LabVIEW™ Real-Time 1 Course Manual

Course Software Version 2010
September 2010 Edition
Part Number 373246A-01

Copyright

©2009–2010 National Instruments Corporation. All rights reserved.
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.

National Instruments respects the intellectual property of others, and we ask our users to do the same. NI software is protected by copyright and other intellectual property laws. Where NI software may be used to reproduce software or other materials belonging to others, you may use NI software only to reproduce materials that you may reproduce in accordance with the terms of any applicable license or other legal restriction.

For components used in USI (Xerces C++, ICU, HDF5, b64, Stingray, and STLport), the following copyright stipulations apply. For a listing of the conditions and disclaimers, refer to either the `USICopyrights.chm` or the *Copyrights* topic in your software.

Xerces C++. This product includes software that was developed by the Apache Software Foundation (`http://www.apache.org/`). Copyright 1999 The Apache Software Foundation. All rights reserved.

ICU. Copyright 1995–2009 International Business Machines Corporation and others. All rights reserved.

HDF5. NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities
Copyright 1998, 1999, 2000, 2001, 2003 by the Board of Trustees of the University of Illinois. All rights reserved.

b64. Copyright © 2004–2006, Matthew Wilson and Synesis Software. All Rights Reserved.

Stingray. This software includes Stingray software developed by the Rogue Wave Software division of Quovadx, Inc.
Copyright 1995–2006, Quovadx, Inc. All Rights Reserved.

STLport. Copyright 1999–2003 Boris Fomitchev

Trademarks

CVI, LabVIEW, National Instruments, NI, ni.com, the National Instruments corporate logo, and the Eagle logo are trademarks of National Instruments Corporation. Refer to the *Trademark Information* at `ni.com/trademarks` for other National Instruments trademarks.
The mark LabWindows is used under a license from Microsoft Corporation. Windows is a registered trademark of Microsoft Corporation in the United States and other countries. Other product and company names mentioned herein are trademarks or trade names of their respective companies.

Members of the National Instruments Alliance Partner Program are business entities independent from National Instruments and have no agency, partnership, or joint-venture relationship with National Instruments.

Patents

For patents covering National Instruments products/technology, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your media, or the *National Instruments Patent Notice* at `ni.com/patents`.

Worldwide Technical Support and Product Information
ni.com

National Instruments Corporate Headquarters
11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices
Australia 1800 300 800, Austria 43 662 457990-0, Belgium 32 (0) 2 757 0020, Brazil 55 11 3262 3599, Canada 800 433 3488, China 86 21 5050 9800, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 358 (0) 9 725 72511, France 01 57 66 24 24, Germany 49 89 7413130, India 91 80 41190000, Israel 972 3 6393737, Italy 39 02 41309277, Japan 0120-527196, Korea 82 02 3451 3400, Lebanon 961 (0) 1 33 28 28, Malaysia 1800 887710, Mexico 01 800 010 0793, Netherlands 31 (0) 348 433 466, New Zealand 0800 553 322, Norway 47 (0) 66 90 76 60, Poland 48 22 328 90 10, Portugal 351 210 311 210, Russia 7 495 783 6851, Singapore 1800 226 5886, Slovenia 386 3 425 42 00, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 (0) 8 587 895 00, Switzerland 41 56 2005151, Taiwan 886 02 2377 2222, Thailand 662 278 6777, Turkey 90 212 279 3031, United Kingdom 44 (0) 1635 523545

For further support information, refer to the *Additional Information and Resources* appendix. To comment on National Instruments documentation, refer to the National Instruments Web site at ni.com/info and enter the Info Code feedback.

Contents

Student Guide
A. NI Certification vii
B. Course Description viii
C. What You Need to Get Started viii
D. Installing the Course Software ix
E. Course Goals ix
F. Course Conventions x

Lesson 1
Introduction to Real-Time Systems
A. What is a Real-Time System? 1-2
B. Real-Time System Components 1-6

Lesson 2
Configuring Your Hardware
A. Hardware Setup and Installation 2-2
B. Configuring Network Settings 2-2
C. Installing Software on Target 2-6
D. Configuring Target I/O 2-6
E. Connecting to Target in LabVIEW 2-7

Lesson 3
Real-Time Architecture: Design
A. Host and Target Application Architecture 3-2
B. Multithreading 3-3
C. Yielding Execution in Deterministic Loops 3-16
D. Improving Speed and Determinism 3-20
E. Sharing Data Locally on RT Target 3-26

Lesson 4
Timing Applications and Acquiring Data
A. Timing Control Loops 4-2
B. Software Timing 4-2
C. Hardware Timing 4-6
D. Event Response – Monitoring for Events 4-8

Lesson 5
Communication
A. Front Panel Communication 5-2
B. Network Communication 5-2
C. Network Communication Programming 5-3

Lesson 6
Verifying Your Application

A. Verifying Correct Application Behavior 6-2
B. Verifying Performance and Memory Usage 6-3

Lesson 7
Deploying Your Application

A. Introduction to Deployment 7-2
B. Creating a Build Specification 7-4
C. Communicating with Deployed Applications 7-6
D. System Replication 7-7

Appendix A
Additional Information about LabVIEW Real-Time

Appendix B
Instructor's Notes

Appendix C
Additional Information and Resources

Student Guide

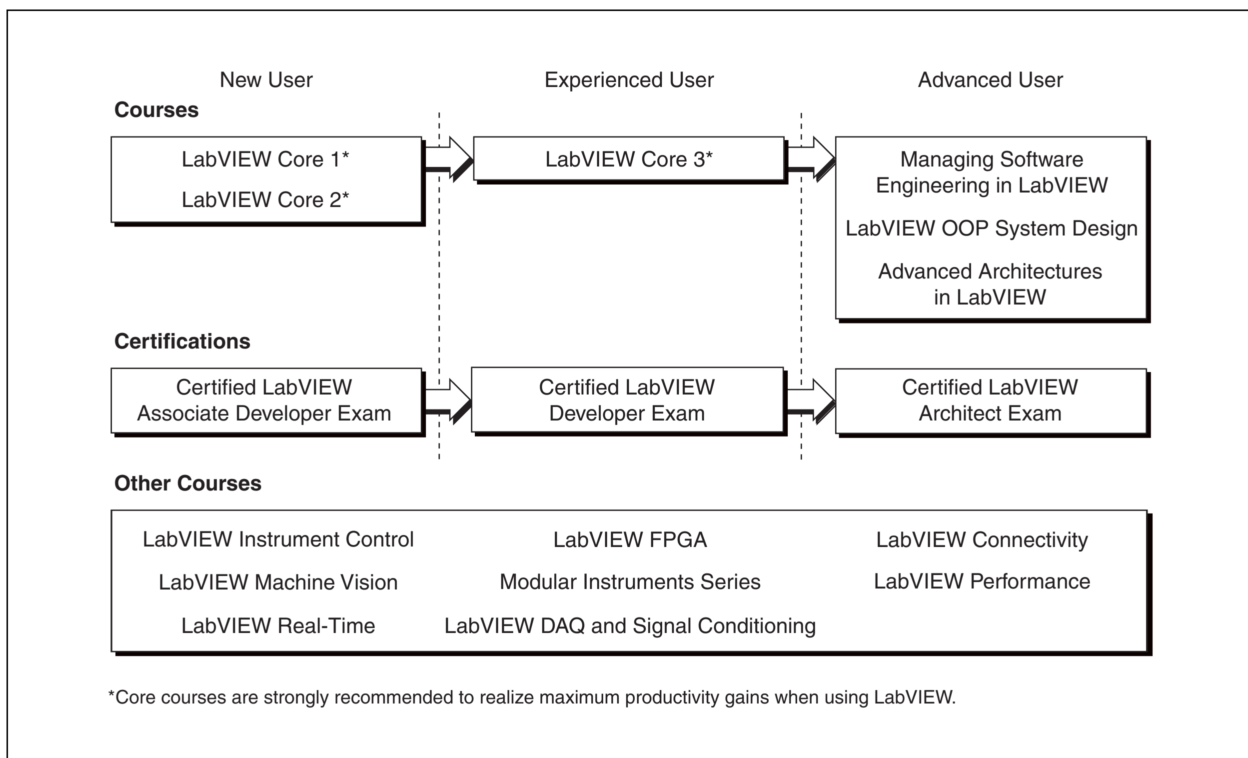
Thank you for purchasing the *LabVIEW Real-Time 1* course kit. This course manual and the accompanying software are used in the 2-day, hands-on *LabVIEW Real-Time 1* course.

You can apply the full purchase price of this course kit toward the corresponding course registration fee if you register within 90 days of purchasing the kit. Visit `ni.com/training` to register for a course and to access course schedules, syllabi, and training center location information.

Note For course manual updates and corrections, refer to `ni.com/info` and enter the Info Code `lvrt1`.

A. NI Certification

The *LabVIEW Real-Time 1* course is part of a series of courses designed to build your proficiency with LabVIEW and help you prepare for exams to become an NI Certified LabVIEW Developer and NI Certified LabVIEW Architect. The following illustration shows the courses that are part of the LabVIEW training series. Refer to `ni.com/training` for more information about NI Certification.



B. Course Description

The *LabVIEW Real-Time 1* course teaches you to use LabVIEW Real-Time to develop a deterministic and reliable application. Most LabVIEW applications run on a general-purpose operating system (OS) like Windows, Linux, Solaris, or Mac OS. Some applications require deterministic real-time performance that general-purpose operating systems cannot guarantee. The LabVIEW Real-Time Module extends the capabilities of LabVIEW to address the need for deterministic real-time performance.

This course assumes you have a level of experience with LabVIEW equivalent to completing the material in the *LabVIEW Core 1* course. In addition, you should be familiar with the Windows operating system and computer components such as the mouse, keyboard, connection ports and plug-in slots, and have experience writing algorithms in the form of flowcharts or block diagrams. The course and exercise manuals are divided into lessons, described as follows.

In the course manual, each lesson consists of the following:

- An introduction that describes the purpose of the lesson and what you will learn
- A description of the topics in the lesson
- A summary quiz that tests and reinforces important concepts and skills taught in the lesson

In the exercise manual, each lesson consists of the following:

- A set of exercises to reinforce topics
- (Optional) Self-study and challenge exercise sections or additional exercises

C. What You Need to Get Started

Before you use this course manual, make sure you have the following items:

❑ Computer running Windows 7/Vista/XP/2000

❑ LabVIEW Full Development System version 2010 or later

❑ LabVIEW Real-Time Module version 2010 or later

❑ Temperature Chamber including a 12 Volt fan, lamp, and a J-type thermocouple

❑ cRIO-9074 integrated chassis and controller with a cRIO-9211 thermocouple module and a cRIO-9474 digital output module

- ❑ *LabVIEW Real-Time 1 Exercises*
- ❑ *LabVIEW Real-Time 1* CD, which contains the following files:

Filename	Description
`Exercises`	A folder containing all files needed to complete the exercises
`Solutions`	A folder containing the solutions to each exercise
`LVRT1_2010_CourseManual_Eng.pdf`	*LabVIEW Real-Time 1 Course Manual*

D. Installing the Course Software

Insert the course CD and follow the onscreen instructions to install the software.

