4 ......................................................................................................................... 6-30
Exercise 6-5 ......................................................................................................................... 6-32
Course Goals

- Understand concepts of real-time and determinism
- Learn how to configure and communicate with real-time hardware
- Understand memory usage in LabVIEW Real-Time
- Understand how multithreading and priorities work in LabVIEW Real-Time
- Understand shared resources
- Learn how to optimize real-time tasks
- Learn about other real-time features

As a prerequisite to this course, you should have at least taken the LabVIEW Basics I course.
Course Non-Goals

- To teach LabVIEW basics
- To teach control theory
- To teach PID and Fuzzy Logic theory
- Analog-to-digital (A/D) theory
- The operation of GPIB, RS-232, Motion, CAN, VISA
- To discuss every built-in LabVIEW object, function, or library VI
- The development of a complete application for any student in the class
Equipment Needed for This Course

- Host PC running Windows 95 or newer
- LabVIEW Real-Time version 6i or newer
- LabVIEW Real-Time Course Manual and disks
- PXI-MIO and PCI/PXI-7030 MIO devices
- PXI-8156B, 8170, 8145, or 8146 controller
- DAQ signal accessory
- FP-2000 module

To complete the exercises in this course, one of the following sets of hardware is required:

- PXI controller (8156B, 8170 with ethernet module 8211, 8145, 8146)
- PXI-MIO board
- PXI chassis
- ethernet or crossover cable
- host PC

OR

- PCI/PXI RT Series DAQ board

OR

- host PC
- ethernet or crossover cable
- FP-2000

Some exercises are hardware specific, so you may not be able to complete all the exercises with your hardware set.
Hands-On Exercises

- Exercises reinforce the topics presented
- Each exercise shows a finished VI after it is run
- Descriptions of how to build the VI follow the picture
- Save exercises in personal folder on the Windows desktop
Course Outline

- Real-time concepts and definitions
- Real-time OS and real-time targets
- Communicating with the real-time targets
- Multithreading and priorities
- Shared resources
- Hardware timing
- Control loop optimization
- Watchdogs
- Application Builder
This course will focus on LabVIEW Real-Time and DAQ. Although Serial, GPIB, and VISA are supported in LabVIEW Real-Time 6, they will not be discussed. To learn how to program with these protocols in LabVIEW, take the necessary courses.
Lesson 1

LabVIEW Real-Time Basics

You Will Learn:
A. Real-time definitions and concepts
B. About the real-time targets
C. About the real-time OS
D. About differences between LabVIEW and LabVIEW Real-Time
What Is Real-Time?

Real-time response is the ability to reliably, without fail, respond to an event, or perform an operation, within a guaranteed time period.

Before delving into how to use LabVIEW Real-Time, it is first important to understand some technical terms related to real-time systems. The first term to understand is what is meant by **real-time**. A misconception is that it means really fast. Real-time is probably more synonymous to **in-time**. In other words, a real-time system ensures that responses will occur in time, or on time. With non-real-time systems, you do not have any way to ensure that your response will occur within any given time period, and calculations may finish much later or earlier than you expect them to.

In order 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 will behave with real-time characteristics. The program may rely on something that does not behave in real-time, which then causes the program to not behave in real-time.
What Is the Goal of Real-Time?

Deterministic loops with I/O (the focus of this course will be I/O using NI-DAQ)

Time critical Control Algorithm or other Real-Time code
The first term to understand is determinism. **Determinism** is a characteristic of a system that describes how reliably it can respond to external events, or perform operations, within a given time limit. High determinism is a characteristic of real-time systems, which guarantees that your calculations and operations will occur within a given time. Deterministic systems are predictable. This is an important characteristic for applications such as control. In a control application, the control program measures inputs, makes calculations based on the inputs, and then outputs values that are a result of those calculations. Real-time systems can guarantee that the calculations will finish on time, all of the time.

Many applications that require a real-time operating system are cyclic, such as a control application. The time between the start of each cycle, T, is called the **loop cycle time (or sample period)**. 1/T is the loop rate or sample rate. Even with real-time operating systems, the loop cycle time can vary between cycles. The amount of time that the loop cycle time varies from the desired time is called **jitter**. The maximum amount that a loop cycle time varies from the desired loop cycle time is called **maximum jitter**.
All real-time systems have some jitter, it is just that the jitter is much smaller than in non-real-time systems, sometimes on the order of nanoseconds. Non-real-time systems have very large or unbounded maximum jitter.
LabVIEW applications running in Windows are not guaranteed to run in real-time. This is because Windows is not a real-time operating system. Windows cannot give you the guarantee that your code will always finish within given time limits. The time your code takes to execute in Windows is dependant on many factors, such as other programs running, like screen saver or virus software. Windows also has to service interrupts (from devices such as a USB port, keyboard, mouse, and other peripherals), which can delay execution of your code. You can increase your chances of having programs run deterministically in Windows by disabling all other programs such as screen savers, disk utilities, and virus software. You can further increase determinism by disabling drivers for devices with interrupts such as the keyboard, mouse, and Ethernet card. And finally, for better determinism, you can write a device driver in Windows to get the most direct access to hardware possible. Even after all these attempts to increase determinism, there is still no guarantee that your code will execute with real-time behavior all of the time. This is because Windows can preempt your LabVIEW programs, even if you use time-critical priority. Priority will be discussed later.

Fortunately, with LabVIEW Real-Time, your programs run in a separate Real-Time operating system, so you do not need to disable programs or write device drivers in order to achieve real-time performance.
Real-Time DAQ and Windows

Buffered analog I/O can occur in real-time with Windows (output does not depend on input)
- DMA
- Hardware clock

If all you need to do is real-time data acquisition, then you do not necessarily need LabVIEW Real-Time. National Instruments has many data acquisition boards that can acquire data in real-time even though they are controlled by programs running in Windows. This is done using an onboard hardware clock that makes sure that data is acquired at constant rates. This data is automatically transferred directly to memory using DMA or interrupt service requests (IRQ).
What Is LabVIEW Real-Time (RT)?

An extension of LabVIEW that enables easy development of real-time applications that run on National Instruments RT Series Hardware

- LabVIEW Real-Time 5.1.2 was a full installation that included all of the functionality of LabVIEW 5.1.1.
- LabVIEW Real-Time 6i is a module that installs on top of LabVIEW 6i.
Exercise 1.1

Students become familiar with the documentation available with LabVIEW Real-Time.

Time to complete: 10 minutes
Real-Time System Overview

Components required:
- Windows PC (Host PC)
- LabVIEW Real-Time Development System (on Host PC)
- National Instruments Real-Time Series Hardware
  - Separate CPU and memory
  - Separate real-time kernel (OS)
- Cables and Accessories

Same LabVIEW programming techniques

All you need to create a real-time system is the following:
- Computer running Windows 2000/NT/9x.
- LabVIEW Real-Time Software
- National Instruments Real-Time Series Hardware
- Cables and accessories

A programmer who already knows how to program using LabVIEW does not have much more to learn in order to create real-time applications. There is very little difference between programming in LabVIEW for Windows and LabVIEW Real-Time. In fact, many existing LabVIEW for Windows programs can be automatically downloaded and run on National Instruments Real-Time Series Hardware with little or no modification. However, to get optimal performance, good programming techniques, as discussed in this course, are required.
LabVIEW Real-Time applications are developed in the Windows environment on a computer called the **Host PC**, and are then downloaded to National Instruments Real-Time hardware, where they run in real-time, independent of the Windows OS. There are several kinds of National Instruments real-time hardware platforms upon which real-time applications can run. These platforms are also referred to as targets, or target platforms. The software that runs on the target platforms is referred to as the LabVIEW RT Engine, or just the **RT Engine**. The RT Engine is the software that runs in the real-time operating system (**RTOS**) on RT Series Hardware.
Why LabVIEW Real-Time?

- Programs run in a separate real-time operating system on RT Series Hardware.
- Real-Time operating system only runs the RT Engine and no other programs for high determinism. *
- Less software components results in reliability and stability.

* Programs related to the RT Engine run at normal priority such as the FTP Server.

There are many applications for which LabVIEW Real-Time is needed instead of LabVIEW for Windows. These are applications that require programs to run with a level of determinism unattainable with Windows.

One advantage is that when an RT application is running on RT Series hardware, if the Host PC crashes, the RT Engine continues to run. Also, since all that is running on the RT Series Hardware is the LabVIEW RT Engine, it is much more stable than Windows, which combines software and drivers from many vendors.

Even though your programs are running in a real-time operating system, it does not mean that they are not susceptible to poor performance due to poor programming techniques. A well-programmed loop will have higher loop rates and less jitter than a poorly programmed loop. There are cases where poor programming will cause your program to crash. For instance, if you make a program that continues to use more and more memory, as soon as all of the memory is used up, the program will crash (no virtual memory is used in the RTOS because it is not deterministic). Also, if you make your program execution dependant upon non-deterministic external processes, such as communication (i.e., GPIB, Serial, or Ethernet), or File I/O, then your program is not guaranteed to run in real-time.
One type of target or platform is a device that plugs into a PCI or PXI slot of a computer running Windows. It is referred to as an RT Series DAQ Device. Another type of platform is the RT Series PXI Controller. There is also a FieldPoint module that can run the RT Engine.
RT Series DAQ Devices contain a **processor board** and an attached **daughtboard** for I/O. The processor board is a self-contained computer on one board, which runs the RT Engine. RT Series DAQ Devices communicate with the Host PC using shared memory located on the RT Series DAQ Device.

Currently, three RT Series DAQ Devices are available:

- PXI/PCI-7030/6030
- PXI/PCI-7030/6040
- PXI/PCI-7030/6533
The RT Series PXI Controllers come with the RT Engine preinstalled. During development, RT Series PXI Controllers communicate with the Host PC through an Ethernet interface. Since RT Series PXI Controllers are controlled remotely, they are considered headless systems. A headless system has no keyboard, monitor, or mouse.

As of September 25, 2001, these are the RT Series PXI Controllers available:

- PXI-8156B
- PXI-8170 (requires a separate Network module)
- PXI-8145
- PXI-8146
The FP-20XX has many features that enable it to act as a stand-alone device. Since most stand-alone devices reside in remote or inaccessible locations, it is important for these systems to be reliable to eliminate the need for constant maintenance. The dedicated processor in the FP-20XX runs a real-time operating system which executes the LabVIEW Real-Time application deterministically. The reliability of the real-time operating system also reduces the chances of system malfunction.

The FP-20XX includes a Watchdog timer that monitors the overall operation of the FieldPoint system. When the watchdog senses that the system has a malfunction, it can run a separate maintenance routine or even reboot the FieldPoint system. The operation of the watchdog is completely user-defined through a simple and intuitive protocol.

The FP-20XX also includes 5 DIP switches, 3 tri-color LEDs, and 1 bi-color LED that can be used for user interaction as a stand-alone system. For instance, the LEDs can be used to signify certain states of operation while the dip switches can be configured to specify the startup state of the FieldPoint system.

Switches 1, 2, 3, 4, and 5 are user-accessible DIP switches. They may be read by the FieldPoint LabVIEW VIs, but have no default functionality. The DISABLE VI, SAFE MODE, and RESET DIP switches are read only when you power up or reboot the FP-20XX network module. You must reboot the module with one of these switches ON for its setting to take effect. If you reboot the system with more than one of these switches ON, only the left-most switch is read, and the others are ignored.

The onboard static memory consists of 3 MB of Flash memory for the FP-2000 and 11 MB of Flash memory for the FP-2010. The Flash memory acts as a hard drive for the system and can be used to store startup applications or data. For instance, you can configure the FP-2000 to monitor the temperature inside an environmental chamber while logging that data to the Flash memory. The FP-2010 is ideal for large data-logging applications with the larger 11 MB of Flash memory. If you need to stream to disk larger memory chunks, you can transfer data back to the host program, and stream it to the host PC hard drive.
The FP-20XX includes a second set of screw terminals to connect a backup power supply. The system will automatically switch to this second power supply, when it senses a problem with the first power supply.

The serial port allows the FP-20XX to connect to existing systems. In addition, it can be used to connect other serial devices such as a serial LED which can be used to display information for the operator.

The FieldPoint I/O modules are compact modules ideal for usage in industrial environments, where space is limited, or where placing a standard desktop computer is not an option. The dimensions of the module are 4.3 in. wide, 4.3 in. deep, and 3.6 in. high. The systems can be din-rail mounted to increase space on the factory floor, and be placed in locations with stringent temperature demands. A FieldPoint 2000 based system can operate efficiently in environments from –25 to 55 ºC.

With LabVIEW Real-Time downloading the code into the FP-20XX, you can create a stand-alone system. For instance, you can create a custom data-logging application that acquires the data from the FieldPoint I/O module and logs the data to the local Flash memory. With over 100 analysis routines such as digital filters, spectrum analysis, signal processing, and curve fitting in LabVIEW, you can include custom in-line data analysis on the FP-20XX node. The in-line analysis can convert the raw data into more effective information by determining and logging the peak frequency of a signal which also reduces the amount of information stored on the local Flash memory.

You can also create a stand-alone control application that continually monitors and responds to a set of stimuli. For instance, this system can be used to monitor and control the pressure in an environmental chamber. With built-in PID, fuzzy logic, and other control algorithms, you can build a custom control routine for the system. With this custom control routine running in the stand-alone control node, the FP-20XX has device level intelligence that doesn’t rely on a main supervisory control station to make decisions for it. In addition to being more reliable, this improves the performance of the system with a real-time system responding deterministically to stimuli.
LabVIEW RT Menu Items and VIs

- Additional Real-Time menu items
  - File»Exit without closing RT Engine VIs
  - Operate»Download Application
  - Operate»RT Engine Info
  - Operate»Switch Execution Target…
  - Tools»RT Target Options…
  - Help»View Printed Real-Time Manuals…
  - Help»LabVIEW Real-Time Help…

- Just a few RT-specific VIs
New features have been added to Measurement & Automation Explorer (MAX). Functionality that was previously found in Remote System Explorer (used with LabVIEW Real-Time 5.1.2) is now found in MAX. In the Configuration window, you will find an entry called Remote Systems. It is here that you configure RT Series Devices on the network. The use of MAX will be discussed later.

Fieldpoint Explorer 3.0 is used to configure Fieldpoint 2000 Series modules. See the user manual for details.
LabVIEW RT Examples Directory

\Examples\RT\n
Choose the VI to open:

Look in: 

- RT Communication.lib
- RT Control (7030).lib
- RT Control.lib
- RT Tutorial.lib
- RT Watchdog (PXI-8156E).lib

File name: 

Files of type: VI's & Controls (*.vi, *.ctl, *.crt, *.ctl)

Open Cancel
Exercises 1.2 and 1.3

- Students become familiar with the documentation available with MAX.
- Students become familiar with the examples specific to LabVIEW Real-Time.

Time to complete: 10 minutes
Chapter 1—Summary

- LabVIEW Real-Time for Determinism
- Development in Windows similar to LabVIEW
- Code downloaded to RT Engine
- RT Engine runs in RTOS on RT Series Hardware
  - RT Series DAQ Devices
  - RT Series PXI Controllers
  - FP-2000 Series
Exercise 1-1
Objective: To become familiar with the documentation available with LabVIEW Real-Time.

1. Launch LabVIEW Real-Time by selecting Start»Programs»National Instruments»LabVIEW 6»LabVIEW. (If a window appears titled Please select target platform, select Host PC (LabVIEW for Windows), and click OK.)
2. Click on the New VI button.
3. From the new VI, select Help»View Printed Real-Time Manuals (be careful that you don’t accidentally select View Printed Manuals…).
4. Take about 5 minutes to browse these documents related to LabVIEW Real-Time:
   - LabVIEW Real-Time Release Notes
   - LabVIEW Real-Time User Manual
   - RT Series DAQ Device User Manual
   - RT Series PXI/CompactPCI Controller User Manual
5. From the new VI, select Help»LabVIEW Real-Time Help…
6. Take a few minutes to browse these help files.
7. Exit LabVIEW.

Exercise 1-2
Objective: To become familiar with the documentation available with MAX.

2. If the User Preferences Dialog Box appears, select Every time I launch MAX, and uncheck the Show this dialog the next time I launch MAX box.
4. Take a few minutes to browse these help files.
5. Close the help window and MAX.
Exercise 1-3
Objective: To become familiar with the examples specific to LabVIEW Real-Time.

1. Launch LabVIEW.
2. Click on the Open VI button and browse to the c:\Program Files\National Instruments\LabVIEW 6\Examples\RT directory.
3. Open each *.llb file and browse the contents of each.

   Most examples in these libraries contain two VIs, one that runs on the RT Engine, and a corresponding VI that runs on the Host PC. The RT Engine VI runs in real-time and communicates with the Host PC VI, which provides a User Interface.