Exercise files are located in the `<Exercises>\LabVIEW Real-Time 1\` folder, where `<Exercises>` represents the path to the `Exercises` folder on the root directory of your computer.

E. Course Goals

This course presents the following topics:

- Concepts of real-time and determinism
- Configuring and communicating with real-time hardware
- Understanding memory usage, multithreading, priorities, and shared resource in the LabVIEW Real-Time Module
- Communicating between a host computer and RT target over the network
- Developing a deterministic, reliable application

This course does not present any of the following topics:

- Information and concepts covered in *LabVIEW Core 1* course
- Control, PID, and/or Fuzzy Logic theory
- Analog-to-digital (A/D) theory
- Operation of GPIB, RS-232, Motion, CAN, or VISA

- Every built-in LabVIEW object, function, or library VI; refer to the *LabVIEW Help* for more information about LabVIEW features not described in this course
- Development of a complete application for any student in the class; refer to the NI Example Finder, available by selecting **Help»Find Examples** for example VIs you can use and incorporate into VIs you create

F. Course Conventions

The following conventions are used in this course manual:

» — The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **Options»Settings»General** directs you to pull down the **Options** menu, select the **Settings** item, and select **General** from the last dialog box.

This icon denotes a note, which alerts you to important information.

bold — Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names, controls and buttons on the front panel, dialog boxes, sections of dialog boxes, menu names, and palette names.

italic — Italic text denotes variables, emphasis, a cross-reference, or an introduction to a key concept. Italic text also denotes text that is a placeholder for a word or value that you must supply.

`monospace` — Text in this font denotes text or characters that you enter from the keyboard, sections of code, programming examples, and syntax examples. This font also is used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames, and extensions.

Platform — Text in this font denotes a specific platform and indicates that the text following it applies only to that platform.

1

Introduction to Real-Time Systems

This lesson introduces real-time concepts such as real time, determinism, and jitter. This lesson also discusses the components of a real-time system, including the host and the target.

Topics

A. What is a Real-Time System?

B. Real-Time System Components

A. What is a Real-Time System?

The LabVIEW Real-Time Module combines LabVIEW graphical programming with the power of a real-time operating system, enabling you to build deterministic real-time applications.

A misconception about real-time is that it means *quick*. More accurately, real-time means in-time. In other words, a real-time system ensures that responses occur in time, or on time. With general purpose operating systems, you cannot ensure that a response occurs within any given time period, and calculations might finish much later or earlier than expected.

For a system to be considered real-time, all parts of it must be real-time. For example, an application that runs in a real-time operating system may not behave with real-time characteristics. The application may rely on something that does not behave in real-time, which causes the application to not behave in real-time.

Terms frequently used in the discussion of real-time systems are deterministic, loop cycle time, jitter, and embedded. Learning more about these terms helps you understand a real-time system.

Real-Time Terms

The following terms apply to real-time applications.

- **Loop Cycle Time**—The time required to execute one cycle of a loop.

  Many applications that require a real-time operating system, such as a control application, are cyclical. The time between the start of each cycle, T, is the loop cycle time, or sample period. $1/T$ is the loop rate or sample rate.
- **Jitter**—The variation of loop cycle time from the desired loop cycle time.

  Even with real-time operating systems, the loop cycle time can vary between cycles. The maximum amount that a loop cycle time varies from the desired loop cycle time is the maximum jitter.
- **Determinism**—Determinism indicates how reliably a system can respond to external events or perform operations within a given time limit. It reflects the magnitude of the jitter.

  High determinism, a characteristic of real-time systems, guarantees that your calculations and operations occur within a given time. Deterministic systems are predictable. This is important in a control application that measures inputs, makes calculations based on the inputs, then returns values that are a result of those calculations. Real-time systems can guarantee that the calculations finish on time, all of the time.

- **Latency**—Time required to respond to an event, or the time between input and output.

  Deterministic systems may still have a high latency. Properly implemented real-time systems have real-time event response, which guarantees a worst case latency.

- **Embedded**—A computer system that is a component within a larger system. Embedded systems operate headlessly.

  A headless system has no user interface, such as a keyboard, monitor, or mouse. In many cases, embedded applications operate within restrictions on the amount of RAM and other resources that you can use, as well as the amount of physical space the embedded application can occupy. Embedded hardware ranges from industrial computers such as PXI/CompactPCI systems that sit within larger machines monitoring and control systems to thin-client Web servers running on a chip.

- **Time Critical Code**—Code that needs to execute on a specific schedule to function as desired.

  Time critical code is code that cannot handle delays in execution. For example, hardware I/O that has specified timing needs to execute exactly when expected. An example of a non-time-critical code is logging data to a file. Time-critical code usually has very high priority.

- **Priority**—A characteristic that defines when a VI or loop should execute relative to other VIs and loops.

  Correctly configured timed structures in RT programs should have a priority associated with them. This priority provides the RTOS with an order of importance when deciding what needs to be executed. The scale for priority ranges from 1 to 65,535 with the larger number indicating greater priority.

Maximum Jitter

All systems have some jitter, but the jitter is lower for real-time systems than for general purpose operating systems. The jitter associated with real-time systems can vary widely. General purpose operating systems have high or unbounded maximum jitter that is inconsistent. Refer to Lesson 3, *Real-Time Architecture: Design*, for more information about programming techniques that reduce jitter.

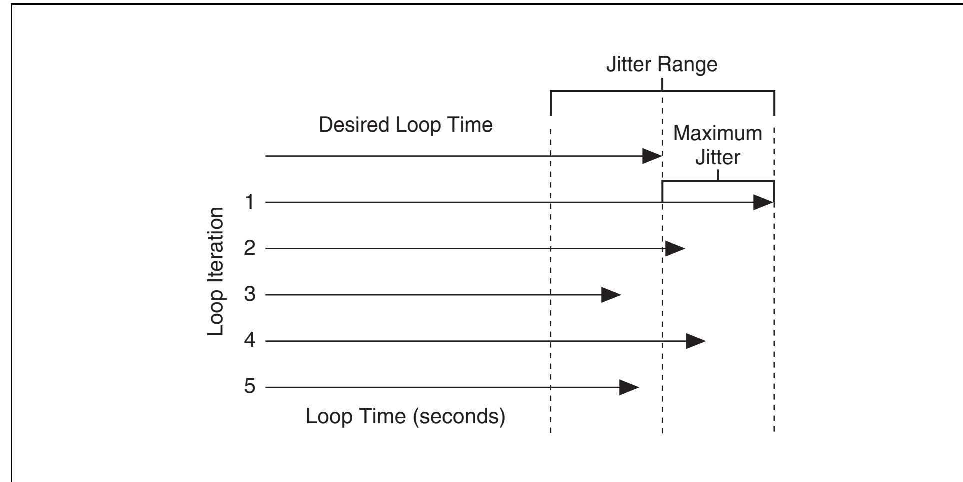


Figure 1-1. Maximum Jitter

Operating Systems

LabVIEW applications running on Windows are not guaranteed to run in real time because Windows is not a real-time operating system. Windows cannot ensure that code always finishes within specific time limits. The time your code takes to execute in Windows depends on many factors, including other programs running in the background, such as screen saver or virus software. Windows also must service interrupts from devices such as a USB port, keyboard, mouse, and other peripherals that can delay execution of code.

You can increase the probability of programs running 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. Finally, for better determinism, you can write a device driver in Windows to get the most direct access to hardware possible. Nevertheless, increasing determinism does not ensure that code always executes with real-time behavior because Windows can preempt your LabVIEW applications, even if you use time-critical priority. Refer to Lesson 3, *Real-Time Architecture: Design*, for more information about priorities.

With the LabVIEW Real-Time Module, your applications run in a separate real-time operating system (RTOS). You need not disable programs or write device drivers to achieve real-time performance. A real-time operating system enables users to prioritize tasks so that the most critical task always takes control of the processor when needed.

Real-Time Operating Systems

National Instruments designed the LabVIEW Real-Time Module to execute VIs on two different real-time operating systems. The LabVIEW Real-Time Module can execute VIs on hardware targets running the RTOS of the NI Embedded Tool Suite (ETS) or Wind River VxWorks.

NI ETS and Wind River VxWorks provide an RTOS that runs on NI RT Series hardware to enable deterministic behavior and extended reliability.

The Real-Time Module platforms do not support some LabVIEW features for VIs that run on ETS and VxWorks targets. Refer to the Unsupported LabVIEW Features (ETS) and Unsupported LabVIEW Features (VxWorks) LabVIEW help topics for information about unsupported LabVIEW features on each Real-Time Module OS.

Selecting an Operating System

If you only want to acquire real-time data, you might not need an RTOS. National Instruments has many data acquisition (DAQ) devices that can acquire data in real time even though they are controlled by programs running in Windows. The DAQ device has an onboard hardware clock that ensures a constant rate of data acquisition. With technologies such as bus mastering, direct memory access (DMA) transfer, and data buffering, the I/O device can collect and transfer data automatically to RAM without involving the CPU.

However, consider an application where every data point must be acquired and analyzed by software before you can determine if an event has occurred that requires a response. Similarly, consider an application where every acquired point must be handled by software in order to determine the output of a control loop. In both these cases, the software and the operating system must behave deterministically. You must predict their timing characteristics—and those characteristics must be the same for any data set, at any time. In these applications, the software must be involved in the loop; therefore, you require an RTOS to guarantee response within a fixed amount of time.

In addition, applications requiring extended run times or headless operation are often implemented with an RTOS.

Real-Time Development Tools

Real-time development tools include code development tools such as the compiler, the linker, and the debugger. In addition, system analysis tools provide advanced insight into optimizing real-time applications.

The LabVIEW Real-Time Module application development environment serves as a complete development and debugging tool. For more advanced diagnostics, use the LabVIEW Execution Trace Toolkit for complete real-time application analysis.

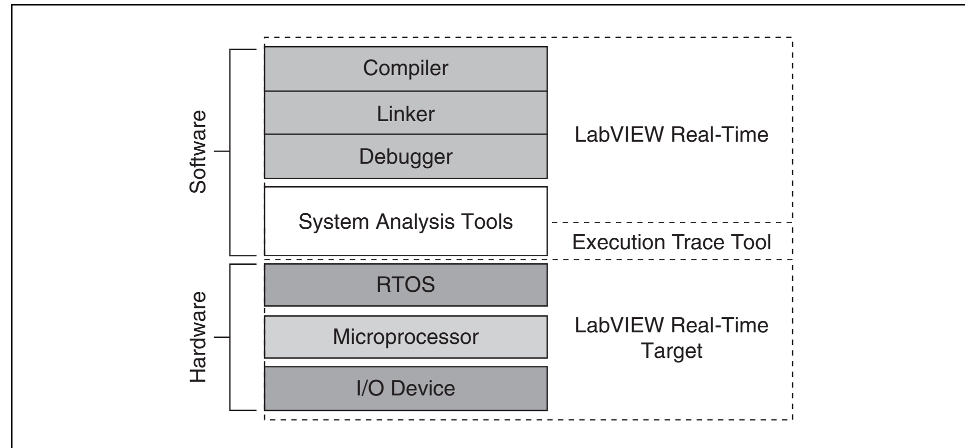


Figure 1-2. Real-Time Development Tools

The LabVIEW Real-Time Module deployment platforms are based on a common hardware and software architecture. Each hardware target uses computing components such as a microprocessor, RAM, non-volatile memory, and an I/O bus interface. The embedded software consists of an RTOS, driver software, and a specialized version of the LabVIEW Run-Time Engine.

B. Real-Time System Components

A real-time system consists of software and hardware components. The software components include LabVIEW, the RT Engine, and the LabVIEW projects and VIs you create. The hardware components of a real-time system include a host computer and an RT target. The following sections describe the different components of a real-time system.

Host Computer

The host computer is the computer on which LabVIEW and the LabVIEW Real-Time Module are installed and on which you develop the VIs for the real-time system. After developing the real-time system VIs, you can download and run the VIs on RT targets. The host computer can run VIs that communicate with VIs running on RT targets to provide a user interface.

LabVIEW

You develop VIs with LabVIEW on the host computer. The Real-Time Module extends the capabilities of LabVIEW with additional tools for creating, debugging, and deploying deterministic VIs.

RT Engine

The RT Engine is a version of LabVIEW that runs on RT targets. The RT Engine runs the VIs you download to RT targets. The RT Engine provides deterministic real-time performance for the following reasons:

- The RT Engine runs on a real-time operating system (RTOS), which ensures that the LabVIEW execution system and other services adhere to real-time operation.
- The RT Engine runs on RT Series hardware. RT targets are designed to run only the VIs and device drivers necessary for RT applications, which prevents other applications from impeding the execution of RT VIs.
- RT targets do not use virtual memory, because virtual memory can cause unpredictable performance.

RT Target

An RT target refers to RT Series hardware that runs the RT Engine and VIs you create using LabVIEW. A networked RT Series device is a networked hardware platform with an embedded processor and a real-time operating system that runs the RT Engine and LabVIEW VIs. You can use a separate host computer to communicate with and control VIs on a networked RT Series device through an Ethernet connection. Some examples of networked RT Series devices include the following:

- NI CompactRIO Series—A reconfigurable control and acquisition system designed for applications that require high performance and reliability.
- NI RT Series PXI Controller—A networked device installed in an NI PXI chassis that communicates with NI PXI modules installed in the chassis. You can write VIs that use all the input/output (I/O) capabilities of the PXI modules, SCXI modules, and other signal conditioning devices installed in a PXI chassis. The RT Engine also supports features of the RT Series PXI controller. Refer to the LabVIEW Real-Time Support page on the National Instruments Web site for information about the features supported by the RT Engine on specific networked devices.
- NI RT Series [c]FP-2xxx—A networked device that runs the ETS RTOS.
- NI 1450 Series Compact Vision System—An easy-to-use, distributed, real-time imaging system that acquires, processes, and displays images from IEEE 1394 cameras.
- Desktop PCs as RT Targets—A desktop PC configured with RT Engine software.

Note The *LabVIEW Help* does not contain hardware-related information about specific networked devices. Refer to the appropriate device documentation for information about the device.

USB Storage Devices

The Real-Time Module includes support for USB storage devices, such as thumb drives and external USB hard drives, for RT targets that have onboard USB hardware. Connect an external USB storage device to a USB port of an RT target and then access the device from VIs running on the RT target.

When you plug a USB thumb drive into the RT system, the thumb drive is automatically assigned a drive letter of `U:`. Each additional drive you add is automatically assigned the next available drive letter. For example, `V:`, `W:`, `X:`, and so on.

Summary – Quiz

Match the following terms with their definitions:

Jitter	A. How reliably a system responds to events or performs operations within a given time limit
Determinism	B. Time taken to execute one cycle of a loop
Real-time	C. Variation of loop cycle time from the desired loop cycle time
Loop cycle time	D. The ability to reliably, and without fail, respond to an event or perform an operation within a guaranteed time period

Summary – Quiz Answers

Match the following terms with their definitions:

Jitter	**C. Variation of loop cycle time from the desired loop cycle time**
Determinism	**A. How reliably a system responds to events or performs operations within a given time limit**
Real-time	**D. The ability to reliably, and without fail, respond to an event or perform an operation within a guaranteed time period**
Loop cycle time	**B. Time taken to execute one cycle of a loop**

Notes

2

Configuring Your Hardware

In this lesson, you learn how to configure your target hardware and the I/O hardware used in the class project. You can apply this knowledge to configuring other types of I/O hardware as well.

In addition, you learn how to download a VI to the target, connect to it, and disconnect from it. This gives you the knowledge to run pre-existing code on an RT target in development mode.

Topics

A. Hardware Setup and Installation

B. Configuring Network Settings

C. Installing Software on Target

D. Configuring Target I/O

E. Connecting to Target in LabVIEW

A. Hardware Setup and Installation

To configure your real-time system, complete the following steps:

1. Set up real-time hardware and host computer.
2. Configure your target.
3. Configure your target I/O.
4. Connect to your target in LabVIEW.

Each target hardware type has its own hardware setup and installation instructions. Refer to the appropriate device documentation for information about the device.

You must install the LabVIEW Real-Time Module on the host computer before you can begin developing real-time applications. Use the host computer to develop applications and then download them across the network to the RT Series target system.

B. Configuring Network Settings

The host computer communicates with the remote system over a standard Ethernet connection. If the host computer is already configured on a network, you must configure your remote system on the same network. If neither machine is connected to a network, you must connect the two machines directly using a CAT-5 crossover cable or hub. You can use the direct connection to configure the remote system from the host computer system.

Using MAX to Detect Remote Systems

Expand **Remote Systems** in the Measurement & Automation (MAX) configuration tree. Previously detected remote systems appear immediately beneath **Remote Systems** in the configuration tree. MAX continues to search for newly attached remote systems on the local subnet. Detected systems are added to the list after a short delay. All detected systems appear beneath **Remote Systems** in the configuration tree. MAX searches for new remote systems every time you launch MAX and expand **Remote Systems**. You can also detect newly connected remote systems by selecting **View» Refresh** or by pressing <F5> to scan for local and remote devices.

Note The IP address of your remote system appears as the default remote system name in the configuration tree. If more than one system appears in **Remote Systems**, select the IP address of the system you want to configure. Use the Network Settings tab of the configuration view to assign a host name, if available, to your remote system. MAX then uses the name to identify the device in the configuration tree. This host name is not necessarily a DNS host name. For more information about host names, refer to

Configuring Network Settings in the *Measurement and Automation Explorer Help*. The system state may be Unconfigured if your target does not support automatic IP assignment and if you have not set the IP address for the target.

Assigning an IP Address

You can connect to your remote system either by connecting it and your host computer to a local area network or by connecting it directly to your host computer using a CAT-5 crossover cable. Either way, your remote system must have an IP address assigned to it. You can either attempt to automatically obtain an IP address or manually specify one.

Some targets, such as all FieldPoint FP-160x RT targets, require that you specify a static IP address, as they do not support automatic IP address assignment. You may also need to specify an IP address if your remote system is not connected to a network and you want to make a direct connection. Refer to the *Specifying a Static IP Address* section for more information about specifying an IP address.

Specifying a Static IP Address

If you choose to specify an IP address, select **Static** from the Network Settings tab, fill in the network parameters described below with correct values for your network, then click **Save**. You must restart the remote system for any changes to take effect.

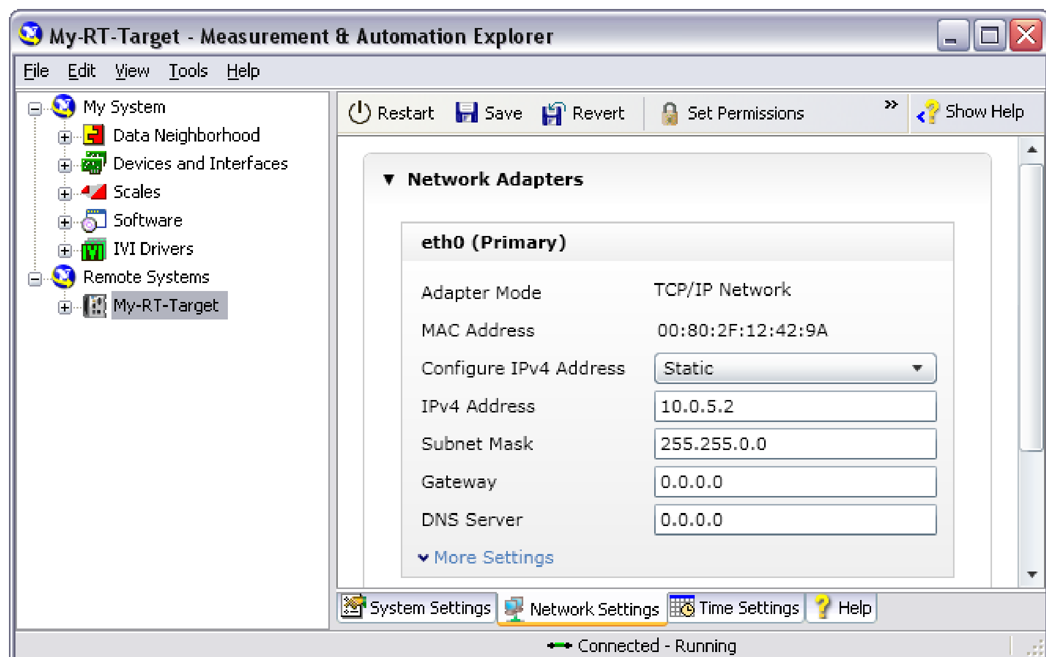


Figure 2-1. Network Settings for Obtaining a Static IP Address

IP Address—The unique address of a device on your network. Each IP address is a set of four one- to three-digit numbers. Each number is in the range from 0 through 255 and is separated by a period. This format is called dotted decimal notation. The IP address `224.102.13.24` is an example of dotted decimal notation.