4. After browsing for a few minutes, exit LabVIEW.
Lesson 2

Communication with RT Series Hardware

You Will Learn:
A. How to configure, target, and download programs to RT Series Hardware
B. How to communicate with the RT Engine

This lesson will briefly describe how to configure RT Series Hardware, but it will not go into specifics of installation or configuration of either software or hardware. This information is covered in detail in the user manuals. This lesson will also discuss communication methods between the Host PC and the RT Engine.
Configuring 7030s in MAX

- **View** > Refresh, or <F5>
- Right-click on 7030, select Properties…
  - Test Resources
  - You can change Device Number
  - No Test Panels (grayed out)

NI-DAQ software should be installed before installing RT Series DAQ Devices. If your device does not show up in MAX, selecting **View** > Refresh, or pressing <F5> will refresh the Configuration window. If it still does not show up, review and repeat the installation procedure in the **RT Series DAQ Device User Manual**.

The **Test Resources** button will test to make sure that the I/O address, Interrupt, and DMA channels for your RT Series DAQ Device are properly assigned and functional.

Because the processor board is not a DAQ board, it has no test panels.
Configuring Daughterboards

You cannot test resources or run test panels for the DAQ daughterboard because, even though the daughterboard is configured using the Host PC, the daughterboard is actually located on the local PCI bus of the RT Series processor board. Only LabVIEW Real-Time applications targeted to the RT Series DAQ Device can access the DAQ daughterboard. Applications run on the Host PC cannot access the DAQ daughterboard directly, even though the DAQ daughterboard is configured as a separate device. For this reason, the resource configuration on the PCI bus cannot be tested from the Host PC. Note that you do not need to test the resource configuration because National Instruments configures it and it does not change.
Create Virtual Channels for RT Series DAQ Devices from MAX:

- **My System ➔ Data Neighborhood ➔ Create New ➔ Virtual Channel**
- Must reset RT Series DAQ Device for virtual channels to take effect—resetting discussed later

After you create any Virtual Channels, you must reset the RT Series DAQ Device in order for them to take effect. The configuration information is downloaded to the RT Series DAQ Device during the reset.
Configuring RT Series PXI Devices

Only a Host PC on the same subnet can initially configure Network Settings. Configure using the Network Settings Tab, click Apply, and reboot the controller. Select Help»Help Topics»Remote Systems for more information.

If you have an RT Series PXI Controller with the RT Engine installed, if you boot it up for the first time after connecting it to the Network, you can see it from a Host PC on the same subnet with MAX. If you have a non-RT Series PXI Controller, you can create a PXI Boot Disk from MAX (Tools»Remote Systems»RT PXI Disk Utilities»Create PXI Boot Disk) which will allow you to boot the controller into the RT Engine. (Only the PXI-8156B and PXI-8170 are supported at this time.) When the PXI Controller is booted from the PXI Boot Disk for the first time, it creates the ni-rt.ini file (which contains configuration information for the network settings and embedded version of LabVIEW) and ni-rt folder on the c:\ drive. You can then configure the controller using MAX. For details on using non-RT Series PXI Controllers with LabVIEW Real-Time, in MAX select Help»Help Topics»Remote Systems»PXI Specific Information»PXI Controller with Windows Installed.

Configure the Network Settings, click Apply, and reboot the RT Series PXI Controller. See the user manual for details.

Host Name refers to the name that the networked device will be known as on the network.

After an RT Series PXI Controller is configured, you can view it in MAX even if it is not on the same subnet. To do this, right click on Remote Systems, select Create New, then select Remote Device (not on the local subnet), click Next, then enter the IP address of the Remote System, and click on Finish.

Other Functions:

- Reboot—this function allows you to remotely reboot the controller.
- Lock—you can protect the controller by providing a password.
Software for RT Series Devices

- Firmware is software that gets installed on embedded hardware.
- Software for LabVIEW Real-Time consists of:
  - Real-time operating system (RTOS).
  - Embedded version of LabVIEW (the RT Engine).
  - Device drivers.
- RT Series PXI—software is installed on hard drive in `c:\ni-rt` directory.
- RT Series DAQ—software is downloaded from Host PC to memory during reset.

RT Series Networked Devices have firmware preinstalled; however, if it needs to be reinstalled or upgraded, the LabVIEW RT Development System installed on your Host PC contains the firmware images. Firmware can be reinstalled on RT Series Networked Devices over the network using MAX.

The directory structure on RT Series Networked Devices is as follows:

- `c:\` Contains `Ph_exec.exe` (the RTOS). With the PXI-8140 Series, the functionality of this executable is contained in the BIOS.
- `c:\ni-rt\system` Contains the RT Engine and device drivers.
- `c:\ni-rt\startup` Initially empty. Stores applications.
Installing RT PXI Software

To install software, right-click in firmware window, and select **Install Firmware**...

If you wish to refresh the **Remote Systems** section of the Configuration window in MAX, you can select **View»Refresh**, which refreshes the whole configuration window, or you can select **Tools»Remote Systems»Find Remote Systems**, which searches for Remote Systems on the same subnet as your Host PC.

Software images for RT Series Networked Devices are installed on the Host PC in the c:\Program Files\National Instruments\RT Images directory by default. Selecting **Install Firmware**..., as shown above, installs the appropriate software over the network on the RT Series Networked Device.

With LabVIEW 5.1.2, the **Remote System Explorer** is used to install firmware on RT Series Networked Devices.
Remote DAQ Configuration

- From MAX: Tools»NI-DAQ Configuration» Remote DAQ Configuration

- Enter Name or IP address of RT Series PXI Controller

- Clicking Save will send the configuration to the RT Series Controller

To configure the DAQ devices in the RT PXI Chassis, you will need to use **Remote DAQ Configuration**. Through Remote DAQ Configuration, you can also configure SCXI, virtual channels, and scales. When you finish the configuration, the configuration file will be sent to the RT PXI Controller. You then need to reboot the controller for some of the configuration changes to take effect.
To test DAQ devices on an RT Series PXI Controller, map the devices to a Windows PC using MAX, then launch test panels on the RDA device.
Testing PXI DAQ Card Resources

Add Remote PXI Cards to Devices and Interfaces on the Host PC (using RDA) to test them (use Test Panels)

The details of how to do this will be covered in an example.
Installing/Configuring FP-2000

FieldPoint Explorer 3.0 can be used for:

- **Password Protect Resets**
- **Install/Upgrade Software**
- **Reboot Device**
- **View Installed Software**
- **Network Security**
- **Lock/Unlock System Configuration**
- **Device Location**
- **View Error Log**

**Password Protect Resets**—Determines whether the user is allowed to reset the FieldPoint bank remotely. If enabled, the user must provide the password in order to reset the system remotely.

**Install/Upgrade Software**—Use to install or upgrade LabVIEW RT software on the selected FP-20XX.

**Reboot Device**—Use to reboot the selected FP-20XX.

**View Installed Software**—Use to view the LabVIEW RT and driver software versions installed on the selected FP-20XX.

**Network Security**—You can limit host access to the FP-20XX by setting access permissions for different host machines. The default setting allows Read/Write access to all host machines.

**Lock/Unlock System Configuration**—Use to lock or unlock the system configuration with a password.

**Device Location**—Use to change the search location for FP-20XX modules to the local subnet or to a specified IP address.

**View Error Log**—Use to obtain detailed error information if you need to contact National Instruments technical support.
Useful DOS Commands

- `winipcfg` or `ipconfig`
  - Useful for finding IP configuration information for the Host PC

- `ping [IP Address]`
  - Useful for testing network connection

- `ftp [IP Address]`
  - Useful for viewing RT Engine File System
Exercise 2.1

Students become familiar with MAX in relation to configuring RT Series Hardware.

Time to complete: 20 minutes
While developing a real-time application, LabVIEW Real-Time provides a user interface for the RT Engine. This is done by “targeting” LabVIEW Real-Time to the RT Engine. When LabVIEW Real-Time is targeted to the RT Engine, it is considered the RT Development System. While developing RT applications in this manner, to the user, development is done the same way as developing programs for Windows. Only, when the program is run, it actually runs on the RT hardware and not on the windows machine.

To target LabVIEW Real-Time to your RT Series Hardware, start a new VI from LabVIEW Real-Time, then:

1. Select Operate»Switch Execution Target.
2. Select the RT Engine you wish to target. In the target list, you will see an entry for every RT Series DAQ Device configured in MAX. These are designated by DAQ::x RT Engine on PCI-7030, where x is the device number in MAX. You will also see an entry called RT Engine on Network. Select this if you wish to target LabVIEW Real-Time to RT Series Networked Devices such as RT Series PXI Controllers. You will also see an entry called Host PC (LabVIEW for Windows). Although this is actually not an RT Engine, if you select this, then LabVIEW Real-Time will act as regular LabVIEW for Windows.
3. While you are targeted, you will see the RT Engine indicated in the lower left corner of both the front panel and diagram.
Target Selection Options

**For PXI/PCI-7030**
- **Reset Device**—resets, installs firmware (~80 s)

**For Network Devices**
- **Machine Name/IP**—for remote device
- **Password**—needed if remote device is password protected
- **Configure…**—launches MAX

When targeting LabVIEW Real-Time to an RT Engine on the network, such as an RT Series PXI Controller, you must enter the RT Series PXI Controller’s machine name or IP address as previously configured in MAX. If you provided a password while configuring the network device, then you must also provide it here.

When targeting an RT Series DAQ Device (plug in PCI or PXI board), if you are targeting the board for the first time, then “Reset device” will be checked by default. This is because it does not yet contain the software. During the reset, the software is installed, which takes about 80 seconds. Each time the device loses power, it must be reset.
Prompt for Target RT Engine

Tools»Options…»Miscellaneous

You can have the Target Selection Dialog Box open automatically when LabVIEW is started by selecting **Prompt for Target Execution Engine**.

If you wish to have the Target Platform Selection Dialog Box appear automatically when you first start LabVIEW, you can check the option as shown above.
Downloading a VI

- Download VIs by selecting: **Operate»Download Application**

- VIs must be saved first.

- VIs are automatically downloaded when the **Run** button is pushed.

When VIs are opened in the RT Development System, they are not downloaded to the RT Engine until you manually download them.

You can see which VIs have been downloaded by selecting **Browse»Show VI Hierarchy**. An inserted thumbtack indicates a downloaded VI. If the thumbtack is not inserted, the VI is not downloaded.
A VI is in development state when it is open in LabVIEW Real-Time targeted to an RT Engine (LabVIEW Real-Time will be in the RT Development System). What actually gets downloaded to the RT Engine is the compiled code for the VI. When a VI is running in this state, there is a thread (an independent sub-program) running on the RT Engine at normal priority called the User Interface (UI) thread. The RT Engine UI thread exchanges messages with the RT Development System to update the front panel indicators and also read the front panel controls. The communication between the compiled code, the user interface thread, and the Host PC is all transparent to the user. We will discuss threads, multithreading, and priorities later in more detail.

While LabVIEW Real-Time is targeted to an RT Engine, any VI that you open and run will run on the RT Engine. You cannot run any VIs targeted to the Host PC while LabVIEW Real-Time is targeted to an RT Engine. This means that while targeted to an RT Engine, you cannot perform any functions related to the Host PC such as File I/O, or accessing DAQ devices in the Host PC.
Development State (Continued)

- When a VI is run, it is put into Run Mode. To edit a VI after running, select Operate»Change to Edit Mode or press <CTRL+M>.
- All debugging features are supported in the RT Development System (probes, execution highlighting, single stepping, etc.), except for the Call Chain ring.

The Call Chain ring appears when a subVI is paused, and is not supported in the RT Development System. (The Call Chain ring is also referred to in the Real-Time User Manual, as is the Call List Window.)
Disconnecting From an RT Engine

- **File»Exit**
  - Results in this dialog box:

- **File»Exit without closing RT Engine VIs**
  - Automatically closes LabVIEW
  - RT Engine VIs will continue to run

- **Operate»Switch Execution Target**
  - RT Engine VIs will continue to run

- **Deployed State – VIs are in Deployed State after selecting Exit without closing RT Engine VIs or Switch Execution Target**

If, while an RT Engine VI is running in the RT Development System, you exit LabVIEW without closing RT Engine VIs, or Switch Execution Targets, then the RT Engine VI continues to run, but instead of being in the development state, it is in the **deployed state**. The UI thread on the RT Engine is no longer communicating with the RT Development System.
Reconnecting to an RT Engine

- Front Panels not downloaded
- Reconnecting requires copy of VI on Host PC
- Changed or Missing VIs Dialog

When you connect the RT Development System to an RT Engine with VIs already downloaded or running, the RT Development will try to locate and open the corresponding Front Panels on the Host PC. The Front Panel information is not downloaded to the RT Engine and is only contained in the local copies of the VIs on the Host PC. If the copy of the VI on the Host PC has been changed, then the Changed or Missing VIs dialog box will appear. The user has the option of closing the old RT Engine VIs and updating them with the new VIs, which will download the new VIs, or the user can switch to Host LabVIEW without closing the RT Engine VIs. If the VIs are missing, you will first be prompted to find them. If you are unable to find the VIs, click cancel to make the Changed or Missing VIs dialog box appear.

**Tip:** You can FTP the Host PC’s copy of the VI to RT Engines that have storage devices. This way, if the Host PC’s copy of the VI is lost or corrupted, you can retrieve the backup copy from the RT Hardware using FTP.
Exercises 2.2 and 2.3

■ Students use the RT Development System as a user interface.
■ Students use the Changed or Missing VIs dialog box.

Time to complete: 20 minutes
After the application has been developed, the RT Development System can be disconnected from the RT Engine, and the RT Engine can run the application independent of LabVIEW Real-Time on the Host PC. When an RT application is running independent of the Host PC, it is considered to be deployed. LabVIEW Real-Time on the Host PC can then be targeted to the Host PC itself. In this mode, LabVIEW Real-Time on the Host PC is no longer running RT Development System, but is running as regular LabVIEW for Windows. After an RT application is deployed, it can still communicate with programs running in regular LabVIEW on the Host PC.

Since deployed applications running on RT Series Hardware do not have a User Interface provided by the RT Development System, there are many cases in which you want to create a companion application that runs on the Host PC or another PC. Your deployed RT Engine VI can communicate with the companion VI. This companion application can receive data from the RT Engine VI, analyze or log the data received, send data to the RT Engine VI, and provide a way for a user to interact with your Deployed RT Engine VIs. This is also true of applications that run on an RT Series DAQ device that need to have data saved to disk. Since RT Series DAQ Devices (which plug into the PCI bus of the Host PC) have no hard drive, they need to send that data to an application on the Host PC, which can then save the data to the hard drive.

- VIs that run on the RT Series Hardware are referred to as **RT Engine VIs**.
- VIs that run on the Host PC are referred to as **Host PC VIs**.
Deployed State Communication
(Network)

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Speed</th>
<th>Loss?</th>
<th>Ease</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>Fast</td>
<td>Lossless</td>
<td>More Difficult</td>
</tr>
<tr>
<td>UDP</td>
<td>Very Fast</td>
<td>Lossy</td>
<td>Moderate</td>
</tr>
<tr>
<td>DataSocket</td>
<td>Fast</td>
<td>Lossy*</td>
<td>Easy</td>
</tr>
<tr>
<td>VI Server</td>
<td>Slow</td>
<td>Lossless**</td>
<td>Moderate</td>
</tr>
</tbody>
</table>

*Data can be overwritten if items are not read from the DataSocket Server before the next write.
**Not for large data transfers. Mostly used for monitoring, or controlling one shot runs of remote VIs. VI Server is primarily suited for remote control of VIs.

It should be remembered that communication over a network is not deterministic.
Deployed State Communication

Networked Devices and RT Series DAQ Devices

TCP VI Server

Networked Devices only

UDP DataSocket

RT Series DAQ Devices only

Shared Memory

It should be remembered that communication over a network is not deterministic. Using DataSocket, networked RT Engine VIs can publish and read data to and from a DataSocket Server running on another computer. Also, the RT Engine can publish data to a DataSocket Server that many other machines can simultaneously read from. LabVIEW Real-Time 5.1.2 RT Engines do not support DataSocket.

UDP is a faster communication protocol than TCP/IP, but it is not connection-based, and data can be lost since there is no handshaking employed.

DataSocket and UDP are covered in more detail in the LabVIEW Advanced Course.
Deployed State Com. (Continued)

- Logos is supported on FieldPoint 2000 modules.
- CAN (Controller Area Network) is supported in PXI for real-time two wire communication. Can be synchronized with NI-DAQ
- Serial and GPIB communication are also possible.
TCP/IP

- TCP/IP functions the same between the RT Engine and LabVIEW as between two PC’s with LabVIEW.
- Covered in detail in the LabVIEW Advanced Course.
- For RT Series DAQ Devices:
  - TCP/IP communication only with LabVIEW on the Host PC (no other computers)
  - Address is `DAQ::x`, where `x` is the device number in MAX
  - The RT Engine cannot open a connection to the Host PC

TCP/IP is good for directly transferring large chunks of data between RT Engine VIs and Host PC VIs.
The example above can be found in \Examples\comm\tcpex.llb.
VI Server

- VI Server functions the same as between the RT Engine and LabVIEW as between two PCs.
- Covered in detail in the LabVIEW Advanced Course.
- For RT Series DAQ Devices:
  - VI Server communication only with LabVIEW on the Host PC (no other computers)
  - Machine name is `DAQ::x`, where `x` is the device number in MAX
  - The RT Engine cannot open an application reference to the Host PC

One of VI Server’s capabilities is that it allows VIs to be called remotely as though they existed on the caller’s system. VI Server has a lot of powerful functionality that will not be covered in this course, but is covered in the LabVIEW Advanced Course.
This is a simple example of using VI Server to run a VI on the RT Engine using a **Call by Reference Node**. The VI being called using the Call by Reference Node would reside on the RT Engine.
DataSocket

- Data is written to and read from a DataSocket Server (The Server runs in Windows)
- Can have multiple clients

DataSocket Write

DataSocket Read
Network x.x.x.x Options…

To set the options for networked RT devices, after targeting the RT engine, select Tools»Network x.x.x.x Options…

Proper settings important for Network Communication

RT Engines on the Network have options that can be configured while targeted by selecting Tools»Network x.x.x.x Options… See the RT Series PXI User Manual for details on configuring these options.
Exercise 2.4

Students open and run a VI on the RT Engine in the deployed state which communicates with a corresponding Host PC VI using TCP.

Time to complete: 15 minutes
RT Series DAQ Com (7030)

- TCP/IP and VI Server can be used with PCI/PXI 7030 RT Series DAQ Devices. This is implemented through a special area of shared memory.
- 1024 KB of shared memory can be accessed directly using Shared Memory VIs.
- The rest of the chapter is optional if the class does not need 7030 specific information.
(7030) Shared Memory VIs

- 1 KB (1024 bytes) of shared memory accessible on RT Series DAQ Devices from both the RT Engine and Host LabVIEW

- Common Shared Memory VI Inputs/Outputs:
  - device
  - byte offset
  - device out
  - next byte offset

There is 1 KB of shared memory located on the RT Series DAQ Device processor boards that can be accessed by both the RT Engine and the Host PC exclusively using Shared Memory VIs. TCP/IP and VI Server also use shared memory to communicate with the Host PC, but it is not accessible by the user, and does not include the 1 KB reserved for the Shared Memory VIs. There is a pallet of VIs designed exclusively for writing and reading values to and from shared memory. From the Functions pallet select RT » RT Series DAQ.

Many of these VIs share common inputs and outputs such as device and byte offset. For programs running on the Host PC, the device number is the device number for the processor board (i.e., 7030) found in MAX. For programs running on the RT Engine, the device number is 0. The byte offset is the location of the beginning byte for the data you will write or read. Since there is one kilobyte of shared memory, the byte offset can be a value from 0 to 1023. If you try to read or write beyond the 1 KB of shared memory, you will get an error. Device out is a duplicate of device input. Another commonly used output is next byte offset which is the next available location after this VI runs.

The shared memory VIs are typically used to pass parameters or variables rather than large chunks of data. TCP/IP is better suited for large data transfers.
(7030) RT Peek Poke with Error

- VIs read and write directly to Shared Memory, with error checking
- Based on RT Low Level Peek Poke VIs
- No guarantee data written will be read (you must write your own handshaking)
- Example - writing and reading int8 and double:

If a peek is done at the same time as a poke, some bytes may contain old data while others contain new data, which results in a corrupt value.
(7030) RT Shared Memory R/W VIs

- Similar to Peek Poke VIs, but use a flag to determine if data is new or old.
- Data will not be written if it is the same as the previous data, or if the data at that byte offset is still new.
- Data is not read if data flag shows data is old, and previous data input will be returned.

If a peek is done at the same time as a poke, some bytes may contain old data while others contain new data, which results in a corrupt value. The Read/Write VIs avoid this problem by not writing a new value until the entire old value has been read. The tip strips on the diagrams show that the highlighted input is for “previous data (DBL).”
(7030) RT Board Utilities

- To find size of shared memory
- To control LEDs
- To Pass Errors back from the RT Engine to the Host PC
- To Write and Read arrays (writes/reads verification number before and after array to ensure data is not corrupt)

Use the RT Board Utilities VIs located on the **Functions»RT»RT Series DAQ» RT Board Utilities** palette to check the shared memory size, toggle the board LEDs, and facilitate high-level communication between the RT Series DAQ Device and the host PC.
(7030) Incremental Single VIs

Rather than read or write many parameters each loop iteration, these VIs write or read a portion of the data array each iteration.

- Reads only 2 bytes of the array each time it is called

- Writes only 2 bytes of the array each time it is called

When a large number of parameters needs to be passed between the RT Engine and the Host PC, the Incremental Single Read and Incremental Single Write VIs are very useful. Rather than transferring all of the parameters each iteration, which can slow down your loops, these VIs allow you to transfer a small portion of the parameters in each loop iteration. Only a portion of the data array is written or read from shared memory each time. Although this allows your loops to run faster, this means that it may take several loop iterations for a parameter to be transferred, and sometimes parameters values are not transferred at all, since they may change many times before actually being transferred.

There is a concept document in the National Instruments Developer Zone called “Incremental Single Read and Incremental Single Write VIs” which discusses these VIs in more detail.
Exercises 2.5 and 2.6

Students use shared memory VIs
(for RT Series DAQ Devices only).

Time to complete: 20 minutes
Chapter 2—Summary

- Use MAX for configuring RT Series Devices
- LabVIEW Real-Time is used to develop VIs on the Host PC which are downloaded to the RT Engine
- Development State vs. Deployed State
- User Interface Thread runs while in development state at normal priority
- User must write communication for VIs in deployed state
Exercise 2-1  
Objective: To become familiar with MAX in relation to configuring RT Series Hardware.

This exercise assumes that the RT Series Hardware that you will use has been previously installed and configured.

1. Launch MAX. To do this, select Start » Programs » National Instruments » Measurement & Automation Explorer.
2. Refresh MAX by selecting View » Refresh, or press <F5>.

PART 1. Finding Device numbers for RT Series DAQ Device processor boards and their daughter cards.
1. Expand My System, and then Devices and Interfaces if they are not already expanded. Do this by double clicking on the item you would like to expand.
2. Any RT Series DAQ Devices in your PC will show up here. Double click on the RT Series DAQ Device (if you have one installed, such as a PCI-7030-6XX) to expand it to see the DAQ daughter card. Write down the device numbers of the processor board and the DAQ daughterboard. (If you do not have an RT Series DAQ Device, skip to Part 2.)

RT Series DAQ Device:  
Processor Board Device Number _________  
Daughterboard Device Number _________

3. Test the resources for each RT Series DAQ Device by right clicking on the processor board and selecting Properties… » Test Resources. Notify the instructor if your device fails the test.

Tested Passed?  ☐ Yes  ☐ No

Note: You can also configure SCXI Chassis, Virtual Channels, and Scales for the RT Series DAQ Device’s daughter card just like you would for normal DAQ cards in MAX, however, this is beyond the scope of this class. You cannot test daughter cards or virtual channels for daughter cards from MAX.

PART 2. Using MAX to view configuration information for RT Series PXI Controllers.
1. Select Tools » Remote Systems » Find Remote Systems. This will locate any RT Series PXI Controllers on the same subnet as your Host PC. They will then be listed in the Configuration window of MAX under My System » Remote Systems.
2. Expand Remote Systems if it is not already expanded.
3. If there are any RT Series PXI Controllers on the same subnet as your Host PC, they will appear here. Click on the name of the RT Series PXI Controller that you will be using (consult the instructor about which controller you will use if there
are multiple controllers listed). If you will not be using a controller, just read through the remaining steps.

4. Select the **Network Settings** tab and write down the IP address.

   IP Address _____ • _____ • _____ • _____

5. Select the **Firmware** tab and observe the versions installed on the RT Series PXI Controller.

6. From MAX, select **Tools»NI-DAQ Configuration»Remote DAQ Configuration…**, enter the IP Address and click **OK**. Expand the **NI-DAQ Devices** section. Write down the Device Numbers for each of the PXI DAQ cards.

   Device ___: PXI-_________
   Device ___: PXI-_________
   Device ___: PXI-_________
   Device ___: PXI-_________

**Note:** You can also configure SCXI Chassis, Virtual Channels, and Scales for the RT Series Controller using **Remote DAQ Configuration**, however, this is beyond the scope of this class.

7. Click **Exit** when you are finished.

**PART 3.** Using MAX and RDA to test PXI DAQ board resources in PXI chassis with RT Series PXI Controllers. (Skip this part if you are using an RT Series DAQ Device.)

1. In MAX, right-click on **Devices and Interfaces** and select **Create New…**
2. Select **RDA/Ethernet Device** and click **Finish**.

![Image of Measurement & Automation Explorer](image-url)
3. In the **Select Remote Computer** dialog screen that appears, enter the **Remote Computer Name/IP Address** for your PXI controller then click **Next**.

4. Select the device you want to test and click **Next**.
5. Select the **Device Number** that the card will be listed under for your local machine, then click **Finish**. The device number for the card on your local machine is different from the device number of the card on the remote system, although they can be the same.

![Device Selection Screen]

Assign a local device number to the remote device. This is the device number you will use in programs run on this computer that use the remote device.

6. The card should now be listed under **Devices and Interfaces**. Test the resources for the remote DAQ board by right-clicking on it and selecting **Properties...» Test Resources**. If the device fails the test, reboot the RT Series Controller and try the test again. Notify the instructor if the device fails the test a second time.

Tested Passed? □ Yes □ No
7. Right-click on the device and choose Test Panels. Test the DAQ channels to verify the device is working properly.

![Test Panels dialog box](image)

8. After testing the DAQ card, you can delete it from Devices and Interfaces.

**Exercise 2-2**

Objectives: 1) Configure LabVIEW Real-Time to prompt for execution each time LabVIEW Real-Time is started. 2) Run a VI on the target platform using the RT Development System as your user interface.

1. Start LabVIEW Real-Time. From a new VI, select **Tools»Options…**
2. Select **Miscellaneous** from the top pull-down menu.
3. Check the box in front of **Prompt for Target Execution Engine** and click **OK**.
4. Close LabVIEW Real-Time and then start it again. This time the **Select Target Platform** dialog box will automatically appear.
5. From the **Select Target Platform** dialog box, select your RT Engine. Click **OK**.
6. Open the Tank Simulation VI located in `c:\Program Files\National Instruments\LabVIEW 6\Examples\Apps\tankmntr\Tank Simulation.vi`.
7. Open the VI Hierarchy window by selecting **Browse»Show VI Hierarchy** and notice that the thumbtack for `Tank Simulation.vi` is in the horizontal position, which indicates that it has not been downloaded.
8. Download the VI by selecting **Operate»Download Application**. In the Hierarchy Window, select **View»Redraw**. Notice in the VI Hierarchy Window that now the thumbtack is in the vertical position, which indicates that it has now been downloaded.
9. Click the Run button. The RT Development System is now providing a user interface for the RT Engine on the target platform. Experiment with the control values to control the Tank Simulation VI.

10. Exit LabVIEW Real-Time by selecting File » Exit Without Closing RT Engine VIs. This exits the RT Development System but leaves the embedded Tank Simulation VI running on the target platform.

11. Start LabVIEW Real-Time again. From the Select Target Platform dialog box, select the same target platform. Clear the Reset checkbox if you target an RT Series DAQ device. Click OK. Because you chose to exit without closing the embedded VIs, LabVIEW Real-Time establishes communication with the target platform, opens the Tank Simulation VI in the RT Development System on the Host PC, and provides the RT Development System with the most recently updated data from the embedded LabVIEW Real-Time VI still running on the target platform.

12. Exit LabVIEW Real-Time by selecting File » Exit. Close all RT Engine VIs and Quit. Do not save changes. This closes LabVIEW Real-Time and stops and closes the Tank Simulation VI on the target platform.

Exercise 2-3
Objective: To see how an embedded VI interacts with a copy of the VI on the Host PC. You will use the Changed or Missing VIs dialog box.

1. Start LabVIEW Real-Time and select your RT Engine from the Select Target Platform dialog box.

2. Select File » Open and open the TCP - RT Engine VI from c:\Program Files\National Instruments\LabVIEW 6\Examples\RT\RT Communication.llb.

3. Select File » Save As and save the VI to your desktop.

4. Download the VI by selecting Operate » Download Application. Do not run the VI.

5. Select File » Exit Without Closing RT Engine VIs.

6. Delete the VI you just saved to the desktop.

7. Launch LabVIEW Real-Time and select the appropriate target platform from the Select Target Platform dialog box. Clear the Reset checkbox if you target an RT Series DAQ device. Because the RT Development System cannot find the local copy of the VI, it prompts you to locate it. You could select the TCP - RT Engine VI from Examples\RT\RT Communication.llb, but for this exercise, click Cancel. The Changed or Missing VIs dialog box appears. When the RT Development System detects that there is an embedded VI on the target platform, it attempts to open the front panel from the local copy of the VI on the host PC hard drive. If the RT Development System detects a change in version or cannot locate the host copy of the VI, the Changed or Missing VI dialog box appears. When the Changed or Missing VI dialog box appears you have the option to close all the embedded VIs and update them with the copy on the host PC or to exit the RT Development System and not update the embedded VIs.
Exercise 2-4

Objective: To use a VI in LabVIEW targeted to the Host PC to serve as a User Interface for a deployed VI.

This example uses an RT Engine VI and a Host PC VI. The RT Engine VI runs on the target platform in deployed state and communicates with the Host PC VI that runs on the Host PC. The Host PC VI provides the user interface.

2. From the Select Target Platform dialog box, select your RT Series device. Click OK.
3. Open the TCP - RT Engine VI, found in c:\Program Files\National Instruments\LabVIEW 6\Examples\RT\RT Communication.llb and click the Run button. The RT Development System automatically downloads the VI to the target platform, which runs the VI.
4. Select Operate»Switch Execution Target. This disconnects the RT Development System from the target platform but leaves the TCP - RT Engine VI running on the RT Engine in deployed state.
5. In the Select Target Platform dialog box, select Host PC (LabVIEW for Windows) to use LabVIEW on Windows to run VIs rather than the RT Engine on the RT Series device. Click OK.
6. Open the TCP – Host PC VI, found in c:\Program Files\National Instruments\LabVIEW 6\Examples\RT\RT Communication.llb.
7. On the TCP – Host PC VI front panel, you must change the machine name/IP address control to the address of the RT Series device. For RT Series DAQ devices, use DAQ::x, where x is the device number of the desired RT Series DAQ device on your system. For networked RT Series devices, use the IP address of your networked RT Series device.
8. Click the Run button. Because you selected the Host PC rather than the RT Engine in the Select Target Platform dialog box, the TCP - Host PC VI runs on the Host PC. It receives and displays data through TCP/IP from the TCP - RT Engine VI running on the RT Series device.
9. Change the function control from random to sine wave. The desired function type is sent through TCP/IP to the RT Engine VI, which then changes the function it generates and sends to the Host PC.

8. Click the Close all RT Engine VIs and Update button in the Changed or Missing VIs dialog box.
9. Repeat steps 2 through 8, except in Step 6, instead of deleting the VI, add an indicator to the front panel and save the VI. To do this, open the VI in LabVIEW targeted to the Host PC, and when finished close LabVIEW. In Step 7, LabVIEW Real-Time will not prompt you for the VI’s location, but will immediately display the Changed or Missing VIs dialog box. Click the Close all RT Engine VIs and Update button in the Changed or Missing VIs dialog box. This will download the new VI.
10. Exit LabVIEW Real-Time by selecting File»Exit and close all RT Engine VIs.
Exercise 2-5  
Objective: To use a VI running on the Host PC to communicate with a deployed VI (also called embedded VI) using shared memory VIs. This example is for use with RT Series DAQ Devices only!

2. From the Select Target Platform dialog box, select the appropriate RT Series DAQ board from the Select Target Platform dialog box. Click OK.
3. Open the Peek Poke I32 - RT Engine VI, found in the c:\Program Files\National Instruments\LabVIEW 6\Examples\RT\RTCommunication.llb library and examine the diagram. Run the VI.
4. Select Operate»Switch Execution Target to close the RT Development System but not the embedded VI.
5. Select Host PC (LabVIEW for Windows) from the Select Target Platform dialog box. Open the Peek Poke I32 - Host PC VI, found in the c:\Program Files\National Instruments\LabVIEW 6\Examples\RT\RTCommunication.llb library and examine the diagram. Refer to Windows»Show VI Info of these VIs for more information. Run the VI.

Tip: Although peek and poke VIs are the low level, fastest form of communication, using several of these VIs in a control loop can slow down your application. If you need a loop with communication to run at maximum rates, use the RT Incremental Single Write VI and RT Incremental Single Read VI, found on the RT»RT Series DAQ»RT Board Utilities palette.