Subnet Mask—A code that helps the network device determine whether another device is on the same network or a different network. `255.255.255.0` is the most common subnet mask.

Gateway—The IP address of a device that acts as a gateway server, which is a connection between two networks.

DNS Address—The IP address of a network device that stores DNS host names and translates them into IP addresses.

Consult with your network administrator before specifying these parameters. If you do not have a network administrator or you are the network administrator, refer to the *IP Settings Information* topic in the *MAX Remote Systems Help* for more information.

If you are assembling your own Ethernet network, you can choose an IP address. The subnet mask determines the format of the IP address. Use the same subnet mask as the host computer when you configure your remote system. For example, if your subnet mask is `255.255.255.0`, the first three numbers in every IP address on the network must be the same. If your subnet mask is `255.255.0.0`, only the first two numbers in the IP addresses on the network must match.

For either subnet mask, you can use numbers between 1 and 254 for the last number of the IP address. (Do not use numbers 0 and 255; they are reserved.) You can use numbers between 0 and 255 for the third number of the IP address, but this number must be the same as other devices on your network if your subnet mask is `255.255.255.0`.

If you are setting up your own network and do not have a gateway or DNS server, set these values to the default configuration, `0.0.0.0`.

To find out the network settings for your host computer, run ipconfig.

To run ipconfig, open a command prompt window, type ipconfig at the prompt, and press <Enter>. If you need more information, run ipconfig with the `/all` option by typing `ipconfig/all` to see all the settings for the computer. Make sure you use the settings for the correct Ethernet adapter to configure your remote system.

Obtaining an IP Address Automatically from a DHCP Server

If your remote system is on a network that has a DHCP server, you may be able to automatically obtain an IP address from the DHCP server. A DHCP server allocates an IP address to your target each time the target is started. You do not need to specify other information such as **Subnet Mask** if you select the **DHCP or Link Local** option. If you do not know whether your network has a DHCP server, check with your network administrator for assistance. To automatically obtain an IP address, select **DHCP or Link Local**, then click **Save**. You must restart the remote system for any changes to take effect.

Not all DHCP servers are implemented in the same manner. Therefore, some might not be compatible with the LabVIEW Real-Time Module. After you select **DHCP or Link Local** and restart the RT target, LabVIEW Real-Time tries to obtain an IP address from the DHCP server. If this operation fails, LabVIEW Real-Time automatically restarts the RT target and attempts to assign a link local IP address (`169.254.x.x`), if your target supports this feature. Link local addresses are network addresses intended for use in a local network only. After three failed attempts, LabVIEW Real-Time returns to the default configuration with IP address `0.0.0.0`. In this case, you need to explicitly specify the network parameters.

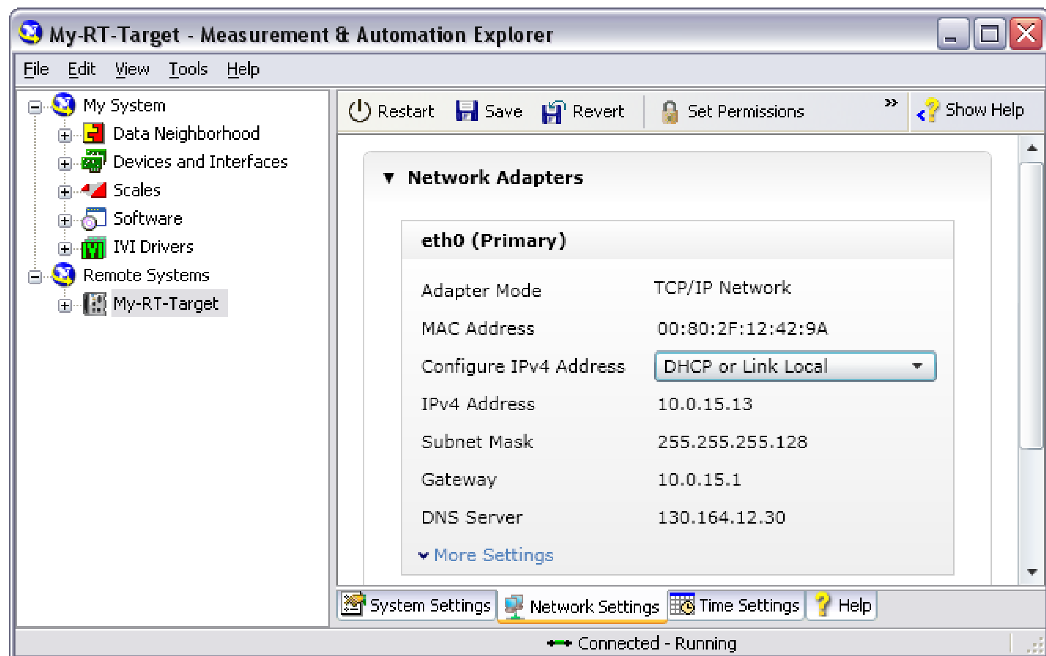


Figure 2-2. Network Settings for Automatically Obtaining an IP Address

In addition, when you use a DHCP server, the server allocates an IP address to the remote system each time you boot the target. The new IP address might be different than the address previously assigned. If you use the DHCP server to assign an IP address to your target, you need to check the

IP address using MAX each time you target LabVIEW Real-Time to the target. To avoid needing to check the IP address each time, specify a static IP address for the target instead of using a DHCP server. Typical DHCP servers allow you to reserve specific IP addresses for static IP addresses.

C. Installing Software on Target

After you have assigned an IP address, you can update or install the LabVIEW Real-Time Module or other driver software on the remote target. If your RT target has the LabVIEW Real-Time Module preinstalled, you may still need to download additional driver software or update existing driver software. The LabVIEW Real-Time Software Wizard facilitates checking and downloading software. To launch it, click the **Add/Remove Software** icon on the toolbar to open the LabVIEW Real-Time Software Wizard window.

Complete the following steps to launch the LabVIEW Real-Time Software Wizard:

1. Expand Remote Systems in the configuration tree and then expand your RT target.
2. Select the Software category. Click the **Add/Remove Software** icon on the toolbar to launch the LabVIEW Real-Time Software Wizard. If your RT target does not have a Software category, it does not support the LabVIEW Real-Time software.
3. Use the LabVIEW Real-Time Software Wizard to add, remove, or update the software on your remote target.

D. Configuring Target I/O

You must configure any National Instruments RT-compatible device before you can access it from a LabVIEW Real-Time Module application targeted to the remote system. If you are using a PXI, Fieldpoint, or Compact Vision System as your RT target, you should configure the I/O of your RT target before you access it in LabVIEW. If you are using CompactRIO as your RT target, you will configure the I/O using a LabVIEW project.

For more information about using any NI products in MAX, refer to the product-specific documentation.

E. Connecting to Target in LabVIEW

To deploy VIs using the LabVIEW Real-Time Module, you must create a project, create and configure a real-time target in the project, and connect to the target.

LabVIEW Projects

Use the LabVIEW project to manage files and targets as you develop a system. You control projects through the Project Explorer window. The Project Explorer window includes two pages, the Items page and the Files page. The Items page displays the project items as they exist in the project tree. The Files page displays the project items that have a corresponding file on disk. You can organize filenames and folders on this page. Project operations on the Files page both reflect and update the contents on disk.

A project can contain LabVIEW files, such as VIs, custom controls, type definitions, and templates, as well as supporting files, such as documentation, data files, or configuration files.

You must use projects to build applications and shared libraries. You also must use a project to work with an RT, FPGA, mobile device, Touch Panel, DSP, or embedded target. Refer to the specific module documentation for more information about using projects with these targets.

Each project can have multiple targets, representing the host computer as well as real-time systems, FPGA systems, and mobile devices. When you place a VI in a target in the Project Explorer, the VI becomes targeted to that system and has palettes appropriate to the target.

Adding Folders to a Project

Use the Project Explorer window to add folders to create an organizational structure for items in a LabVIEW project.

Adding auto-populated folders adds a directory on disk to the project. LabVIEW continuously monitors and updates the folder according to changes made in the project and on disk. A blue folder icon with a yellow cylinder identifies this type of folder. To disconnect an auto-populated folder from disk, right-click the auto-populated folder on the Items page and select Stop Auto-populating from the shortcut menu. LabVIEW disconnects the folder from the corresponding folder on disk. This option is available only to top-level folders and applies recursively to subfolders of auto-populated folders.

A virtual folder is a folder in the project that organizes project items and does not represent files on disk. A silver folder icon identifies this type of folder. You can convert a virtual folder to an auto-populated folder.

Right-click the virtual folder and select **Convert to Auto-populating Folder** to display a file dialog box. Select a folder on disk to auto-populate with. An auto-populated folder appears in the project. LabVIEW automatically renames the virtual folder to match the disk folder and adds all contents of the disk folder to the project. If items in the directory already exist in the project, the items move within the auto-populated folder. Items in the virtual folder that do not exist in the directory on disk move to the target.

Project Libraries

LabVIEW project libraries are collections of VIs, type definitions, shared variables, palette menu files, and other files, including other project libraries. When you create and save a new project library, LabVIEW creates a project library file (`.lvlib`), which includes the properties of the project library and the references to files that the project library owns.

Use libraries to group and control a set of VIs, controls, and variables. A library does not affect the location of files on a disk. However, files in a library are explicitly linked to that library. Adding a file in a library to a project adds the entire library to the project. LabVIEW reports an error if a file cannot locate the library it is a part of, or if a library cannot locate files that are part of it.

Libraries define a namespace, which prevents name conflicts between files inside a library and files outside a library. This allows you to have multiple VIs with the same name in memory at the same time, as long as each VI resides in a separate library.

You can define each item in a library as public or private. VIs outside the library can use public items, but VIs can use private items only within the same library. By defining public and private items for a library you provide a controlled interface to anyone using the library and prevent users of the library from directly accessing low-level, private items.

Refer to the *Using Project Libraries* topic of the *LabVIEW Help* for more information about project libraries.

Libraries are required to use shared variables. Refer to Lesson 3, *Real-Time Architecture: Design*, and Lesson 5, *Communication*, for more information about shared variables.

Creating a Project

Complete the following steps to create a project.

1. Select **File»New Project** to display the Project Explorer window. By default the new project includes the My Computer target that represents the host computer.
2. Add items you want to run on the host computer to the My Computer target.
3. Select **File»Save** to save the project.

Adding a Real-Time Target

To add a target to a LabVIEW project, you must have a module or driver that supports targets installed. Complete the following steps to add a target or device to an existing project.