The size of the shared memory is limited to 1 KB. If you need to transfer several megabytes of data, you must separate the data into smaller portions and then transfer them. In doing so, you must make sure that you do not overwrite data in the shared memory before you read it. TCP/IP VIs manage flow control and are more convenient for bulk transfers.

Exercise 2-6  
Objective: To use Shared Memory VIs to communicate between the Host PC and the RT Engine running on an RT Series DAQ Device. This example is not necessarily a use case, but rather an exercise that teaches the concept of how some of the shared memory VIs work.

1. Build the following VI for the Host PC.
   - This VI will write an array of 10 elements to shared memory using the RT write SGL array VI in each iteration. This array will be written to byte offsets 0 through 43 of the RT Series DAQ Device’s shared memory.
   - It will then read an array that is written by the RT Engine to byte offsets 44 through 103 using the Incremental Single Read VI. The Incremental Single Read VI does not read the array all at once. Instead, each main loop iteration it
checks a flag in front of one of the array elements in shared memory. If the flag indicates new data (flag = 1), then over the next 2 main loop iterations, it will read the element, and then in the next main loop iteration it will change the flag to a zero, indicating that the data is now old.

- At the end of each main loop iteration, it will also check for an error message from the RT Engine VI starting at byte offset 104.
- The shared memory VIs are written so that the next byte offset is calculated automatically, so it does not need to be specified explicitly.
- Change the representation Element data input of the initialize array to single precision.
- Change the device number to that of your RT Series DAQ Device.
- Save the VI when you are finished.
2. Build the following VI. This VI will read an entire array of 10 elements from shared memory each loop iteration and write that array back incrementally (only a small amount of the array will be written in each iteration) to a different location in shared memory.

- The VI first initializes the shared memory to zero.
- In each main loop iteration, the VI reads an array of 10 elements from shared memory using RT read SGL array VI.
- In each main loop iteration, the Incremental Single Write VI will be performing one of several operations, depending on what it did on the last iteration:
  - Check if one of that data input elements is new or not. If it is not new, do nothing. If it is new, check the shared memory flag to see if the previous data element written to shared memory was read yet.
  - Write the first half of the single precision element (two bytes).
  - Write the second half of the single precision element (two bytes).
  - Write a 1 to the flag location indicating an array element in shared memory is new (two bytes).
- Save and run the VI targeted to the RT Series DAQ Device.
3. Switch the execution target to the Host PC and run the VI created in Step 1.
4. Change the element data and notice how the data gets read back in increments.
   When trying to achieve faster loop rates, writing whole arrays each iteration may
   hinder your loop rates. This is where the Incremental Single Read/Write VIs help.
   Instead of writing a whole array each iteration, they only write or read a portion of
   the array. The trade off is that some of the elements of the array may not be
   communicated right away, and some values may be missed entirely.
   If you are only monitoring values, or communicating parameters that change
   slowly, then the Incremental Single Read/Write VIs will allow you to do this
   while maintaining faster loop rates.
5. Exit LabVIEW.
Lesson 3

Multithreading and Priorities

You Will Learn:
A. About multithreading and why it is useful in LabVIEW RT applications
B. About VI priorities
C. Inter-thread communication with global variables
What Is Multithreading?

- **Thread**—an independent sub-program that runs within a program
- **Multithreading**—the ability to break your program into multiple threads of varying priority
- **Priority**—threads can be given different priority levels. Higher priority threads preempt lower priority threads.

A thread is an independent sub-program that runs within a program or process. LabVIEW is a process.
### Multithreading Metaphor

<table>
<thead>
<tr>
<th>Time Critical Priority (one VI only)</th>
<th>Normal Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasks</td>
<td></td>
</tr>
<tr>
<td>Receptionist (OS)</td>
<td>One Mechanic (processor)</td>
</tr>
</tbody>
</table>

**MECHANIC = PROCESSOR:** In this example there is only one mechanic (Processor) which performs the repairs (processing).

**RECEPTIONIST = OPERATING SYSTEM:** The receptionist (Operating System) lines up the service requests as they arrive. He will schedule them in order of arrival, unless a higher priority customer arrives, in which case the receptionist will put that customer before the lower priority customers.

**ONE TOWN AMBULANCE = ONE TIME CRITICAL PRIORITY VI ONLY:** In this example, there is one automobile in the town which has the highest priority: the ambulance. This is recommended for Real-Time programs as well.

**WORK ON CARS OF EQUAL PRIORITY AT THE SAME TIME = MULTI-TASKING WITHIN EQUAL PRIORITIES:** The mechanic is able to work on multiple cars of the same priority at the same time, and progress them all to completion, working a little bit on one car, and a little on the next. (Equal priority threads share the CPU.)

**PREEMPTION:** If the mechanic is working on lower priority cars, and a car of higher priority arrives, he will put the lower priority cars aside to work on the higher priority cars. When he is finished with the higher priority cars, he can return to the lower priority cars.

**NEVER FINISH LOW PRIORITY CARS = STARVATION:** If there always are higher priority cars that need work, then lower priority cars will never get worked on.
Scheduling Threads

- **Round Robin**
  - Threads of equal priority receive equal time slices
  - Might take several turns for a thread to complete

- **Preemptive**
  - Higher priority thread immediately suspends execution of all lower priority threads

LabVIEW Real-Time’s real-time operating system (RTOS) implements a combination of *round robin* and *preemptive* thread scheduling.

*Round robin* scheduling applies to threads of equal priority. Equal shares of CPU time are allocated between the equal priority threads. For example, each *normal priority* thread is allotted 10 milliseconds to run. It executes all the tasks it can in 10 milliseconds and whatever is incomplete at the end of that period must wait to complete during the next allocation of time.

Conversely, *preemptive* scheduling means that any higher priority thread that needs to execute will immediately suspend execution of all lower priority threads, and will begin to execute.
Why Multithreading?

- We need to differentiate between time-critical tasks and non-critical tasks.
  - Some time-critical tasks:
    - Control Loop
    - Safety Monitoring
  - Some non-critical tasks:
    - Communication
    - Data logging

- For Real-Time performance, an OS that can give scheduling priority to time-critical tasks is needed.

Multithreading is useful when you have parts of your code that do not need to be deterministic, or if parts of your code rely on non-deterministic I/O. For example, putting network communication tasks inside the time-critical loop may harm determinism. If you make your time-critical code rely on responses from another PC over the network, then if the other PC does not reply in time, it may cause you to miss a deadline. In order to remedy this, you can break the threads up into time-critical tasks and tasks that are not time-critical, then you can assign higher priority to time-critical tasks to ensure that they always finish on time.
Multithreaded RT Application

Time Critical

- NI-DAQ
- NI-CAN
- NI-VISA with PXI
- NI-Motion

Not Time Critical (Also not deterministic)

- Communication with Host
- File I/O to hard drive
- Serial Communication
- GPIB Communication
Time Critical Priority

- To achieve “hard real-time” determinism is to have the VI priority set to **time critical**.
  In this priority on the RT Engine, no parallelism is allowed, thus no parallel loops. This is to ensure deterministic behavior.
- Only one VI should be set to time-critical priority (with one loop). Nested loops are allowed.
To set the priority of a VI, right-click on the icon and select **VI Properties**…, then select the **Execution** Category and then the Priority Level. The highest priority level that allows multithreading is time-critical priority. Subroutine priority does not allow multithreading, and will be discussed later.

SubVIs will inherit the priority of any of its callers in memory. This inheritance property allows for upward priority inheritance only. For example, even if a normal priority VI calls a normal priority subVI, if that subVI is in a time-critical priority VI that is in memory, then the subVI will run at time-critical priority, even though a normal priority VI called it.
Sleeping

- Sleeping is the act of suspending the execution of a VI or thread.
- VIs can go to sleep, allowing lower priority VIs to run.
- There are many ways to cause a VI to sleep, the first of which is:

  ![Wait Until Next ms Multiple](image)

More ways discussed later…

Sleep mode can be thought of as a programmatic tool that a VI can use to proactively take itself off of the LabVIEW and operating system scheduling mechanisms. Users can programmatically provide a sleep mode in a VI by using the functions **Wait Until Next ms Multiple**, or **Wait (ms)**.
Sleeping and Time Critical Priority

- If any VI in a time-critical thread goes to sleep, the entire thread goes to sleep – (Feature unique to LabVIEW Real-Time)

- Parallel loops not recommended in time-critical VIs—multi-tasking is disabled (Feature unique to LabVIEW Real-Time)

LabVIEW Real-Time’s time-critical priority threads have a unique characteristic that is different from normal LabVIEW scheduling. If any VI on a time-critical thread goes to sleep, the entire thread goes to sleep. Even other VIs on the same thread that did not request to go to sleep will sleep. This is only the case for the time-critical priority (highest) setting. Conversely, if two VIs (or two loops for that matter), are executing on the same thread (other than time-critical priority), and one of them goes to sleep, the other VIs on the same thread will continue to execute. In other words, LabVIEW Real-Time’s execution system will not schedule time-critical priority operations from parallel VIs, or loops, when any one of them sleeps in the same time-critical priority thread. All other priority threads in LabVIEW Real-Time, and all threads in normal LabVIEW, will continue to schedule operations from parallel loops, and/or VIs, in similar threads.

Given the cooperative multitasking nature of scheduling multiple time-critical priority (highest) threads, it is recommended that only one VI, and loop, ever be used with the time-critical priority (highest) setting. This is the only way to receive a guarantee in deterministic execution.
Sleep Mode (Continued)

• Millisecond resolution is available using Wait Until Next ms Multiple (loop rates of 1000 Hz, 500 Hz, 333 Hz, 250 Hz...).

• What if we want a smaller sleep resolution, i.e., on the order of microseconds or nanoseconds?
  We will see later that we can achieve that using hardware tools.

When controlling the rate of a software loop by using the Wait Until Next ms Multiple function users can only achieve rates in 1 millisecond multiples. This means you can either run your loop at full speed providing no sleep at all, or you can achieve loop rates of 1000 Hz, 500 Hz, 333.33 Hz, 250 Hz, 200 Hz, etc. We will discuss a method of implementing loop rates with a finer sleep mode resolution than 1 millisecond multiples later in this presentation.
To understand starvation, consider the following example:

Three processes compete for processor or any other resource. Since process A has a time-critical priority, it will run until it is done using that resource. At that point, one of the two other processes, say Process B, will be allowed to use the resource for one time slice. If at that point process A is ready to run; it will again take control of the resource and run until it has finished its critical section. Again, one of the two processes will be allowed to run. Once again, process B will be chosen. Process C can be blocked from the resource indefinitely.

If process A is not put to sleep for enough time for both processes to run, it can happen that a lower priority process is never allowed to run.
When in the development state, the compiled code will only run in real-time if there is no interaction with the front panel.
Exercise 3.1

Students observe the effects of changing the priority of a VI.

Time to complete: 15 minutes
Global variables pass information between locations in your application that you cannot connect with a wire. You can use global variables to access and pass data among several VIs.

Global variables are built-in LabVIEW objects. When you create a global variable, LabVIEW automatically creates a special global VI, which has a front panel but no block diagram. Add controls and indicators to the front panel of the global VI to define the data types of the global variables it contains. In effect, this front panel is a container from which several VIs can access data.

Global variables can add jitter to the program because they are a shared resource. Shared resources and better methods of inter-thread communication are discussed later.
Here is an example of a simple distributed application with multithreading. The RT Engine is in the Deployed State. It has two loops, a time-critical loop and a normal priority loop. This can be achieved by having the top level VI set to normal priority, with a communication loop inside the normal priority VI. The time-critical loop can be inside a subVI of the normal priority VI. This subVI should be set to time-critical priority. The time-critical VI should execute independently of the normal priority VI. The Normal Priority Communication loop can have some type of communication method in it like TCP/IP, UDP, or DataSocket.

In this architecture, we separate all the tasks that are critical for the application from non-critical ones. For example, control loops and critical event responses should all be in a separate time-critical loop. All the tasks that are less critical or tasks that can jeopardize determinism such as communication back to the host or logging to file, should occur in a separate normal priority loop.
You can replace the normal priority communication loop with a VI that just reads and writes to the global variables which the time-critical VI also reads and writes to. The Host PC could then call the normal priority VI through VI Server to get the variable values. This form of communication is lossy, though, since the global variables could be updated many times before the normal priority VI gets called by the Host PC. There are many examples of this type of communication in the Examples\RT folder.
The simplest way to assure that starvation does not occur is to add Wait VI’s to your loops. This will put your high-priority thread to sleep and will give the other threads processor time. You want to make your sleep time long enough for the other threads to accomplish their tasks, but short enough that your determinism is not compromised.
It is important to do all file I/O operations in a normal priority loop. This is because file I/O is inherently non-deterministic. File I/O calls in the time-critical loop can jeopardize the determinism of the entire program.

When transferring data to disk, the transfer rate is largely dependent on the size of the read or write you are performing. In reading and writing, the optimal size is 512 bytes. This is because 512 bytes is the size of a sector on a hard disk, and is the basic unit that is read or written from a disk. If you wanted to read 32 bytes, you would have to read an entire 512 byte sector first, then pull out the 32 bytes that you need from it. The same issue arises with writing—if you want to write 32 bytes, you have to read the full sector first, then replace the bytes, then write the entire sector back to disk. Since LabVIEW Real-Time does not do disk caching on the embedded side, we see these effects clearly. On platforms with disk caching, the performance is typically higher and more balanced at the expense of determinism.
Exercise 3.2

Students use global variables to communicate between threads and the VI Server Call by Reference Node to communicate between the RT Engine and Host PC.

Time to complete: 20 minutes
Exercises 3.3 and 3.4 (Optional)

Students build multithreaded RT Engine VIs that communicate with the Host PC VI using:

- DataSocket—Exercise 3.3
- VI Server Get/Set Control Value—Exercise 3.4 (Optional)

Time to complete: 40 minutes
Chapter 3—Summary

- LabVIEW uses a combination of round robin and preemptive scheduling
- Time-critical tasks need to be in a separate thread from rest of tasks
- Time-critical threads should have some sleep time to avoid starvation of lower priority threads
- Only one time-critical VI with no parallel loops should be used
Exercise 3-1
Objective: To observe the effects of changing the priority setting of a VI.

1. Launch LabVIEW Real-Time and select the appropriate RT Series device from the Select Target Platform dialog box. Close any VIs already running on the target platform.
2. For RT Series DAQ Devices, select the Reset checkbox to reboot the hardware if needed.
3. Click the OK button.
4. Open the Priority Trouble VI from c:\Program Files\National Instruments\LabVIEW 6\Examples\RT\RT Tutorial.llb.
5. Right-click on the VI icon on the front panel and select VI Properties…
6. Select Execution from the pull-down menu. Notice that the priority is set to time-critical priority (highest). This makes the VI run in the highest priority thread on the RT Engine.
7. On the front panel, set the msec to wait control to greater than 0. 10 is the default. Run the VI. The chart displays a sine wave.
8. Slowly decrement the msec to wait control. With each step, the chart display speeds up. However, when the msec to wait control reaches zero, the front panel no longer updates.

The reason for this behavior is that the RT Engine can no longer communicate with the RT Development System front panel. Because the time-critical loop has the highest priority, the user interface thread that updates front panel objects cannot use any processor time. As a result, the front panel and VI appear locked. However, the code is still running.

While msec to wait control is greater than 0, the time-critical loop will “sleep” for that many milliseconds. During this sleep, you can run non time-critical tasks like updating user interfaces.

9. Click the Abort button. After a short period of time, the RT Development System recognizes that it has lost communication with the RT Series device, and displays a warning message. Click OK to close the RT Development System. You must reset the RT target before the RT Development System can communicate with the target platform again.

Exercise 3-2
Objective: To build a Multi-threaded RT Engine VI that communicates with the Host PC through VI Server.

1. Build the following VI and set it to Time Critical Priority. Save the global variables in a file called Global.vi. This Time Critical Loop represents a VI that needs to run in real-time. It generates “data” in the form of a numeric.
We would like to monitor the data from the Host PC, so we will use global variables for inter-thread communication to another VI that will assist in communication with the Host PC. Remember to change the continuation terminal of the while loop to **Stop If True**.

2. Next create a VI as shown below using the global variables from Global.vi. Modify the connector for this VI so that **stop** is an input, and **Numeric** is an output. This VI will be called from the Host PC using a Call By Reference Node VI.

3. Build the following Top Level VI using Parameters VI and Time Critical Loop VI as subVIs. This way you only have to download one VI instead of two. **Download** and **run** your top level VI on your RT Engine.
4. Check to make sure that VI Server on the RT Engine is configured to allow your Host PC to access it. For RT Series DAQ Devices, this configuration is done automatically. For RT Engines on the network, you will need to do the following:
   a. With LabVIEW Real-Time targeted to the RT Engine, from a VI select Tools»Network: x.x.x.x Options…
   b. From the top pull-down menu of the Options dialog box, select VI Server: Configuration and make sure that TCP/IP is checked and that the Port is 3363.
   c. From the top pull-down menu of the Options dialog box, select VI Server: TCP/IP Access and make sure your Host PC’s IP address is in the TCP/IP Access List. (Hint: if you put * in the access list, then all clients are allowed VI Server Access.)
   d. From the top pull-down menu of the Options dialog box, select VI Server: Exported VIs and add * to the Exported VIs list.
   e. Click OK when finished.

5. Switch execution target to the Host PC. Create the following VI. To create the type specifier VI Refnum, right-click on that output terminal of the Open VI Reference and select Create»Control. Right-click on the type specifier VI Refnum (for type only) control and choose Select VI Server Class»Browse…, then select the Parameters.vi. The type specifier VI Refnum tells the Call by Reference Node what type of inputs and outputs are used by Parameters.vi. Since the Parameters.vi VI is in memory on the RT Engine, the Open VI Reference only requires the VI’s name, and not a path.

This VI will call Parameters VI remotely using a Call By Reference Node. This will have the effect of writing the Stop Remote VI value to the stop global variable on the RT Engine, and reading Numeric from the RT Engine.
6. Enter the IP Address or machine name for the RT Engine, and run the VI with LabVIEW targeted to the Host PC.

7. Start and stop the Host PC VI and notice the behavior. The time-critical VI is updating the Numeric global variable on the RT Engine. When we call Parameters VI, it reads that global variable on the RT Engine and returns it to the Host PC.

8. Click on Stop Remote VI, and then exit LabVIEW. This method of communication is easy to program, but is used primarily for monitoring the data on the RT Engine.

Exercise 3-3 (RT Engines on Network Only)
Objective: To build a Multi-threaded RT Engine VI that communicates with the Host PC through a DataSocket Server.

1. Start the DataSocket Server on your Host PC by selecting Start»Programs»DataSocket>DataSocket Server. If this option is not available, run C:\Program Files\National Instruments\DataSocket\cwdss.exe.

2. Start the DataSocket Server Manager on your Host PC by selecting Start»Programs»DataSocket>DataSocket Server Manager. If this option is not available, run C:\Program Files\National Instruments\DataSocket\cwdssmgr.exe.

3. In the DataSocket Server Manager under Permission Groups, add “everyhost” to DefaultReaders, DefaultWriters, and Creators, then select Settings»Save Settings Now.

4. Build the following VI and use the Time Critical Loop VI and Global VI that you created in Step 1 of Exercise 3-2.
5. Enter `dstp:\\[IP address]\\Numeric` (use your Host PC’s IP address) for the URL of the DataSocket Item to write to.

6. Enter `dstp:\\[IP address]\\stop` (use your Host PC’s IP address) for the URL of the DataSocket Item to read from.

7. Run the VI targeted to your Networked RT Engine.

8. Switch execution target to the Host PC and build the following VI below.

9. Enter `dstp:\\[IP address]\\stop` (use your Host PC’s IP address) for the URL of the DataSocket Item to write to.

10. Enter `dstp:\\[IP address]\\Numeric` (use your Host PC’s IP address) for the URL of the DataSocket Item to read from.

11. Run the VI targeted to your Host PC.
12. Stop and start the Host PC VI and observe the behavior.
13. Click on the **Stop Local VI** button. This will stop the RT Engine VI.
14. Exit LabVIEW. Keep in mind when using DataSocket that the writer can write more than once before a read occurs, resulting in lost data.

**Exercise 3-4**
**Objective:** To build a Multi-threaded RT Engine VI that communicates with the Host PC through a VI Server and Get/Set Control Value.

1. Build the VI shown below. It is similar to the Time Critical VI used earlier. Set it to time-critical priority.
2. Check to make sure that VI Server on the RT Engine is configured to allow your Host PC to access it. For RT Series DAQ Devices, this configuration is done automatically. For RT Engines on the network, you will need to do the following:
   a. First target LabVIEW Real-Time to the RT Engine, and from a new VI select Tools»Network: x.x.x.x Options…
   b. From the top pull-down menu of the Options dialog box, select VI Server: Configuration and make sure that TCP/IP is checked and that the Port is 3363.
   c. From the top pull-down menu of the Options dialog box, select VI Server: TCP/IP Access and make sure your Host PC’s IP address is in the TCP/IP Access List. (Hint: if you put ‘*’ in the access list, then all clients are allowed VI Server Access.)
   d. From the top pull-down menu of the Options dialog box, select VI Server: Exported VIs and add ‘*’ to the Exported VIs list.

3. Run the VI targeted to the RT Engine.
4. Target LabVIEW to the Host PC and build the VI below. The Flatten To String VI, and Unflatten From String VI, are found in the Functions»Advanced»Data Manipulation palette.

5. Enter the machine name or IP address of your RT Engine.
6. Run the VI in LabVIEW targeted to the Host PC.

   When the Time Critical VI on the RT Engine is sleeping, the VI Server on the RT Engine can communicate with the Host PC.

7. Stop and start the VI and observe the behavior.
8. Click on the Stop Remote VI button.
9. Exit LabVIEW.
Lesson 4

Timing and Sleep

You Will Learn:

A. About the difference between software and hardware timing
B. How to use various tools to control and measure loop rates and induce sleep
C. About caching effects and how to counteract them
Software Determinism and Jitter

- Software jitter, though unpredictable, can be measured, controlled, and bounded
- Use Wait Until Next ms Multiple to introduce software timing and achieve µs jitter (~15 µs)

We will soon see that we can easily implement software-timed control loops, where all analog acquisitions and analog outputs are initiated through software. As a result we are limited to the deterministic capabilities of software. Fortunately, with LabVIEW Real-Time, software determinism is frequently adequate. By timing our loops with Wait Until Next ms Multiple, we can mask out most of the software jitter within the loop, leaving only the inherent software jitter incurred from the Wait function itself, which is typically around 15 µs. One caveat about this method is that you must ensure that your code within the loop executes faster than the requested Wait Until Next ms Multiple period. This will become more clear once we have covered the Wait Until Next ms Multiple function in detail.
Hardware Determinism and Jitter

- DAQ-STC has a 20 MHz time base
- Hardware jitter is typically on the order of nanoseconds
- Current NI-DAQ versions* offer new event messaging techniques using E-Series devices

In systems that require more determinism (i.e., less jitter), we can use DAQ event messaging techniques to achieve control response with nanosecond determinism. By anchoring our timing to the DAQ time base, we can expect output responses to occur at some user-defined interval, plus or minus a few nanoseconds!

By leveraging the determinism of the hardware, software jitter only becomes a limiting factor of the overall performance of the control loop (i.e., it limits the maximum control loop rate).

*NI-DAQ 6.7 introduced the AI SingleScan event message, and NI-DAQ 6.9 introduced the Counter Control—Wait event message.
Before we move into great detail about timing and sleep, we can summarize the difference between software and hardware determinism with a few key points. Software determinism is typically much worse than hardware determinism because code execution time almost always varies more than a timebase governed by a hardware oscillator (i.e., the 20 MHz time base on the DAQ-STC). Depending on the time base and accuracy of the oscillator, hardware jitter generally ranges from 1 to a few hundred nanoseconds. On the other hand, software jitter within the real-time OS is around 15 µs at best. We will learn how to control and minimize software jitter in this lesson and in Lesson 5. Once we have the tools to measure software determinism, we can demonstrate these points more clearly.
Every millisecond, the OS updates an internal clock to keep track of time. Wait Until Next ms Multiple synchronizes itself with this clock. When called, Wait Until Next ms Multiple waits until the OS clock counts up to the next millisecond multiple requested. Each time Wait Until Next ms Multiple is called in the above diagram, it waits until the OS clock has reached the next multiple of 100 ms. If the loop begins executing arbitrarily between 100 ms multiples, Wait Until Next ms Multiple waits long enough to synchronize the next loop cycle to the next 100 ms multiple of the OS clock. In the slide shown above, My Proc may take a variable amount of time to finish executing, but by calling Wait Until Next ms Multiple afterwards, we enforce a loop frequency of 10 Hz (1/100 ms). The maximum loop rate that can be achieved using this method is 1 kHz with a wait multiple of 1 ms. Since Wait Until Next ms Multiple only accepts integers, loop rates are limited to only a few frequencies: 1000, 500, ~333, 250, 200, ~167 Hz, etc.

In the example above, the 100 ms timer is initialized by calling the Wait function immediately before the while loop begins. Otherwise, the loop time for the first iteration would be indeterminate. In the while loop, the Wait function is enclosed in a sequence structure to assure that the delay is added after My Proc has finished. The output of My Proc is wired to the sequence structure to create data dependency. In this way, the order of execution is guaranteed.

Before deciding on a millisecond multiple, you must be certain that the code in your loop can execute faster than the wait multiple. If your code takes slightly longer than the ms multiple, then your loop will actually wait two ms multiples, since code was running when the first ms multiple arrived. In this case, the wait until ms multiple will not be aware that the first ms multiple occurred, and will wait until a second ms multiple occurs before returning.

In addition to controlling loop rates, Wait Until Next ms Multiple forces time-critical VIs to “sleep” until the wait is finished. When a VI sleeps, it relinquishes the CPU, allowing other VIs or threads to execute. Sleep time is required in time-critical VIs because the user interface and other background processes (ftp server, etc.) need CPU time to survive. Without any sleep, a time-critical VI starves other processes, including the user interface and communication thread in development mode.
Software-Timed Analog Input

Accurate software timing requires $T_{wc}$ be less than the wait multiple, $\Delta T$.

The Wait Until Next ms Multiple function masks software jitter within the loop. The Wait function has some inherent jitter, which is acceptable for many real-time applications. In this slide, the Wait function synchronizes with each 1 ms tick of the OS clock, allowing the loop to achieve 1 kHz software-timed analog input. The timeline illustrates the 1 ms wait multiple in yellow (top arrow), the actual time required to execute the code in red, worst case jitter in purple, and worst case time in green. As long as the worst case time is smaller than the wait multiple, the actual loop period will equal the wait multiple, plus or minus the jitter incurred by the wait function itself.
Using the Wait Until Next ms Multiple function, you can achieve a loop rate of up to 1000 Hz. Alternately, loop rates can achieve their maximum rate if no delay is present, but without a delay, you cannot precisely control the rate of the loop. Furthermore, if the VI is set for time-critical priority, you risk starving other processes of CPU time. NI-DAQ 6.7 introduced AI SingleScan event messaging, which is almost identical to the existing DAQ occurrences. AI SingleScan event messaging, however, is faster and easier to program than DAQ occurrences because no extra VIs are required to anchor loop timing to the scan clock signal on E-Series DAQ devices.

As you may have already noticed, the loop rate matches the scan rate programmed at AI Start, and this means your loop rates are no longer restricted to 1000 Hz. The scan clock is simply an onboard signal which initiates analog to digital (A/D) conversions at the user-defined scan rate. Each scan clock edge corresponds to a single scan across all channels in the scan list. Calling AI SingleScan can have many different effects, depending on how the buffer and hardware have been configured. To activate AI SingleScan event messaging, you must configure the scan clock (i.e., set up hardware timing as opposed to software timing), and you must use no buffer (i.e., buffer size = 0). In “hardware-timed, non-buffered” mode, AI SingleScan collects data from the onboard FIFO buffer each time it is woken up by a new scan clock edge. For relatively slow scan rates, AI SingleScan can retrieve new data as fast as it enters the onboard FIFO. Overwrite errors can occur, however, when the requested scan rate is too fast relative to the code execution time. In other words, if the CPU is not powerful enough to execute the code within the loop at least as fast as the scan rate, the FIFO will eventually overflow with new, unread data.
Hardware-Timed Analog Input

Accurate hardware timing requires $T_{wc}$ be less than the scan interval, $\Delta T$.

AI SingleScan acts like Wait Until Next ms Multiple because both functions synchronize themselves to a clock, and both functions induce sleep within the calling VI during the wait period. Unlike Wait Until Next ms Multiple, however, AI SingleScan has an output parameter, which indicates when the loop is keeping up with the requested timing interval. “Data remaining” is a convenient indicator of whether the overhead in the loop is preventing AI SingleScan from keeping up with the scan clock. You can verify that AI SingleScan’s sleep mode is enabled by looking for the following entry in your NI-RT.INI file associated with the target on which you are executing code:

[NI-DAQ]
UseSleepingTimedNonBuffered=TRUE

This entry is present by default for all networked RT Series PXI Controller targets. Networked RT Series PXI Controller’s NI-RT.INI files can be found in C:\NI-RT.INI of the target’s hard drive. The AI Single Scan sleep mode is not turned on by default for the 7030 family RT Series DAQ Devices. The 7030 product family’s NI-RT.INI file can be found in your ~\LabVIEW RT\Resource folder.

In addition to providing greater-than-millisecond resolution to control loop timing, AI SingleScan’s “sleep mode” benefits multithreaded applications. A control loop running at time-critical priority will starve other low priority threads if there is no delay or sleep mode built into the loop. AI SingleScan’s sleep mode ensures that your time-critical code relinquishes the CPU while it is waiting for the next scan clock edge, allowing other lower priority processes, including the user interface, TCP/IP communication, and file I/O to run. Furthermore, only the time-critical priority setting ensures a 1-to-1 correlation between your control loop and scan clock signal. If set to normal priority, for instance, AI SingleScan is allowed to read from the onboard FIFO when a scan clock edge arrives OR when there is a backlog of data in the FIFO. This is important because the “scans remaining” output parameter of AI SingleScan can be misleading when run at normal priority. Because AI SingleScan is not necessarily 1-to-1 with the scan clock, there is no guarantee that zero scans remaining means AI SingleScan never missed a scan clock edge. Note: When using hardware-timed, non-buffered AI SingleScan, you must be sure that no VIs are called between AI Start and the first call to AI SingleScan to avoid a backlog of scans in the FIFO. Later, we will learn about the input “read newest data,” which can counteract this effect in certain cases.
Measuring Loop Rates—Software

Subtracting two well-placed Tick Counts can yield accurate timing information.

The easiest method for measuring how fast a loop executes involves Tick Count. By simply taking a tick count before and after several iterations of a loop, we can estimate average loop cycle time with millisecond resolution. In this slide, we use a sequence structure to ensure Tick Count gets called immediately before and immediately after the loop we are timing.
Exercise 4.1

Students control loop cycle time with AI Single Scan event messaging, and then measure average cycle time with Tick Count.

Time to complete: 30 minutes
Hardware-Timed Analog I/O

System under control experiences only hardware jitter when AI and AO share a clock.

By leveraging the determinism of the hardware, software jitter only becomes a limiting factor of the overall performance of a control loop, limiting the maximum control loop rate, not the determinism felt by the system under control. Software jitter is no longer an issue for the system under control because the system under control only experiences hardware jitter.

This slide depicts a modified example, which uses AI SingleScan event messaging. At very accurately spaced hardware clock cycles, analog inputs and outputs are acquired/updated simultaneously because they are sharing the same clock on PFI7. The diagram ensures that updates sent to the system under control occur exactly when scans occur with only a few nanoseconds of jitter. Therefore the system being controlled by LabVIEW Real-Time will not experience any software jitters long as the nominal “Code Execution Time,” $T_e$, plus the “Worst Case Jitter” spike, $T_j$, is less than the “HW scan interval,” $\Delta T$. If this condition is always true, the system under control will always experience hardware determinism, which is on the order of nanoseconds!
This diagram illustrates the key ingredients to achieving hardware determinism with AI SingleScan event messaging. As described on the previous slide, this diagram features analog input and analog output sharing the same clock. AI SingleScan does not initiate the A/D conversion, rather the scan clock does that. The same is true for D/A conversions, which are initiated, not by the call to AO SingleUpdate, but rather by scan clock. The system under control, therefore, is scanned and updated at precisely the same time with each scan clock edge. Inside the loop, AI SingleScan gathers a point (say point N) from the FIFO, which stores A/D samples. After the scan is collected and processed, AO SingleUpdate loads a response to point N on the D/A register. However, the D/A conversion of response to point N does not occur until the next scan clock edge, when point N+1 is sampled. Using this architecture, D/A response is delayed by one scan interval but is guaranteed to occur with only a few nanoseconds of jitter.

**Note:** AO SingleUpdate must use the opcode “output only” when analog output shares the scan clock with analog input.
As of NI-DAQ 6.9, Counter Control has a new parameter called “wait,” and Counter Set Attribute has a new parameter called “timeout” (seconds). When performing pulse train generation, Counter Control will induce sleep with the wait parameter and wake up when the counter outputs the next pulse. To be precise, the VI wakes up one active source edge (which we will call the waking edge) before the output begins the next pulse. If a waking edge occurs before the Counter Control is entered, then the VI will return immediately. So if waking edges are occurring too fast, your VI will not sleep at all. Because Counter Control induces sleep, it can be a useful tool for preventing synchronous calls from hanging your application.