1. Right-click the project root and select **New»Targets and Devices** to display the Add Targets and Devices dialog box. If a target in the project supports other targets, you also can right-click the target and select **New»Targets and Devices** to add a target under the existing target. Examples of external targets include RT cRIO, PXI, cFP, and RT Desktop systems.
2. Select the type of RT target you want to add from the Targets and Devices section of the Add Targets and Devices dialog box. You can select from the following types of RT targets:
   - Existing target or device.
   - New target or device.
3. Select a target and click **OK**. An item representing the RT target appears in the Project Explorer window.

Note You cannot add non-real-time desktop computers to a project as targets.

Connecting to a Target

Right-click a target and select **Connect** to open a front panel connection with the target. LabVIEW verifies that the target responds and checks for VIs running on the target that do not match the current project. If you do not manually connect to the target, LabVIEW connects automatically when you run a VI on the target.

Note You can change the IP address of the target by right-clicking the target and selecting **Properties**.

Adding VIs to a Target

To run VIs on an RT target, add the VIs to the project tree under the appropriate target. You can add new or existing VIs by right-clicking the target or by selecting the **Project** menu. You also can drag items from other locations in the project tree or drag files from the Windows Explorer.

When you add a VI to the project tree under a target, the VI becomes targeted to that target. Real-time targeted VIs display specific real-time **Controls** and **Function** palettes. When you run a real-time targeted VI, LabVIEW automatically downloads and runs the VI on the target.

Running VIs on a Target

After you connect to a target, you can download a VI to the RT target. The block diagram runs on the RT target.

The communication between the compiled code and the host PC is transparent to the user and occurs through *front-panel communication*. Refer to Lesson 5, *Communication*, for more information about front-panel communication and other communication methods.

The Real-Time Development System can use all the debugging features in LabVIEW except the call chain ring. Refer to Lesson 6, *Verifying Your Application*, for more information about debugging your application.

Closing a Front Panel Connection Without Closing VIs

You can exit LabVIEW on the host computer without closing the VIs on the RT target. Select **File»Exit** to close LabVIEW on the host computer. A dialog box prompts you to exit LabVIEW without closing RT Engine VIs. Select **Close** to abort the VIs running on the target before exiting. Select **Disconnect** if you want the VIs running on the RT target to continue running.

You also can disconnect the RT target connection from the Project Explorer. When you disconnect, any running VIs continue to run on the target, but debugging and front panel communication are disabled. Reconnecting to the target automatically opens all VIs running on that target and re-establishes the connection for debugging and front panel communication.

If you connect to a target that is running VIs that are not in the active project, LabVIEW prompts you to abort the VIs or add them to the project before opening them.

Summary – Quiz

1. Which of the following are methods for connecting target and host computers?
   a. Connect target and host computers to the same local area network
   b. Connect target computer directly to host computer using an Ethernet crossover cable
   c. Both a & b

2. True or False? If your target is configured to obtain an IP address automatically from a DHCP server, the target will have the same IP address every time it boots up.

3. For LabVIEW to connect to and run VIs on the RT target, you must create a ____________.
   a. DHCP server
   b. Local Area Network
   c. LabVIEW Project

Summary – Quiz Answers

1. Which of the following are methods for connecting target and host computers?
    a. Connect target and host computers to the same local area network
    b. Connect target computer directly to host computer using an Ethernet crossover cable
    c. Both a & b

2. True or False? If your target is configured to obtain an IP address automatically from a DHCP server, the target will have the same IP address every time it boots up.

    False

3. For LabVIEW to connect to and run VIs on the RT target, you must create a ____________.
    a. DHCP server
    b. Local Area Network
    c. LabVIEW Project

Notes

3

Real-Time Architecture: Design

When implementing a system with the LabVIEW Real-Time Module, consider whether you need to use determinism. If your application only needs the embedded qualities of the LabVIEW Real-Time Module, including the reliability of the LabVIEW Real-Time Module, the ability to off-load processing, and headless black box design, it does not need to be deterministic. However, if your application must guarantee a response to an external event within a given time or meet deadlines cyclically and predictably, it must be deterministic.

When designing applications within real-time constraints, you must employ certain programming techniques to achieve determinism. When programming a LabVIEW Real-Time Module application, you can decide how to establish communication between multiple tasks or threads without disrupting determinism. This lesson discusses how to design your application to achieve determinism.

Topics

A. Host and Target Application Architecture

B. Multithreading

C. Yielding Execution in Deterministic Loops

D. Improving Speed and Determinism

E. Sharing Data Locally on RT Target

A. Host and Target Application Architecture

Figure 3-1 demonstrates the basic architecture of a well-designed real-time application. The overall task is divided into two parts—the host application and the target application. The host application contains the user interface. The target application is divided into two parts—the deterministic loop and the non-deterministic loops. These loops are contained within separate VIs.

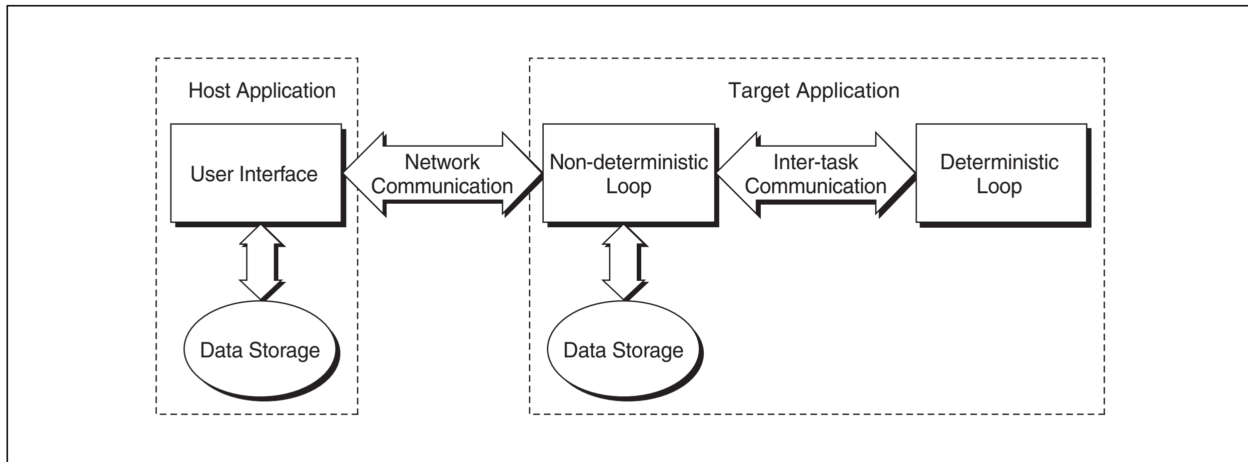


Figure 3-1. Host and Target Application Architecture

Deterministic applications depend on deterministic tasks to complete on time, every time. Therefore, deterministic tasks need dedicated processor resources to ensure timely completion. Dividing tasks helps to ensure that each task receives the processor resources it needs to execute on time.

Place any code that must execute deterministically in the deterministic loop. Place all other code in non-deterministic loops. In most applications, the deterministic loop handles all control tasks and/or safety monitoring and the non-deterministic loops handle all communication and data logging.

Host Application

The host application runs on the host computer and communicates with VIs running on the target computer. This communication may involve user interface information, data retrieval, data broadcast to other systems needing data from the target application, and any other non-deterministic tasks that you may need.

Target Application

The target application consists of deterministic code and non-deterministic code. Use a priority scheme to separate the portions of the program that must behave deterministically from the rest of the application.

Deterministic versus Non-Deterministic Processes

Deterministic applications often perform a critical task iteratively, so that all iterations consume a measurably precise amount of processor time. Thus, deterministic applications are valuable not for their speed, but for their reliability in consistently responding to inputs and supplying outputs with little jitter.

A common example of a deterministic application is a deterministic control loop, which gathers information about a physical system and responds to that information with precisely-timed output. Consider the oil industry where thousands of feet of pipes are assembled daily. As two pipes are mechanically threaded together end-to-end, the torque required to twist the pipes increases until the pipes are fully connected. Suppose the machine connecting the pipes uses a control loop to respond to an increase in resistance between the pipes by applying more torque. After a critical level of torque is attained, the control loop is triggered to terminate. Under these conditions, the loop must execute deterministically because lag in the software could result in severe damage to the pipes and other equipment.

Understanding multithreading is a prerequisite to understanding priority levels. Multithreading expands the idea of multitasking.

B. Multithreading

Multitasking refers to the ability of the operating system to quickly switch between tasks, giving the appearance of simultaneous execution of those tasks. For example, in Windows 3.1, a task is generally an entire application, such as Microsoft Word, Microsoft Excel, or LabVIEW. Each application runs for a small time slice before yielding to the next application. Windows 3.1 uses a technique known as cooperative multitasking, where the operating system relies on running applications to yield control of the processor to the operating system at regular intervals. Occasionally, applications either do not yield or yield inappropriately and cause execution problems.

Windows 2000/XP relies on preemptive multitasking, where the operating system can take control of the processor at any instant, regardless of the state of the application currently running. Preemptive multitasking guarantees better response to the user and higher data throughput. This minimizes the possibility of one application monopolizing the processor.

What is Multithreading?

Multithreading applies the concept of multitasking to a single application by breaking it into smaller tasks that execute in different execution system threads. A *thread* is a completely independent flow of execution for an application within the execution system. Multithreaded applications maximize the efficiency of processors because the processors do not sit idle if there are other threads ready to run. An application that reads and writes from a file, performs I/O, or polls the user interface for activity can benefit from multithreading because it can use processors to run other tasks during breaks in these activities.

For example, in a LabVIEW multithreaded program, the application might be divided into three threads—a user interface thread, a data acquisition thread, and an instrument control thread—each of which can be assigned a priority and operate independently. Thus, multithreaded applications can have multiple tasks progressing in parallel with other applications. Multithreading allows LabVIEW to run tasks in true parallel on multi-core symmetric multiprocessing (SMP) systems.

The operating system divides processing time on the different threads similarly to the way it divides processing time among entire applications in an exclusively multitasking system.

Advantage of Multithreading

Multithreading provides several advantages for a real-time system. First, multithreading allows you to conceptually divide your code into independent tasks, which can effectively execute at the same time. Second, multithreading allows you to take full advantage of multi-core or multiple processor systems. In order to utilize the capabilities of multi-core systems, you must have multiple tasks in your code that can execute at the same time. Finally, multithreading is useful for dividing a program into deterministic and non-deterministic tasks.

Multithreading is useful when parts of your code are inherently non-deterministic or parts of your code rely on non-deterministic I/O. A control loop and safety monitoring are considered deterministic because both must execute on time, every time to ensure accuracy. Communication is non-deterministic because a person or computer may not respond on time, every time. Likewise, data logging is non-deterministic because an accurate time stamp can identify when the data was collected or calculated.