Like AI SingleScan event messaging, Counter Control event messaging allows us to anchor loop timing to hardware events. The only difference is Counter Control waits for pulses generated by a general purpose counter on the DAQ-STC instead of pulses generated by the scan clock, which wakes up AI SingleScan.
Exercise 4.2

Students control loop cycle time with Counter Control event messaging, and then use Tick Count to measure average cycle time.

Time to complete: 15 minutes
In some of the previous exercises, we used the Tick Count VI to time our loops. Though Tick Count offers a simple way to measure average loop speed, it only provides us with millisecond resolution. Consequently, calling Tick Count each time within the loop to obtain an execution profile over time would not benefit us much if the loop were running faster than 500 Hz. On the other hand, onboard general purpose DAQ counters and even the CPU time-stamp counter offer much better timing resolution than 1 millisecond. In fact DAQ-STC counters (i.e., those found on E-series devices) have up to a 20 MHz time base, and NI-TIO counters, which are present on 660x devices, have a time base of up to 80 MHz. If the RT target is a PXI controller, you can even use the processor’s built-in time-stamp counter to keep track of time. The CPU counter increments with each CPU clock cycle and has a resolution of 1/CPU frequency, which amounts to 3 ns (1/333 MHz) on the PXI-8156B and ~1.17 ns (1/850 MHz) on the PXI-8170. In the next few slides we will examine one way to time loops using counters and how to time loops using the CPU’s time-stamp counter.
Measure Timing with a Counter

Configure a counter for simple event counting then obtain a count within each iteration.

By configuring a counter for simple event counting, we can utilize the accurate time base of the DAQ-STC as the counter’s source. Once the counter has begun incrementing, we can read the count register each time our loop iterates and store these count values as time-stamps within an array for later processing. Subtracting adjacent time-stamps yields approximate loop execution time for each loop iteration. This information is more useful than an average loop cycle time because we can also determine how much software jitter is present by plotting the time-stamps on a histogram.

Of course, you have probably realized that by calling Counter Get Attribute within the loop, we are adding overhead (and possibly jitter) to the actual loop execution time. As an alternative to calling Counter Get Attribute, you can instead call Port Write to toggle a digital line on the E-Series board. By wiring the output of the digital line to the gate of a counter performing buffered event counting, you can acquire timestamps with Counter Read Buffer.

**Note:** Buffered event counting is not covered in this course because it involves advanced DAQ knowledge that is beyond the scope of this course. Please ask the instructor to explain this method in more detail if Counter Get Attribute incurs too much overhead relative to your actual loop execution time.
Measure Timing with RDTSC

A faster, more precise timing method involves the CPU’s time-stamp counter. On PXI controllers only, LabVIEW Real-Time users can call a special assembly instruction to measure loop timing at the resolution of the processor clock (1/processor frequency). The assembly instruction is exposed through a VI called Read Time Stamp Counter or RDTSC. This VI reads the 64-bit time-stamp counter value from Pentium-class processors. The processor counter starts from zero each time the controller is turned on. If the reset button is used to reset the controller, the processor counter will reset. If MAX is used to reset or reboot the controller, the processor counter will reset. In order to convert the time stamp information to a time value, you need to input the processor speed of your machine to this VI. The timing code takes the array of time-stamps and converts them into actual loop cycle times based on the CPU speed. The results are then plotted as Loop Cycle Time vs. Loop Iteration.

This VI can yield erroneous data when run on laptops because laptops frequently adjust the processor clock speed dynamically to conserve energy when there is no load. To remedy this problem, place an empty running WHILE loop set at NORMAL PRIORITY in parallel with the application being timed. The RDTSC instruction also returns erroneous data from dual processor machines because the instruction does not always grab the time stamp from the same processor. Each processor's Time Stamp Counter runs independently of the other.

Note: AMD processors used in the PXI-8156B Controller are Pentium-class.
The timing shell’s front panel consists of several controls and indicators, most notably the one large graph. The most relevant controls include **iterations** and **processor speed**. Iterations dictates the number of test code iterations the FOR loop will execute, and processor speed lets the timing code accurately convert time-stamp data into actual loop cycle times. As you can see, the timing shell displays loop cycle time for each loop iteration.
Exercise 4.3

Students examine RDTSC benchmark utility and time various diagrams.

Time to complete: 40 minutes
Memory Cache

Cache memory lets CPU retrieve instructions and data quickly.

The CPU may swap new instructions between main memory and cache.

The most commonly used instructions are kept in a small cache, generally made up of SRAM located on or near the processor die. Level 1 cache typically resides on the processor die, while level 2 cache is separate from the processor but can be accessed faster than normal memory. When new instructions must be processed, caching could occur and might affect execution times. Processors are designed to optimize execution time by placing more commonly used instructions in cache memory, which can be accessed faster than main memory.
Caching Effects

The first few cycles of your control code might run slowly.

Use AI SingleScan’s opcode to counteract caching effects.

Processor cache effects the speed that code can execute in a loop. The first time the loop is run, the cache does not necessarily contain the instructions that are used in the loop. Instead the processor most likely still has instructions that were most frequently used before the loop began running. When instructions that the processor needs are not in the cache, it takes longer for them to load, but if they are in the cache, they load quickly from the cache. This means that your initial loops may execute slower than later loops.

If you are using software or hardware timing to time your loops, and the loop execution time is much faster than wait multiple (software timing) or scan interval (hardware timing), then you do not need to worry about caching effects. However, in order to achieve optimal loop rates, you will need to account for caching effects.

With AI SingleScan, you can counteract caching affects by using the opcode. For the first few iterations, use the opcode “read newest data.” This assumes the following:

- Buffer Size = 0 (Sleeping will not work otherwise.)
- Hardware Timing
- RTOS
- Sleep Mode is enabled
- AI SingleScan is in a loop in a Time Critical VI

Read Newest will return the newest scan in the FIFO when the scan clock rising edge occurs and will empty the FIFO. After each loop iteration, the loop will execute faster and asymptotically approach some maximum loop rate. When the loop rate is running faster than 1/scan rate, you can then switch the opcode to read oldest data.
Caching effects generally disappear after the first few iterations unless you have other threads running while your time-critical code sleeps. Consider a scenario where you have achieved an optimal loop execution time in a time-critical loop that sleeps a lot. You also counteracted initial caching effects by switching from read newest to read oldest after many iterations. Now if you introduce lower priority code that can run while your time-critical code is sleeping, it will invalidate your cache, causing the processor to swap instructions back and forth between main memory and cache memory each time LabVIEW RT switches threads. However, since the time-critical code is already running at near optimal speeds, it will have less time to sleep when caching occurs. As the time-critical code sleeps less, the other lower priority threads are allotted less CPU time or possibly none at all. In the worst case scenario, the time-critical loop will not keep up with the wait multiple or scan interval, depending on whether you have implemented software or hardware timing. When the loop execution time cannot keep up with the requested loop interval due to caching effects, you must slow down the loop timing. Generally, you do not have to compromise your loop rate to run lower priority threads. This is only true if caching effects prevent your time-critical code from keeping real-time.

Read newest data with AI SingleScan throughout the initial iterations. Using the “read newest data” opcode, only the newest scan in the FIFO is collected from the DAQ device, while the remaining points are thrown out. Reading newest data prevents the FIFO from overflowing during the caching iterations and removes stale data points, which can disrupt control response.
Counteracting Caching Effects

After initial caching effects have subsided, switch AI SingleScan’s opcode to “read oldest data.” By switching the opcode, we can now monitor AI SingleScan’s “scans remaining” output parameter. This parameter is a measure of real-time because scans remaining must always be zero to ensure our loop execution time occurs faster than the requested scan interval. On the other hand, if scans remaining grows larger than zero, we can deduce that AI SingleScan is not keeping up with the scan interval (i.e., not keeping real-time).
Counteracting Caching Effects

Use the iteration terminal to switch the opcode.
Cache Effects and RDTSC

Use “Initial points to ignore” in RDTSC utility to only analyze data after caching effects.

“N” initial loop iterations are thrown out of the timing code before mean cycle time and jitter are evaluated.

The RDTSC timing shell has provisions for initial caching effects. The “Initial points to ignore” allows you to discard the first “N” time-stamps so as not to group the caching iterations with the optimal loop iterations when running the timing code to compute mean, minimum, and maximum loop cycle time.
Chapter 4—Summary

Software versus hardware jitter
- Software jitter is usually larger than hardware jitter
- Hardware timing masks software jitter

Controlling and measuring loop rates
- Use Wait Until Next ms Multiple to control loop rate
- Use Tick Count to measure loop rate with ms resolution
- Use onboard counters to control or measure loop rate
- Use RDTSC to measure loop rate—highest resolution

Caching Effects can induce jitter
- Use AI Single Scan “read newest data” to compensate for caching
- Use “Read N Extra Initial Entries” to avoid caching in RDTSC
Exercise 4-1

Objective: To control loop rates by using AI Single Scan.

1. Launch LabVIEW Real-Time and select the appropriate RT Series Device from the Select Target Platform dialog box. Close any VIs already running on the target platform.
2. For RT Series DAQ Devices, select the Reset checkbox to reboot the hardware.
3. Select File»Open and open Cont Acq&Chart (hw timed) VI from c:\Program Files\National Instruments\LabVIEW 6\Examples\daq\anlogin\anlogin.llb.
4. Right-click on the VI icon on the front panel and select VI Properties.
5. Select Execution Options from the pull-down menu. Set the priority to time critical priority (highest). This makes the VI run in the highest priority thread on the RT Engine.

To achieve real-time control, we must run the VI at time-critical priority to ensure that 100% of the CPU time is dedicated to acquiring and processing within the loop. AI Single Scan will synchronize itself to the scan rate and will also provide sleep within the loop when the software finishes before the next scan clock edge. This sleep time yields CPU time to other lower priority processes like the communication thread that sends new values to our user interface running on the host machine.

Before we run the example, we will add a Tick Count before and after the loop to determine whether AI Single Scan really is synchronized with the scan clock.

6. Open the functions palette by right-clicking on the diagram window and select the sequence structure located under the structures sub-palette.
7. Wrap the sequence structure around the while loop in the example.
8. Next add a sequence frame before and after the while loop and place Tick Counts in these new sequence frames. In frame two, perform a subtraction on the Tick Count values to determine approximate execution time.

This timing shell will only tell us how long overall the code took to execute. To determine how fast each loop cycle executed on average, we need to divide the overall time by the number of loop cycles. This can easily be accomplished by passing the loop iteration terminal from the while loop in frame 1 to the timing code in frame 2.

9. Modify the code in frame 1 and frame 2 to determine the average execution time per loop cycle.

10. Run the VI with a scan rate of 100 Hz, and then stop the VI after a few seconds. Notice that the average loop cycle time was approximately 1/100 second, which corresponds to a period of 10 milliseconds.

Exercise 4-2
Objective: To control loop rates by using a DAQ-STC counter.

1. Launch LabVIEW Real-Time and select the appropriate RT Series Device from the Select Target Platform dialog box. Close any VIs already running on the target platform.

2. For RT Series DAQ Devices, select the Reset checkbox to reboot the hardware.

3. Select File » Open and open counterwait VI from Exercises\Chapter 4\counterwait.llb.
4. Study the diagram and notice that the VI simply performs pulse train generation with a couple of modifications.

The first thing that sets this example apart from normal pulse train generation is that we configure a **timeout** (in seconds) using Counter Set Attribute VI. The second node that makes this example unique is Counter Control within the while loop. Notice that we use the **wait** parameter, which causes Counter Control to become synchronized to the pulse train generated by the counter. If for some reason the counter failed to start, we need it to time out to prevent the loop from hanging indefinitely, hence the **timeout** mentioned earlier.

5. Now add a sequence timing shell around the while loop as you did with AI Single Scan in the previous exercise.

6. Experiment with different **pulse spec** parameters and time the loop for several iterations to verify that Counter Control really is synchronized to the counter’s pulse train.

**Exercise 4-3**

**Objective:** To examine the Read Time Stamp Counter (RDTSC) timing shell. (For RT Series PXI Controllers Only)

1. Launch LabVIEW Real-Time and select the appropriate RT Series device from the **Select Target Platform** dialog box. Close any VIs already running on the target platform.
2. Select **File»Open** and open **RDTSC Timing Shell VI** from Exercises\Chapter 4\RDTSC Timing Shell.llb.
3. Look at the diagram. Inside the VI, you will find a For Loop and a subVI called **RDTSC.vi**, which is merely a call to some external assembly-level code that reads the time-stamp count register on Pentium processors. RDTSC returns two 32-bit numbers corresponding to the upper and lower halves of a Pentium processor’s 64-bit time stamp counter register. Each time the Pentium clock generates a new clock edge, the time stamp counter register increments once. By tracking the counter register each time our loop executes, we can extract valuable timing information with a resolution equal to 1/CPU speed. For our fastest controller, which has an 850 MHz processor, we can measure loop rates with nanosecond timing resolution.

When timing code with RDTSC, it is best to ensure that RDTSC executes at exactly the same time with respect to the rest of the code inside your loop to ensure accurate timing data is collected. To achieve this effect, we recommend you wire RDTSC to your code such that RDTSC is either the last node of the first node to execute within your loop.
The remaining parts of the diagram convert RDTSC data to actual times based on the CPU speed. The timing data is then displayed on a graph with cursors indicating the mean loop execution time and standard deviation.

4. Use RDTSC Timing Shell to time simple multiplication.
5. Time Cont Acq&Chart (hw timed) VI using the RDTSC Timing Shell.
Lesson 5

Shared Resources

You Will Learn:
A. About different types of shared resources
B. About priority inversions and priority inheritance
C. About communicating between VIs using queues and functional global variables
Shared Resources

A shared resource in LabVIEW RT is anything that can only be used by one process at a time.

LabVIEW RT shared resources include:

– Global variables
– LabVIEW RT Memory Manager
– Non-reentrant subVIs
– Single-threaded DLLs
– Semaphores
– Networking code (TCP/IP, UDP, VI Server)
– File I/O

Certain data structures, driver libraries, and variables can only be accessed serially, one process at a time. A simple example of a shared resource common to all programming languages is the global variable. As you know, global variables cannot be accessed simultaneously from multiple processes to prevent race conditions. Therefore, compilers automatically protect the global variable as a shared resource while one process needs to access it. In the meantime, if a second process tries to access the global variable while it is protected, the second process must wait until the first process has finished with the global variable. As we will soon see, understanding shared resources and how to identify them is an important skill when programming real-time applications.

**Note:** These operations are inherently non-deterministic and should never be used inside a *time-critical priority* loop if you’re attempting to achieve real-time performance.
Before a process can begin using a shared resource, it must obtain a mutual exclusion or mutex.

After Process 1 finishes, Process 2 can proceed.

For our purposes a shared resource is defined as a software object that can only be used by one thread at a time. In this slide, let’s assume both processes constantly need to access the shared resource. Let’s also assume Process 1 is running at normal priority, while Process 2 runs at time-critical priority. Normally, when a time-critical thread needs to execute, it preempts all other code running on the real-time target; however, a normal priority thread can block the time-critical thread if it has not released a mutex that the time-critical thread needs. This is known as a priority inversion because the real-time operating system cannot allow the time-critical thread to preempt the normal priority thread, merely because of the mutex around a shared resource.

A typical scenario of a priority inversion caused by a shared resource follows:

- When a lower priority thread needs a resource such as the LabVIEW Memory Manager, it protects the resource from external access while in use. The lower priority thread has acquired a mutex and begins allocating memory.

- Now, a time-critical priority thread wakes up while the lower priority thread is using the resource and kicks the lower priority thread off of the processor.

- Then the time-critical priority thread wants to use the same protected resource (i.e., wants to allocate memory), but is forced to wait because of the protection (the mutex) around the resource. This is an inversion of priorities; a higher priority thread is forced to wait because a lower priority thread has protected a resource the higher priority thread needs.
In this slide the shared resource is a global variable, which is shared by two VIs: one set to normal priority and one set to time-critical priority.
Priority Inversion

Time Critical Priority - waiting

Priority inversion:
Normal priority VI blocks the higher priority VI with a mutex around the shared resource.

Normal Priority - running

Priority inheritance:
Normal priority VI inherits higher priority to release mutex.

The real-time OS uses a method called priority inheritance to resolve the priority inversion as quickly as possible. It does this by:

- Allowing the lower priority thread to temporarily “inherit” the time-critical priority setting long enough to finish using the shared resource and to remove the protection.
- Once the protection is removed the lower priority thread resumes its original lower priority setting and is taken off of the processor.
- Now the time-critical priority thread is free to proceed and use the resource (i.e., access the global variable).

A result of this priority inversion is increased software jitter in the time-critical priority thread. The jitter induced by a protected global variable is generally quite small compared to the jitter induced by protecting the LabVIEW Memory Manager. Unlike accessing global variables, performing memory allocations is unbounded in time and can introduce a broad range of software jitter while parallel operations try to allocate blocks of memory in a wide variety of sizes. The larger the block of memory to be allocated—the longer the priority inheritance takes to resolve the priority inversion.
Sharing a Global Variable

Only one loop at a time has access to the global variable “stop.”

1) Top loop acquires a mutex to the global and writes to it.

2) Lower loop becomes locked while waiting to acquire mutex.

3) Top loop releases mutex around global, allowing lower loop to proceed.

The slide animation depicts one loop acquiring a mutex to the global variable, blocking the other loop for a brief moment. The global variable is a very common type of shared resource, which can cause problems for real-time applications. Every shared resource conflict inevitably manifests itself as software jitter. Therefore, we must learn how to avoid priority inversions and shared resource conflicts in general to minimize software jitter.
Functional Global Variables (FGV)

FGVs can have several inputs and outputs.

FGVs can be “skipped if busy.”

Instead of using regular global variables, LabVIEW Real-Time programmers should employ functional global variables to minimize software jitter. A functional global variable (or LabVIEW 2-style global) is merely a subVI set at subroutine priority that contains a WHILE loop with a nested case structure for read or write access. FGVs always run once when they are called and store data in their shift registers, which allow subsequent FGV calls to access the most recent data.
A time-critical thread should be allowed to skip the FGV call.

When skipped, the FGV returns default values for each data type.

The “skip if busy” setting ensures that the time-critical thread skips the FGV if another process is currently accessing the same FGV. When skipped, the FGV returns default values for each data type on the FGV diagram: numerics return 0, Booleans return false, etc. The FGV images used in the slides depict only one set of shift registers for “A” and “B.” FGVs are similar to queues because you can add more shift registers to store a longer history of values of “A” and “B.” You can also add more variables to the FGV, to store “C,” “D,” “E,” etc. An advantage of FGVs over normal global variables is that FGVs help prevent shared resource conflicts while still offering the same functionality as normal global variables. A disadvantage to FGVs is that you may lose data during inter-thread communication when the FGV gets skipped inside the time-critical thread.
Exercise 5.1

Students measure software jitter incurred from global variables and learn how to use Functional Global Variables (FGVs) for inter-thread communication.
Sharing SubVIs

You can configure subVIs for reentrancy to avoid shared resource conflicts.

1) Top loop acquires a mutex to the VI and writes to it.

2) Lower loop becomes locked while waiting to acquire mutex.

3) Top loop releases mutex around VI, allowing lower loop to proceed.

As you saw in the previous slides, sharing subVIs cause priority inversions the same way global variables do. When a subVI is set to subroutine priority, that subVI can be “skipped” within time-critical code to avoid software jitter that would have occurred from a priority inversion. If you are running unrelated parallel processes that call the same subVI, you can configure the subVI for reentrant execution. A reentrant subVI establishes a separate data space in memory each time it is called. This feature allows LabVIEW RT to call several instances of a particular subVI simultaneously. Because reentrant subVIs use their own data space, however, they cannot be used to share or communicate data between threads. You should only use reentrancy when you must simultaneously run two or more instances of a subVI within unrelated processes that do not need to share data within the reentrant subVI.
Reentrant SubVIs

Configure reentrancy in the Execution window under VI Properties.

Reentrant subVIs do NOT act like global variables!

To configure a subVI for reentrancy, go to **VI Properties » Execution**, and then check the box “Reentrant Execution.” Be careful when using this feature because each call to the subVI establishes a new unique data space for all controls and indicators in memory. Reentrant subVIs cannot be used as LabVIEW 2-style global variables for this reason. You should also be aware that configuring the DAQ VIs for reentrancy does not solve shared resource problem with NIDAQ32.dll. As we will soon see, all DAQ VIs load the DAQ driver, NIDAQ32.dll. This dll is not multithreaded safe, which means it cannot be called simultaneously from multiple processes.
Sharing the DAQ Driver

NI-DAQ32.dll is single-threaded, making it a shared resource.

1) Top loop acquires a mutex to the DLL.

2) Lower loop becomes locked while waiting to acquire mutex.

3) Top loop releases mutex around DLL, allowing lower loop to proceed.

As mentioned in the previous slide, NIDAQ32.dll is not multithread safe. This has serious implications for parallel DAQ processes because the DAQ driver can only be accessed by one process at a time, much like a global variable. Therefore, you must architect your DAQ applications carefully to avoid parallel DAQ calls like the one in this slide. The recommended method for architecting a DAQ application is to call DAQ VIs in only one loop, serially. You can ensure that DAQ VIs are called serially by routing one error cluster through every DAQ VI in the loop.
Data Acquisition

Key points to remember about DAQ and real-time:

- The NI-DAQ driver is a shared resource.
- All data acquisition VIs call NIDAQ32.dll.
- All DAQ calls should be made serially from within a single loop within one thread.
- To achieve real-time, the DAQ loop must be given time-critical priority.

The only way to achieve true real-time DAQ, is to call all DAQ VIs from within the same loop inside the same thread, which must be given time-critical priority.
LabVIEW RT Memory Manager

- LabVIEW RT manages memory automatically.
  - The user does not explicitly have to allocate or de-allocate memory.
  - This means memory management is easy but harder to control.
- LabVIEW RT has a “Memory Manager,” which is a shared resource.
  - We must control memory allocations to avoid shared resource conflicts with the memory manager.
  - Statically allocate memory before time-critical process begins.

LabVIEW RT has a memory manager, which automatically allocates and de-allocates memory at run-time. Allocating memory can consume a significant amount of CPU time, depending on the amount of memory needed. If we allow LabVIEW RT to dynamically allocate memory at run-time, our application could suffer from software jitter for two reasons:

1. The memory manager may already be mutexed by another process, causing a shared resource conflict.
2. Even if the memory manager is immediately available, allocating memory is inherently non-deterministic because there is no upper bound on the memory allocation’s execution time.

To avoid shared resource conflicts or, more specifically, priority inversions, with the LabVIEW RT memory manager, statically allocate memory before any real-time loops. For example, instead of using Build Array within your loop to index new data into an array, use Replace Array Subset to insert new array values into pre-allocated arrays. We will learn how to do this shortly.
Sharing the Memory Manager

Some LabVIEW RT functions allocate memory and must therefore access the memory manager.

1) Top loop acquires a mutex to the memory manager.

2) Lower loop becomes locked while waiting to acquire mutex.

3) Top loop releases mutex around memory manager, allowing lower loop to proceed.

The slide features two parallel loops. The top loop dynamically builds an array using the Build Array function, while the lower loop builds the same array using auto-indexing.
Pre-allocate Arrays

Avoid shared resource conflicts by pre-allocating arrays.

Initialize Array before entering the loop.

Replace Array Subset to insert new elements.

Initialize Array in combination with Replace Array Subset allows you to pre-allocate arrays inside a loop to avoid shared resource conflicts with the LabVIEW RT memory manager.
Memory Management

Key points to remember about memory management and real-time:

- The LabVIEW RT memory manager is a shared resource.
- All memory allocations must be removed from the time-critical loop to guarantee robust, hard real-time performance.
- Pre-allocate arrays outside of the time-critical loop.
- Cast all data to the proper data type.
- Use inplaceness when possible to reuse memory buffers.

In general, memory allocations within a time-critical loop will induce jitter and affect the deterministic properties of a LabVIEW RT program. All memory allocations must be removed to guarantee robust hard real-time performance. You must pre-allocate your arrays outside of the loop if you want your application to run deterministically. Certain LabVIEW RT functions, such as Build Array and Bundle, allocate memory.

Also try to cast data to the proper data type. Every time LabVIEW RT has to perform a type conversion, either implicitly or explicitly, a copy of the data buffer is made in memory in order to retain the new data type after the conversion. Use the smallest data type possible, and if you need to convert the data type of an array, try to do that conversion before the array is built. Also keep in mind that many LabVIEW RT functions reuse their input buffers when the output data representation, size, and dimensionality matches the input. The same structure and number of elements. This ability for a function to reuse buffers is called inplaceness.

Note: More information about inplaceness and LabVIEW memory management can be found in the LabVIEW Advanced Course.
Exercise 5.2

Students learn how to pre-allocate arrays and measure software jitter incurred from dynamic memory allocations.
Shared Resources and Handshaking

Avoid these functions inside time-critical code:
- Semaphores
- Networking functions (TCP/IP, UDP)
- File I/O

Avoid handshaking protocols too:
- GPIB
- RS-232 with h/w or s/w handshaking
- DAQ handshaking
- Any other handshaking protocol…

We did not cover all types of shared resources in this chapter. Avoid semaphores, TCP/IP, UDP, VI Server, and File I/O functions within your time-critical loop. These functions are inherently non-deterministic and use shared resources. For example, semaphores are themselves shared resources, network functions use the Ethernet driver, and file I/O functions use the hard disk. These functions can introduce severe software jitter in time-critical code on account of priority inversions.

Also be aware that all handshaking protocols are non-deterministic. GPIB, RS-232, and TCP/IP are just a few things that should never be run at time-critical priority. DAQ handshaking protocols like burst mode and 8255 emulation mode on the 653x boards are also non-deterministic and should be avoided in time-critical loops.
Real-Time Queues (RTQ)

Real-Time Queues offer a safe method of inter-thread communication.

Real-Time Queues are like the regular LabVIEW Queues except they pre-allocate enough memory required to store up to “n” elements of a user-defined numeric representation (i.e., DBL, SGL, U32, etc.). As we learned earlier, pre-allocating memory is important because dynamic memory allocations within a time-critical loop can add a significant amount of software jitter. Each RT Queue allows users to push and pop elements simultaneously because RT Queues actually consist of two queue copies in memory. Simultaneous read and write access is a necessary feature of RT Queues to prevent shared resource conflicts. Like FGVs, RT Queues allow you to safely pass data between threads. A benefit to using RT Queues instead of FGVs is that you can pass data between threads without losing any data. Recall that FGVs become “lossy” when they are skipped by the time-critical thread. One disadvantage to RT Queues is that data cannot be accessed arbitrarily or in blocks. RT Queue elements must be popped from the queue one element at a time.
Exercise 5.3

Students learn how to use RT Queues for inter-thread communication.
Chapter 5—Summary

A shared resource is a software object that can only be used by one thread at a time.

- LabVIEW Memory Manager
- Global variables
- Non-reentrant subVIs
- Synchronization code
- Networking protocols
- File I/O
Chapter 5—Summary (Continued)

To avoid priority inversions:
- Pre-allocate arrays before entering loop
- Keep all DAQ calls in one time-critical thread
- Keep all networking and file I/O calls in a separate normal priority thread

To communicate data between threads:
- Use Functional Global Variables (FGVs)
- Or use RT Queues

Avoid shared resources and handshaking within time-critical code.
Exercise 5-1
Objective: To transfer data between threads using a standard global variables and functional global variables.

PART 1.
1. Launch LabVIEW Real-Time and select the appropriate RT Series device from the Select Target Platform dialog box. Close any VIs already running on the target platform.
2. Select File»New, and create a new VI set for normal priority that writes to a global variable.

3. Select File»Open and open RDTSC Timing Shell VI from Exercises\Chapter 5\RDTSC Timing Shell.llb.
4. Now open the RDTSC timing shell and modify it to write to the same global variable. Add a small wait statement.
5. Run the timing shell while `globalWriter VI` is running in the background.
6. Run the timing shell while `globalWriter VI` is idle and compare the timing differences.

The timing shell should reveal slightly more jitter when `globalWriter VI` is running in the background. If the global were much larger than a simple Boolean, such as a large array or cluster, the jitter would be much more pronounced.

7. Try other types of global variables to see what effect they have on jitter.
Try keeping the user interface for the global VI open to see the effect on jitter.

PART 2.
1. Select `File»Open` and open `Normal Priority Loop VI` from `Exercises\Chapter 5`.
2. Examine the block diagram and study `LV2Global VI`.

A Functional Global, also called a LabVIEW 2-Style Global, is a VI that uses a loop with uninitialized shift registers to hold global data. A Functional Global usually has an input parameter that specifies the VI function, such as writing or reading. `LV2Global VI` is a Functional Global, with its priority set to subroutine. When called, the subroutine receives 100% of the CPU time allocated to its calling thread. While `LV2Global VI` is running, no other processes in the same thread can execute. A benefit to using subroutine priority is that your code can execute very quickly. Another benefit to marking a subVI for subroutine priority is that you can skip over a call to the subroutine if it is currently being used in another thread (i.e., if it is a shared resource). This option is called “Skip Subroutine Call If Busy.”
3. Open Time Critical Loop VI. Copy the block diagram code into the RDTSC Timing Shell so we can benchmark it. Make sure the “Skip Subroutine Call If Busy” option is checked when you right-click on LV2Global VI.

4. Run Normal Priority Loop VI without modifications. Then run the timing benchmark VI you’ve created.
5. Now run the benchmark while the Normal Priority Loop is idle and compare the timing differences. Generally, you’ll expect to see little difference in the jitter when using Functional Globals because the time-critical VI will skip the call if the subroutine is busy.

What are the timing differences between global variables and Functional Globals? Which case had the least jitter?

Exercise 5-2
Objective: To examine the LabVIEW RT Memory Manager as a shared resource using the RDTSC Timing Shell.

1. Launch LabVIEW Real-Time and select the appropriate RT Series device from the Select Target Platform dialog box. Close any VIs already running on the target platform.

2. Select File»Open and open RDTSC Timing Shell VI from Exercises\Chapter 5\RDTSC Timing Shell.lib. Also open Background Memory Allocation.vi.

PART 1.

3. Modify the RDTSC Timing Shell VI to build an array using the Build Array function. This will cause the array to grow in size while the VI is running. Use the Initialize Array function to start the array with a size of 10,000. Add a small wait statement to the loop as well.

4. Run the Background Memory Allocation VI. Then run the timing benchmark you’ve created.
5. Try running the timing benchmark with the Background Memory Allocation VI stopped. Notice that the maximum jitter was greater when the memory allocation VI was running.

You should avoid allocating memory within your time-critical code for two reasons. First, other background memory allocation can cause a priority inversion, which adds jitter to your loop. Second, the process of allocating memory is not bounded in time. The time required to resolve a memory allocation depends on the amount of memory requested and the amount of free memory available. As you saw in this example, the amount of requested memory grows as we build larger arrays while the amount of available memory diminishes.
PART 2.

1. Modify RDTSC Timing Shell VI again to build an array using LabVIEW’s auto-indexing feature instead of the build array function. Add a small wait statement to the loop.

2. Run the Background Memory Allocation VI. Then run the timing benchmark you’ve created.

3. Try running the timing benchmark with the Background Memory Allocation VI stopped. Notice that the maximum jitter was not any greater when the memory allocation VI was running. Memory is allocated before the For Loop begins when using auto-indexing.

What conclusions can you draw from the timing results? How are memory allocations handled differently with auto-indexing turned on than they are with the Build Array function?
PART 3.

1. Modify RDTSC Timing Shell VI again to pre-allocate an array using Initialize Array before entering the loop. Use Replace Array Subset to insert random values into the initialized array. Include a small wait statement inside the loop.

2. Run the Background Memory Allocation VI. Then run the timing benchmark you’ve created.

3. Try running the timing benchmark with the Background Memory Allocation VI stopped. Notice that the maximum jitter was not any greater when the memory allocation VI was running. Memory is allocated before the For Loop begins and the Replace Array Subset VI does not invoke the memory manager.

   How does performing memory management in a time-critical VI affect the timing?

Exercise 5-3
Objective: To transfer data between threads using RT queues.

1. Select File»Open and open Simple RT Queue Example – Time Critical Loop VI from Exercises\Chapter 5\RT queue\Simple RT Queue Example - Time Critical Loop.llb.
2. Examine the block diagram and priority of this VI.
This VI consists of two queues—Control Data Queue and Result Data Queue. Simple RT Queue Example – Time Critical Loop VI reads two elements from Control Data Queue and stuffs one element into Result Data Queue.

3. Now select **File»New**, and build the following block diagram. Use the RT Queue VIs from RTQ.llb.

![Simple RT Queue Example VI Diagram](image)

The subVI node labeled “Real Time Loop” in the above illustration is merely Simple RT Queue Example – Time Critical Loop VI. Simple RT Queue Example VI pushes two new elements (a stop flag and amplitude) into Control Data Queue and pops one element (sine data) from Result Data Queue. When Result Data Queue has no new data available, the case structure switches to TRUE and does nothing.

RT queues are superior to standard LabVIEW queues because they pre-allocate memory based on queue size, number of queue elements, and the data representation for each queue element. For example, Control Data Queue defaults to a size of 10 elements and stores two DBLs per queue element. Given these parameters, LabVIEW Real-Time pre-allocates 160 bytes
(8 bytes per DBL * 2 DBLs per element * size of 10 elements). By pre-allocating queue memory, we avoid dynamic memory allocations, which are known to induce jitter.

RT queues are also superior to normal global and functional global variables because RT queues notify the user when they have lost data with the Boolean parameter **overwrite**. When writing to global variables faster than reading them, you will inevitably lose data, but there is no easy way to determine when exactly this has happened.