What could happen to a deterministic process if a non-deterministic task were involved? Placing network communication tasks (non-deterministic tasks) inside the deterministic loop may harm determinism. For example, if deterministic code relies on responses from another PC over the network

and if the other PC does not reply in time, the deterministic code may miss a deadline. To prevent missed deadlines, separate the threads into deterministic tasks and non-deterministic tasks. Then you can assign a higher priority to deterministic tasks to ensure that they always finish on time.

The ability to assign leveled priorities is an important feature of real-time operating systems.

Real-Time Multithreading Analogy

To illustrate real-time multithreading, imagine a car repair garage. There is only one mechanic—he represents the processor and his work represents processing. The receptionist who lines up the service requests as they arrive is the Operating System. All service requests are scheduled in the order they arrive, except when a higher priority customer arrives. When this happens, the receptionist schedules that customer ahead of the lower priority customers.

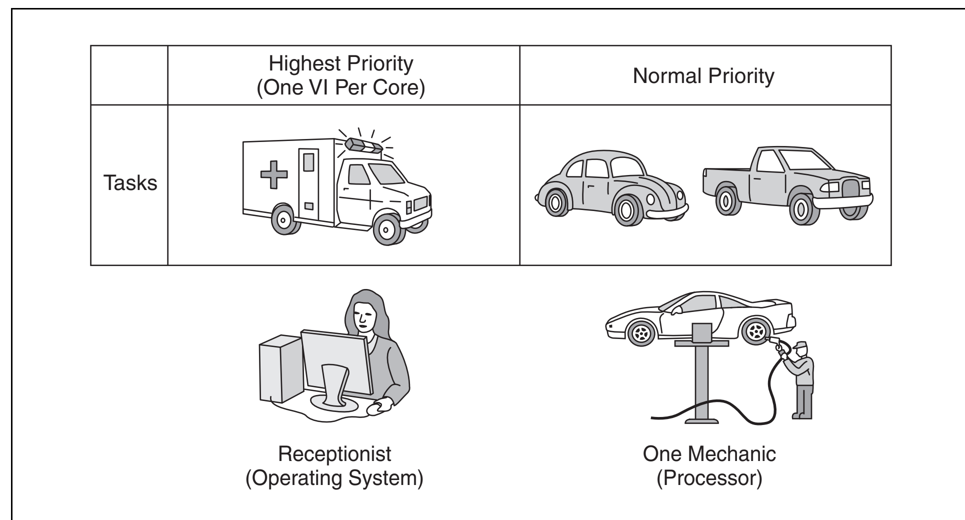


Figure 3-2. Real-Time Multithreading Analogy

In this town, there is one ambulance which is the highest priority repair. Similarly, in LabVIEW Real-Time Module applications, National Instruments recommends that you limit yourself to one deterministic loop per core. Meanwhile, the mechanic can work on the other cars of equal priority at the same time, making progress on each of them. Similarly, equal priority threads share the same CPU. However, if a higher priority car arrives while the mechanic is working on these lower priority cars, he will put them all aside to work on the higher priority car until completion. This is called *preemption*. When he is finished with the higher priority cars, he will return to the lower priority cars. If, however, there are always higher priority cars to work on, he can never return to the lower priority cars. This is called *starvation*.

Scheduling Threads

There are two methods for scheduling threads—round robin and preemptive. The RTOS on NI RT targets uses a combination of round robin and preemptive scheduling to execute threads in the execution system.

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

Preemptive scheduling means that any higher priority thread that needs to execute immediately pauses execution of all lower priority threads and begins to execute. The deterministic loop should be set to the highest priority and preempt all other priorities.

Round Robin Scheduling

Round robin scheduling shares processor time between threads based on equal shares of processor time. The time allocation for a LabVIEW Real-Time thread is 10 ms.

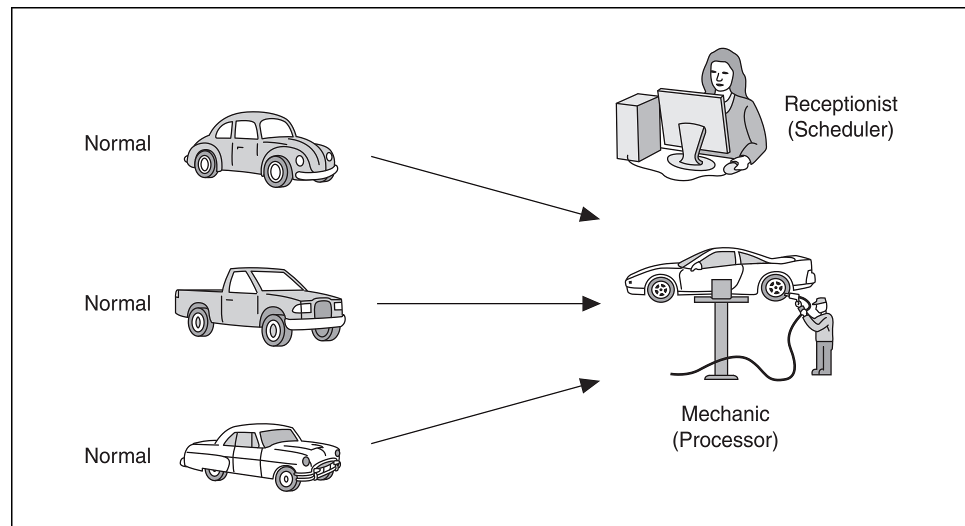


Figure 3-3. Round Robin Scheduling

To illustrate round robin scheduling, recall the analogy of an automobile repair shop. In this case, one mechanic represents the processor, and a receptionist represents the scheduler. Multiple cars represent the multiple threads of the system. Using round robin scheduling, the mechanic cycles between each car for a set period of time.

Round robin scheduling guarantees each thread has some time with the processor; however, there is no prioritization of tasks. For example, if the town has only one ambulance and it needs to be serviced, round robin scheduling would not allow the mechanic to give priority service.

Preemptive Scheduling

With preemptive scheduling, you can give priority to tasks. In this case, one thread can be designated as the most important. When the highest priority thread needs processor time, the other threads must wait until the highest priority thread is finished.

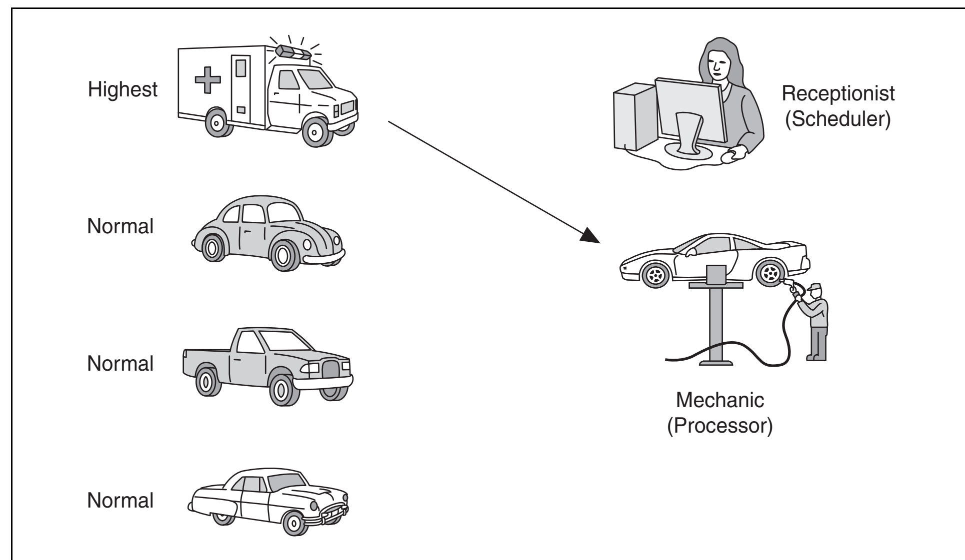


Figure 3-4. Preemptive Scheduling

In the repair shop analogy, the ambulance is assigned the highest priority. As a result, the mechanic services it as soon as it arrives. Repairs on all other cars are delayed until the ambulance service is complete. After the ambulance service is complete, the other cars resume sharing time with the mechanic.

Note A thread swap occurs when the processor switches between threads. Every thread swap takes additional time from the processor.

LabVIEW Real-Time Scheduling

Each VI in an RT application is assigned a priority. Thread priority determines the execution of VIs, with higher priority threads preempting lower priority threads. Threads with equal priority use round robin scheduling. The deterministic loop should receive the processor resources necessary to complete the task and does not relinquish control of the processor until it cooperatively yields to non-deterministic loops or until it completes the task. The non-deterministic loops then run until preempted by

the deterministic loop. The deterministic loop releases control of the processor by completing the operation or by sleeping. Without sleep time built into the deterministic loop, all other lower priority operations on the system are unable to execute.

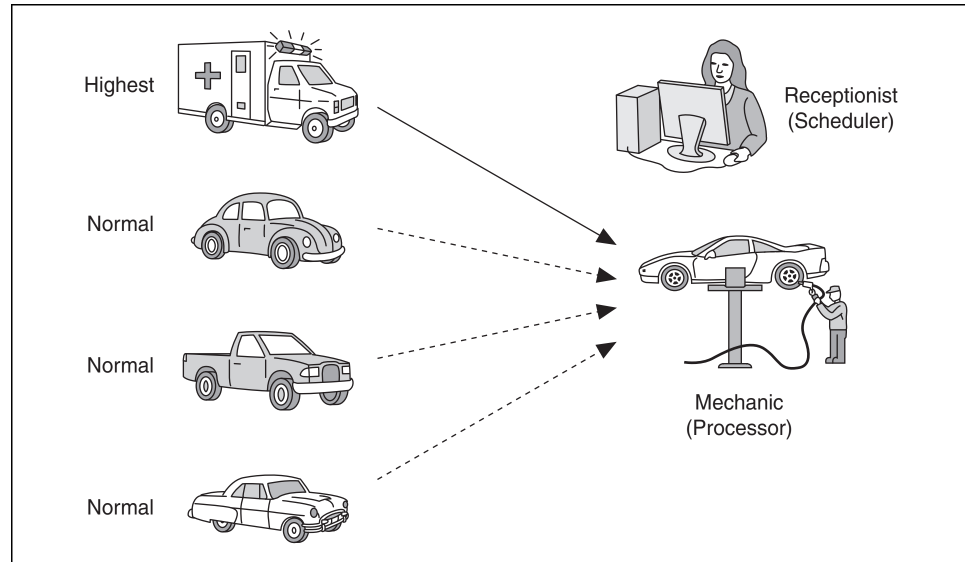


Figure 3-5. LabVIEW Real-Time Scheduling

In the repair shop analogy, the ambulance is assigned the highest priority. The mechanic services the ambulance as soon as it arrives. After the mechanic arrives at a designated sleep time or finishes service on the ambulance, the mechanic services other vehicles on a shared basis until break time is over. The mechanic then returns to working on the ambulance.

Setting Priorities

You can use Timed Loops or VIs with different priorities to control the execution and timing of deterministic tasks.

Dividing Tasks to Create Deterministic Multithreaded Applications

Deterministic applications depend on deterministic tasks to complete on time, every time. Therefore, deterministic tasks need dedicated processor resources to ensure timely completion. Dividing tasks helps to ensure that each task receives the processor resources it needs to execute on time.

Separate deterministic tasks from all other tasks to ensure deterministic tasks receive enough processor resources. For example, if a control application acquires data at regular intervals and stores the data on disk, you must handle the timing and control of the data acquisition deterministically. However, storing the data on disk is inherently a non-deterministic task because file I/O operations have unpredictable response times that depend

on the hardware and the availability of the hardware resource. You can use Timed Loops or VIs with different priorities to control the execution and timing of deterministic tasks.

Note Within deterministic tasks, ensure that each operation receives dedicated processor resources by avoiding unnecessary parallelism. In a multiple CPU system, avoid creating more parallel operations deterministic operations than the number of available CPUs. Because it is impossible to determine the execution order of parallel operations, unnecessary parallelism can impede determinism.

Creating Deterministic Applications Using VIs Set to Different Priorities

Separate deterministic tasks from non-deterministic tasks and place deterministic tasks in different VIs to ensure they receive enough processor resources. You can prioritize the VIs and then categorize them into one of the available execution systems to control the amount of processor resources each VI receives.

LabVIEW assigns each VI to an execution system thread according to the VI priority and execution system you specify. The threads execute on the processor accordingly.

Assigning Priorities to VIs

You can change the priority of a VI by right-clicking the VI in the Project Explorer window and selecting **Properties** from the shortcut menu to open the VI Properties dialog box. Select **Execution** from the Category pull-down menu in the VI Properties dialog box to open the Execution Properties page, where you can set the priority of a VI. You can select from the following VI priorities, listed in order from lowest to highest, to assign VIs a priority level:

- Background priority (lowest)
- Normal priority (default)
- Above normal priority
- High priority
- Time-critical priority (highest)

Normal priority is the default priority for all VIs you create in LabVIEW. However, subVIs inherit the priority of the caller VI. For example, a subVI called in a deterministic VI runs at time-critical priority.

Time-critical VI Priority

The time-critical priority preempts all other priorities. A time-critical priority VI does not relinquish processor resources until it completes all tasks. However, a deterministic VI can explicitly relinquish control of processor resources to ensure that the VI does not monopolize the processor resources.

Note Because time-critical priority VIs cannot preempt each other, create only one deterministic VI per CPU to guarantee deterministic behavior.

In addition to the five priority levels previously listed, you can set VIs to subroutine priority. VIs set for subroutine priority do not share execution time with other VIs. When a VI runs at the subroutine priority level, it effectively takes control of the thread in which it is running, and it runs in the same thread as its caller. No other VI can run in that thread until the subroutine VI finishes running, even if the other VI is at the subroutine priority level.

Creating Deterministic Applications Using the Timed Loop

Separate deterministic tasks from non-deterministic tasks and place them in a different Timed Loop in an RT target VI to ensure the deterministic tasks receive enough processor resources. A Timed Loop executes a subdiagram each iteration of the loop at the period and priority you specify. The higher the priority of a Timed Loop, the greater priority the structure has relative to other timed structures on the block diagram.

Timed Loops execute at a priority below the time-critical priority of any VI but above high priority, which means that Timed Loops execute in the data flow of a block diagram ahead of any VI not configured to run at a time-critical priority.

What is a Timed Loop?

The Timed Loop includes the Input, Left Data, Right Data, and Output nodes, as shown in Figure 3-6.

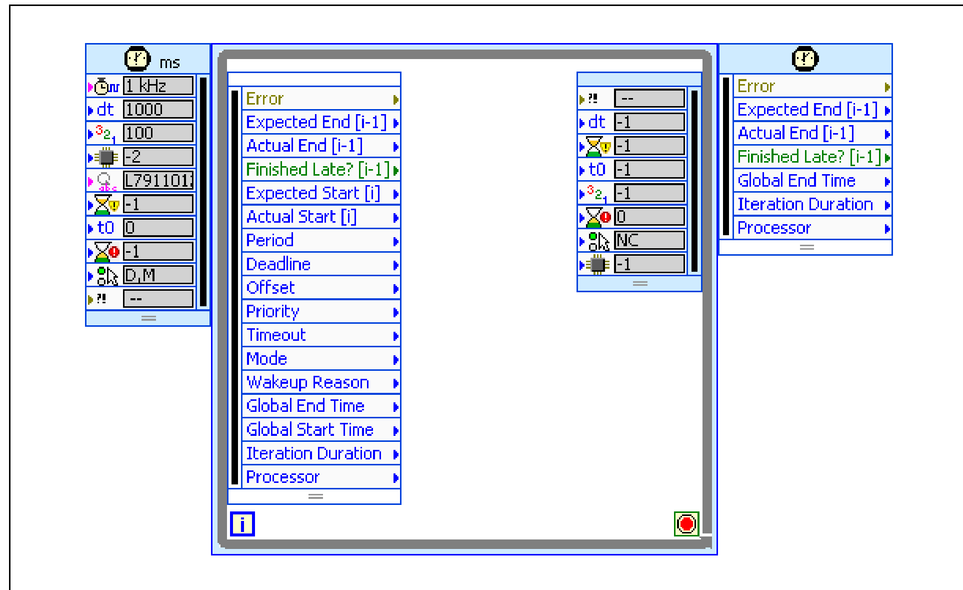


Figure 3-6. A Timed Loop

You can set configuration options of the Timed Loop by wiring values to the inputs of the Input node, or you can use the Loop Configuration dialog box to enter values for the options. By default, the inputs of the Input node appear as icons with the values you specified in the Loop Configuration dialog box. Refer to the *Timed Loop – Configuration* section for more information about configuring a Timed Loop.

The Left Data node of the Timed Loop provides timing and status information about the previous loop iteration, such as if the iteration executed late, the time the iteration actually began executing, and when the iteration should have executed. You can wire data from the Left Data node to the Right Data node to configure future iterations of the Timed Loop. You can resize the Left Data and Right Data nodes. Refer to the *Timed Loop – Changing Input Node Values Dynamically* section for more information on using the Left Data and Right Data nodes.

The Output node returns information from the final iteration of the While Loop, including whether the final iteration completed on time, and any errors that occurred during loop execution.

Timed Loops execute at a priority below the time-critical priority of any VI but above high priority, which means that Timed Loops execute in the data flow of a block diagram ahead of any VI not configured to run at a time-critical priority.

LabVIEW executes timed structures threads below time-critical priority and above high priority. You can specify the priority level of Timed Loops relative to other timed structures within a VI by setting the priority of the Timed Loop. Use the Configure Timed Loop dialog box to configure a timing source, period, priority, and other advanced options for the execution of the Timed Loop.

The higher the priority of a timed structure, the higher the priority the structure has relative to other timed structures and code on the block diagram. All timed structures execute at a priority relative to the LabVIEW execution system, between high and time-critical priority. To avoid priority inversions, National Instruments recommends using timed structures only in VIs set to normal priority.

Timed Loop – Configuration

Use the Configure Timed Loop dialog box to configure how the Timed Loop executes. Double-click the **Input** node or right-click the **Input** node and select **Configure Input Node** to display the Configure Timed Loop dialog box.

Use this dialog box to specify a timing source, period, offset timing, and other options. When you wire a value to an input node terminal, the corresponding field in the Configure Timed Loop dialog box becomes disabled.

After the loop begins, you can use the Right Data node to dynamically adjust the period, offset, priorities, and mode values for the Timed Loop. The updates take effect the next iteration of the loop. Refer to the *Timed Loop – Changing Input Node Values Dynamically* section for more information about dynamically adjusting the Timed Loop values.

Timed Loop – Setting Priorities

Each Timed Loop on the block diagram creates and runs in its own execution system that contains a single thread, so no parallel tasks can occur. The priority of a Timed Loop specifies when the loop executes on the block diagram relative to other Timed Loops. Use the priority setting of a Timed Loop to write applications with multiple tasks that can preempt each other in the same VI. The higher the value you enter in Priority of the Timed Loop, the higher the priority the Timed Loop has relative to other Timed Loops on the block diagram. The value you enter in the Priority input must be a positive integer between 1 and 65,535.

If two Timed Loops have the same priority, the first Timed Loop that executes completes its execution before the other Timed Loop starts its execution.

Timed Loops execute at a priority below the time-critical priority of any VI but above high priority, which means that Timed Loops execute in the data flow of a block diagram ahead of any VI not configured to run at a time-critical priority.

Timed Loop – Timing Source

A timing source determines when a Timed Loop executes a loop iteration. By default, the Timed Loop uses the 1 kHz clock of the operating system as the timing source and can execute only once every 1 ms because that is the fastest speed at which the operating system timing source operates. If the system does not include a supported hardware device, the 1 kHz clock is the only timing source available. If the system does include a supported hardware device, you can select from other timing sources, such as the 1 µs in CPU cycles available on some real-time hardware; or events, such as the rising edge of a DAQ counter input or output; or the end-of-scan interrupt of a DAQ device.

A 1 MHz clock is available on controllers that use a Pentium 3 or 4 processor.

Use the Source type listbox in the Configure Timed Loop dialog box to select a timing source or use the Create Timing Source VI to programmatically select a timing source.

Timed Loop – Period and Offset

The period is the length of time between loop executions. The offset is the length of time the Timed Loop waits to execute the iterations. The timing source determines the time unit of the period and the offset. If the timing source is a 1 kHz clock, the unit of time for the period and the offset is in milliseconds. If the timing source is a 1 MHz clock on an RT target with a Pentium processor, the unit of time for the period and the offset is in microseconds. The time the first timing source starts determines the start time of the offset.

Timed Loop – Naming Timed Loops

By default, LabVIEW automatically identifies each Timed Loop you place on the block diagram with a unique name, which appears in the Loop name text box of the Loop Configuration dialog box. You can rename the Timed Loop by entering a name in this text box. You can use the unique name of the Timed Loop with the VIs on the Timed Structures palette to programmatically stop the Timed Loop and to synchronize a group of Timed Loops to use the same start time.