RT queues can be complicated to new LabVIEW programmers. This drawback, however, is generally outweighed by the advantages, which make RT queues the preferred method for transferring data between threads within LabVIEW Real-Time.
Lesson 6

Other Features of LabVIEW Real-Time

You Will Learn:
A. How to automatically restart your application if the RT target or host computer reboots
B. About the FTP server and Web server running on RT PXI and FP-2000 targets
C. How to implement hardware and software watchdogs
LabVIEW Real-Time’s Application Builder compiles VIs into executables.

The LabVIEW Real-Time Application Builder compiles VIs into executables. LabVIEW Real-Time executables can be automatically downloaded and run on RT targets, which is an important feature for some users.
The application builder consists of several tabs. The tabs are used to target a VI for compilation, configure exe settings, and even create an installer for the RT engine.

We will not cover the details of Application Builder in this class. To learn more about the Application Builder settings, please review LabVIEW Basics II.
App. Builder While Targeted

Application is downloaded to RT Device after it is built.

When targeted to a networked RT Series Device, after the application is built, it is automatically downloaded to the Destination directory on the RT Engine.
Configuring RT Executables

When targeted to an RT PXI Controller, the menu item “Network: Options” under Tools becomes enabled.

Check the “Launch Application at Boot-Up” box to automatically launch exes when the RT target reboots.

While targeted to an RT Series networked device, LabVIEW Real-Time offers special menu items under Tools»Network: x.x.x.x Options. One of the network options is RT Target: Miscellaneous. Selecting this option allows you to specify an application path, which is the path our executable will be downloaded to. Also, if the Launch Application at Boot-Up option is checked, the downloaded application will automatically run each time the PXI device reboots.

To download an executable to an RT Engine immediately after building it, simply invoke the Application Builder while targeted to an RT target. The executable’s destination directory will change to the RT Engine’s network path instead of the local machine. Upon rebooting the device, the application will automatically launch.

To turn off this feature, simply target LabVIEW Real-Time to the RT Engine, go to Tools»Network Options»RT Target: Miscellaneous and deselect the check box Launch Application at Boot-Up.
Use command-line arguments within a shortcut to target an exe. Use “-target,” “-quithost,” or “-reset.”

Rather than downloading an executable to the RT-PXI hard disk, you can keep the executable on the Host PC and run it targeted to the RT-PXI using a command line arguments. This slide depicts the properties page of a shortcut to an executable called Charts.exe. If we launch the shortcut, it will download and run Charts.exe on the RT-PXI target at 130.164.48.131. By simply adding “-target” at the end of the executable’s path, LabVIEW RT knows to download the executable to the RT-PXI target and run it. The command line argument “-quithost” closes the user interface to the executable on the Host PC and the communication thread on the RT-PXI. To reset the device before or after running and executable, use “-reset.”

If the Host PC crashes while an executable is running on the RT Engine with a user interface on the Host PC, if you later relaunch the executable targeted to the RT Engine, the Front Panel will synchronize with the code still running on the RT Engine, and continue to provide the user interface.
Batch files and Windows shortcuts can be placed in the `~\startup` directory to automatically run a targeted exe if the Windows host reboots.

Batch files offer an easy method of launching multiple executables to various targets at once. You can create a batch file by simply opening a new text file with Notepad and saving the document with the extension `.bat`. Inside the batch file, type command lines as though you were at a command prompt, supplying the full path and name of the executable(s) you wish to launch, followed by any command line arguments. Batch files and shortcuts are very handy when placed in the `~\Windows\startup` folder, because they allow you to automatically download and run executables to any target if the Host PC loses power.
Exercise 6.1

Students build an RT executable to automatically start after power failure.
The PXI and FP-2000 Controllers run an FTP server. Log on initially with no user name or password.

Logging on to the RT-PXI’s FTP server can be done from a command prompt or from a Web browser window. In this slide, we are demonstrating how to log onto the FTP server from the command prompt. To transfer files, you must have some knowledge of FTP commands, which can be displayed by typing “?” at the FTP prompt. Be sure to use DOS 8.3 file name format when specifying the path to the file you wish to transfer. Example:

ftp> put c:\windows\progra~\nation~\labview\charts.vi

You should not exceed 8 characters per directory when specifying the path name.
FTP transfer is easier with a Web browser if you are not familiar with FTP commands.

Simply type the IP address preceded by `ftp://` in the web browser window.

If you are unfamiliar with FTP commands and feel more comfortable using a web browser, simply launch Internet Explorer and type the IP address to the RT-PXI Controller.
Exercise 6.2

Students transfer files between host PC and RT Series PXI Controller using FTP.
Web Server (PXI and FP-2000 only)

The PXI and FP-2000 Controllers also run a Web server, which can be accessed with a web browser like Internet Explorer.

While targeted to a PXI or FP-2000 Controller, you can enable the web server by going to Tools » Network Options.

The RT-PXI also runs a Web server, which must be enabled through LabVIEW RT while targeted to the RT-PXI Controller. Go to Tools » Network Options, and check Enable Web Server. The default web server path is c:\NI-RT\system\WWW. You can place image files, HTML files, etc. in this directory on the RT-PXI and then browse to them remotely from a web browser.
Once the PXI or FP-2000 Web server has been enabled, you can browse .html files or images across a network using a web browser.

By saving data to an .html file within LabVIEW Real-Times’ web server, you can publish your data across the network formatted in HTML.

The web server is a powerful tool on the RT-PXI because you can incorporate real-time control and file I/O to publish your results over the Web.
Exercise 6.3

Students browse RT Series Networked Device’s Web server
Watchdogs

- The RT Series PXI Controllers and FP-2000 Controllers have built-in watchdogs
  - Programmable through watchdog VIs
  - Can reset controller or set occurrence when watchdog expires
  - Can activate trigger on PXI back plane
- The FP-2000 is also equipped with a network watchdog to respond to a connection failure
  - Disabled by default
  - Enable through FieldPoint Explorer

The point to having a software watchdog is to disable a critical system in the event of a software crash. The software watchdog on the PXI Controller is actually a counter, which can trigger several different expiration actions if it expires, including resetting the PXI Controller, setting an occurrence in software, and activating a PXI back plane trigger.
Watchdog Timer

Expires when not whacked in time.

The RT PXI and FP-2000 Controllers can respond to an expired watchdog by:

- Resetting
- Setting a software occurrence
- Setting an external trigger line

The watchdog API can be found under the RT functions palette. The top-level watchdog API consists of three main VIs—Configure Watchdog, Whack Watchdog, and Clear Watchdog.
Exercise 6.4

Students implement a watchdog timer on the RT Series PXI Controller.
Software Watchdogs II

First enable the network watchdog on the network module
Next enable network watchdog for each output module.
Finally, specify channels and output response to the network watchdog.
Other Types of Watchdogs

- Other software watchdogs
  - Increment a byte of shared memory and monitor “heartbeat” on host or RT PXI
  - Use TCP/IP or VI Server to increment a variable and monitor “heartbeat”

- Hardware watchdogs
  - Toggle digital line on a DAQ device and monitor line state on RT PXI or external device
  - Perform single pulse generation with a counter and reprogram counter within a loop
  - Perform pattern generation on 653x device and reset the buffer before it finishes

In addition to the built-in watchdog available on the FP-2000 and RT PXI Controllers, programmers can implement their own software watchdog using shared memory (7030 only), TCP/IP, or VI Server. The RT examples have an example that implements a software watchdog through shared memory on the 7030 boards.

Besides software watchdogs, users can also implement hardware watchdogs to alert other hardware if your real-time application crashes or fails. Counters are especially useful for watchdog applications because they behave very similar to PXI watchdog and can be programmed similarly too. A less obvious watchdog solution involves the 653x devices. By configuring a 653x device for pattern output, you can output a buffer of known length at some user-defined frequency, and reset the buffer in a software loop before the device reaches the end. If the device reaches the last pattern, for example, switches from 0 to 255, we can deduce from the digital lines on the 653x device that the software has failed to reset the buffer on time.
Exercise 6.5 (Optional)

Students implement a hardware watchdog on an E-Series device and on a 653x device.
Chapter 6 - Summary

- Applications can be embedded or downloaded
- FTP Server and Web Server available
- Hardware and Software Watchdogs for safety
Exercise 6-1

Objective: To build an RT executable that automatically starts itself after power failure.

1. Launch LabVIEW Real-Time and select the appropriate RT Series device from the Select Target Platform dialog box. Close any VIs already running on the target platform.
2. Select File»Open and open MyApp VI from Exercises\Chapter 6.
3. Examine the diagram and run the VI on the target, watching the signal accessory. Be sure to select the appropriate DAQ device number and make it the default value. (Note: If you fail to make the DAQ device number the default value, later steps will result in errors.)
4. Save the VI.

PART 1. (PXI only)
1. Close the VI from memory and open a new VI while still targeted to your PXI Controller.
2. Select Tools»Network Options…
3. Select RT Target Miscellaneous and take note of Application Path, which defaults to c:\ni-rt\startup\startup.exe.

4. Check the box labeled Launch Application at Boot-Up, and then click OK.
5. Next select Tools»Build Application and click on the Source Files tab.
6. Add MyApp VI as the top-level VI.
7. Click on the **Application Settings** tab, and uncheck the option to Show **LabVIEW Real-Time** target selection dialog box when launched.

8. Click **Build**, then click **Done**.

9. When prompted to create script file, choose **No**.

We have just compiled MyApp VI into an executable. Because we built the executable while targeted to our PXI, the executable was downloaded to `c:\ni-rt\startup\startup.exe` on the PXI’s hard disk. Next time the PXI reboots, the executable will automatically begin running.

10. Simulate a power failure by recycling power on the PXI controller then monitor the signal accessory as the PXI boots up.
PART 2. (For RT Series DAQ devices, although the exercise can be completed with an RT Series PXI controller.)

1. Launch LabVIEW Real-Time targeted to the host computer.
2. Open a new VI and select **Tools»Build Application**.
3. On the **Target** tab, enter the following information in each field:

   ![Build Application or Shared Library (DLL) - New script *](image)

   - **Build target**: Application (EXE)
   - **Target file name**: MyApp.exe
   - **Destination directory**: c:\WINDOWS\desktop
   - **Support file directory**: c:\WINDOWS\desktop\data

4. Then follow the remaining steps as above, adding MyApp VI as the top-level VI and building the application.

   This time we compiled the VI while targeted to the host computer. (This is needed for RT Series DAQ Devices because the processor board has no hard drive on which to store the executable.) We will need to use command-line arguments to configure the executable to target itself to the RT device. We will also need to put the executable in our Windows startup directory.

5. Once you have built the VI into an executable, it should be located on your desktop, labeled MyApp.exe.
6. Right-click on it and choose **Create Shortcut**.
7. Right-click on the shortcut, and choose **Properties**.
8. In the field labeled **Target**, add a command-line argument to target the executable to the 7030 device (i.e., -target DAQ::1, or -target [IP Address]).
9. Next place the shortcut in the Window startup folder, which may be located at C:\WINDOWS\All Users\Start Menu\Programs\StartUp.
10. Power off the desktop PC to simulate power loss and watch the signal accessory (which should now be attached to your DAQ device) as the PC boots up again.

**Note:** For users of Windows NT, you can download a utility called TweakUI.exe from Microsoft, which allows you to configure Windows NT to automatically log in upon bootup. Also, the paths may be different for Windows NT.
Exercise 6-2
Objective: To become familiar with the LabVIEW Real-Time FTP server on PXI controllers.

PART 1.
1. On your Windows PC, click **Start»Run**, then type `ftp 130.164.48.22`, except use the IP address of your controller instead of `130.164.48.22`.
2. When prompted for a username and password, just hit **Enter**.

![ftp command prompt](image)

3. Once you have reached the ftp command prompt, type “?” and then hit **Enter**.
4. After studying the list of ftp commands, try using the **put** and **get** commands to transfer files back and forth. You may need to change local directory with **lcd**, or change target directory with **cd**. You can list the contents of the remote working directory with **ls**.
PART 2.
1. Launch a Web browser, such as Internet Explorer.
2. Enter ftp://130.164.48.22 (use your own IP address here) in the address field, and then hit Enter.

3. You should have immediate access to the remote hard drive via the Web browser, so try dragging and dropping contents to transfer files.

Exercise 6-3
Objective: To become familiar with the LabVIEW Real-Time Web server on PXI controllers.

1. Launch LabVIEW Real-Time and select New VI.
2. Select Tools»Network Options…
3. Select Web Server: Configuration and check the box labeled Enable Web Server. Take note of the web server’s root directory, which can be modified.
5. Locate image.gif under \Exercises\Chapter 6 on the host PC running Windows.
6. Open an FTP connection to your PXI and transfer image.gif to ftp://130.164.48.22/NI-RT/SYSATEM/WWW/
7. Open a web browser and navigate to http://130.164.48.22/image.gif.

Exercise 6-4
Objective: To implement a watchdog timer, which will reboot the PXI controller when the watchdog expires.

1. Complete Part 1 of Exercise 6-1 above.
2. Select File » Open and open Watchdog Reset – RT Engine VI from c:\Program Files\National Instruments\LabVIEW 6\Examples\RT\RT Watchdog (PXI-8156B).llb.
3. The block diagram comprises 3 subVI nodes: Watchdog Configure, Watchdog Whack, and Watchdog Clear. Watchdog Configure configures and arms a hardware counter/timer built into the RT Series PXI Controller to count down for a period equal to the timeout input parameter. Once the internal counter has expired (i.e., counted past the timeout period), the Controller responds according to the expiration actions. In this example, we will choose to reset the PXI Controller.
Watchdog Whack simply resets the watchdog timer back to zero each time it is called. We can conclude then that by calling Watchdog Whack more frequently than the period set up in Watchdog Configure, the watchdog counter will not expire. The point to having a software watchdog is to disable a critical system in the event of a software crash.

4. Study the front panel of Watchdog Reset – RT Engine VI. Notice the watchdog timeout period is 0.25 seconds and the loop period is set for only 100 milliseconds or 0.10 seconds, which ensures that Watchdog Whack will have plenty of time to reset the watchdog counter before it resets.
5. Download the VI to the RT PXI target and run it.
6. Slowly increase the loop cycle time by adjusting the Wait control on the front panel.
7. Once you reach 250+ millisecond loop periods, notice the PXI Controller reboots itself.
8. Also monitor the signal accessory, which should be blinking as it did in Exercise 6-1 once the controller boots up again.

Exercise 6-5 (Optional)
Objective: To implement a hardware watchdog using pulse train generation (Part 1) and pattern output (Part 2).

Note: This section requires knowledge of advanced DAQ programming and timing concepts and a 653x board for PART 2.

PART 1.
1. Launch LabVIEW Real-Time and select the appropriate RT Series device from the Select Target Platform dialog box. Close any VIs already running on the target platform.
2. Select File » Open and open Watchdog Example VI from Exercises\Chapter 6\counterWatchdog.llb.
3. Study the diagram, which is divided into three simple parts.

4. The first subVI sets up a DAQ counter on either a 660x device or an E-Series device to perform pulse generation.
5. The second subVI resets the counter within the WHILE loop.
6. Finally, the last subVI clears and disarms the DAQ counter.
   This example is almost identical to Watchdog Reset – RT Engine VI because both examples configure a counter to perform a task after counting down some time period. The WHILE loops in both examples reprogram the counter to start counting down again, and if the WHILE loops are not executed on time, the counter performs the watchdog expiration task. In this case, however, the watchdog merely outputs a pulse rather than resetting the PXI controller or setting an occurrence.
7. Download and run Watchdog Example VI using a Watchdog Delay of 5 seconds and a Loop Delay of 3 seconds.
8. Slowly increase the loop delay to 5+ seconds while monitoring the output of counter 0 on the DAQ device.

PART 2.
1. Launch LabVIEW Real-Time and select the appropriate RT Series device from the Select Target Platform dialog box. Close any VIs already running on the target platform.
2. Select File»Open and open digWatchdog VI from Exercises\Chapter 6\digWatchdog.l1b.
3. Study the diagram, which contains mostly advanced digital I/O functions.
The example initializes a buffer of several 0’s followed by a 1. The example programs a 653x device to output this buffer on one digital line at a user-define rate. The WHILE loop simply programs the device to restart the output at the beginning of the buffer. If the WHILE loop executes more frequently than device can output one entire buffer, the device will never reach the last pattern in the buffer (i.e., the board will never change the output state on line 0).

On the other hand, if the loop dies or slows down significantly, the 653x device will reach the end of the buffer and output a 1 or logic high. This voltage change could be tied to a relay or some other external circuit to shut down a critical system in the event of a software failure.

4. Download and run the VI with a clock frequency of 10 Hz and a sleep time of 9 seconds or 9000 ms.
5. While the VI is running, monitor the state of digital line 0 on the 653x device. It should be 0 volts.
6. Now increase the sleep time to 10+ seconds. What happens to digital line 0?