If a reentrant VI includes a Timed Loop and you use two or more instances of that reentrant VI as subVIs on a block diagram, you must programmatically change the name of the Timed Loop for each instance of

the reentrant VI. Ensure that the reentrant VI that includes the Timed Loop has an input terminal on the connector pane connected to a string control wired to the Structure name input of the Timed Loop on the block diagram. On the block diagram where two or more instances of the reentrant VI are used as a subVI, wire unique string values to the Structure name input on the reentrant subVI to uniquely identify each Timed Loop within each instance of the reentrant subVI.

Refer to the *Naming Timed Structures* topic of the *LabVIEW Help* for more information about naming Timed Loops. Refer to the *Suggestions for Using Execution Systems and Priorities* topic in the *LabVIEW Help* for more information about using reentrant VIs.

Timed Loop – Assigning Processors

LabVIEW is compatible with multi-processor and multi-core machines. When you execute LabVIEW code in a multi-processor environment, LabVIEW separates the code into threads and assigns threads to available processors. This makes efficient use of the processors because it prevents a processor from waiting on a particular thread. You can override the default processor assignment in LabVIEW by using a Timed Loop. Timed Loops allow you to assign a core or processor to each Timed Loop by specifying the processor number. Manually assigning processors can help to improve the determinism of time-critical code by causing it to execute on a dedicated processor while non-critical code shares the other processor(s). Manually assigning processors can also help to improve performance in some systems because threads do not need to be swapped on and off the processor as often. However, manually assigning processors generally uses the processors less efficiently, because a processor can be idle waiting for a particular thread while other threads are ready to execute.

Timed Loop – Modes

Occasionally, an iteration of a Timed Loop might execute later than the time you specified. The mode of the Timed Loop determines how the loop handles any late executions. Use the options in the Action on Late Iterations section of the Configure Timed Loop dialog box or the Mode input of the Input node to specify the mode a Timed Loop uses to handle the late execution of a Timed Loop iteration.

You can handle the late execution of a Timed Loop in the following ways:

- The LabVIEW Timed Loop Scheduler can align the execution with the original established schedule.
- The LabVIEW Timed Loop Scheduler can define a new schedule that starts at the current time.

- The Timed Loop can process the missed iterations.
- The Timed Loop can skip any missed iterations.

For example, if you set a Timed Loop with a period of 100 ms and an offset of 30 ms, you expect the first loop iteration to execute 30 ms after the first timing source starts running and in multiples of 100 ms after that at 130 ms, 230 ms, 330 ms, and so on. However, the first execution of the Timed Loop might occur after 240 ms have elapsed. Because other Timed Loops or hardware devices might already be running at the schedule you specified, you might want to align the late Timed Loop with the already running global schedule, which means the Timed Loop should align itself as quickly as possible with the schedule you specified. In this case, the next Timed Loop iteration would run at 330 ms and continue to run in multiples of 100 at 430 ms, 530 ms, and so on. If aligning the Timed Loop with other Timed Loops or other hardware devices is not important, the Timed Loop can run immediately and use the current time as its actual offset. In this case, the subsequent loop iterations would run at 240 ms, 340 ms, 440 ms, and so on.

If the Timed Loop is late, it might miss data other Timed Loops or hardware devices generate. For example, if the Timed Loop missed two iterations and some of the data from the current period, a buffer could hold the data from the missed iterations. You might want the Timed Loop to process the missed data before it aligns with the schedule you specified. However, a Timed Loop that processes the missed iterations causes jitter. If you do not want to process the missed data, the Timed Loop can ignore the older data in the buffer that the loop iterations missed and process only the latest data, such as the data available at the next period and the subsequent iterations.

Timed Loop – Changing Input Node Values Dynamically

Use the Left Data node to acquire information about the execution of the Timed Loop, such as if the timing source executed late or if the offset or period values changed. You can wire the values the Left Data node returns to the Right Data node or to nodes in the subdiagram within the Timed Loop.

Use the Right Data node to dynamically change the input values of the Timed Loop on the next loop iteration.

If you dynamically change the offset of the Timed Loop by wiring a value to the Offset input of the Right Data node, you also must specify a mode with the Mode input of the Right Data node. To set the mode, right-click the Mode input of the Right Data node and select **Create»Constant** or **Create»Control** to create an enumerated constant or control you can use to select a mode.

Timed Loop – Aborting Execution

Use the Stop Timed Structure VI to abort the execution of a Timed Loop programmatically. Specify the name of the Timed Loop you want to abort by wiring that name in a string constant or control to the name input of the Stop Timed Structure VI.

Synchronizing Timed Loop Starts

Use the Synchronize Timed Structure Starts VI to ensure all Timed Loops on a block diagram use the same start time and the same timing source. For example, you might have two Timed Loops and you want to ensure that they execute on the same schedule relative to each other. You might want the first Timed Loop to execute first and generate data, then have the second Timed Loop process that data when the Timed Loop execution finishes. To ensure that both Timed Loops use the same start time as the basis for their execution, you create a Timed Loop group by wiring a name for the group to the synchronization group name input and wiring an array of Timed Loop names to the structure names input.

Use the Timed Loop when you want to develop VIs with multirate timing capabilities, feedback on loop execution, timing characteristics that change dynamically, or several levels of execution priority. Use the VI Priority method when you do not need any of the added functionality in Timed Loops and wish to run your code faster. The VI Priority method uses less overhead than the Timed Loop method. The VI Priority method also allows you to set a VI to a wider range of priorities such as background priority which runs below normal priority tasks. Timed Loops allow you to set code to a higher number of priorities, but all Timed Loops must run at a priority between high and time-critical.

C. Yielding Execution in Deterministic Loops

Because of the preemptive nature of deterministic loops, they can monopolize processor resources. In a single-core system, a deterministic loop might use all the processor resources, not allowing non-deterministic loops in the application to execute. You must build deterministic loops that periodically yield or sleep, to allow lower priority tasks to execute without affecting the determinism of the deterministic loop. In a multiple CPU system, you can put a deterministic loop on a dedicated core.

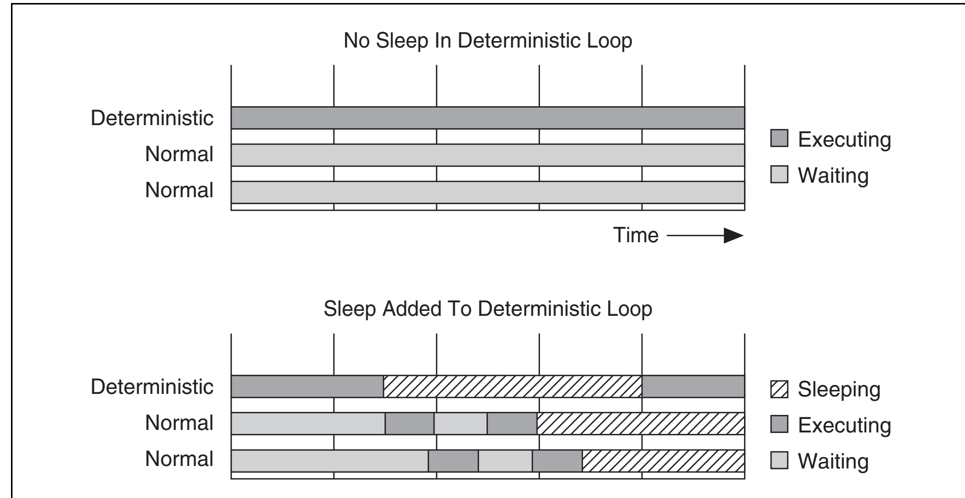


Figure 3-7. Yielding Execution in Deterministic Loops

Consider sleep mode as a programmatic tool that a VI can use to proactively remove itself from the LabVIEW and operating system scheduling mechanisms. Sleeping pauses the execution of a VI or a thread. By taking advantage of sleep mode, you can allow a lower priority VI to run by putting a higher priority VI to sleep.

In the top graph in Figure 3-7, the deterministic Loop starves the two normal threads because no sleep has been placed in the deterministic loop.

Starvation

To understand starvation, consider Figure 3-8.

Three processes—A, B, and C—compete for a resource. Because process A is a deterministic loop and has the highest priority, it runs until it finishes using the resource, thus freeing the resource. At that point, another process may use the resource—in this situation, either process B or process C. If process A is ready to run again, it takes control of the resource and runs until it is finished with the deterministic loop code. Again, one of the two processes may run. If process A does not sleep long enough for both processes to run, a lower priority process may never run. This situation is called starvation.

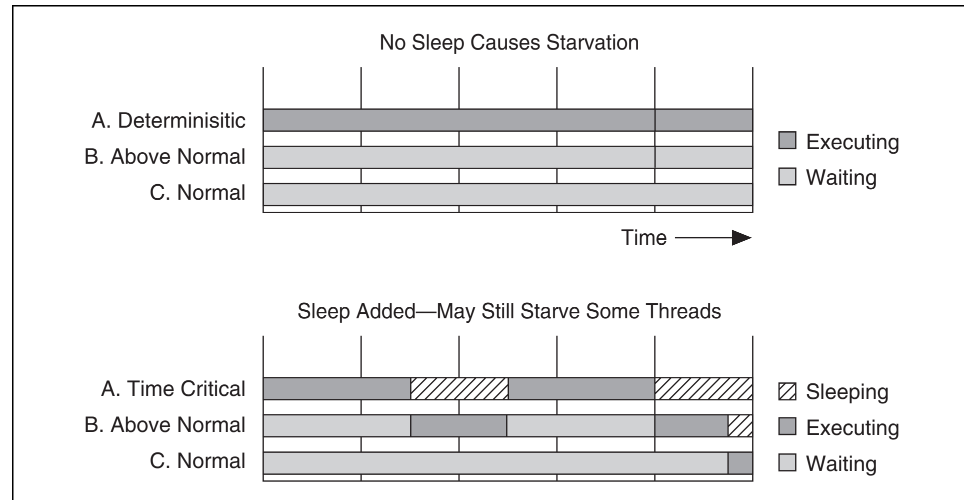


Figure 3-8. Starvation

Initially, process A has the resource and processes B and C wait for process A to sleep so that they may use the resource. When process A sleeps, the next highest priority process runs. In this case, because process B has a higher priority than process C, process B may run before process C. When the sleep time of process A ends, process A takes the resource back from process B. If this situation continues indefinitely, process C may be blocked from the resource and become starved. To prevent starvation of a lower priority process, the higher priority process must sleep long enough to allow lower priority processes time to execute.

Providing Sleep

You can program a sleep mode in a VI by using the timing functions or by using a Timed Loop. When controlling the rate of a software loop by using the Wait Until Next ms Multiple function from the Timing palette, you only can achieve rates in 1 ms multiples. This means you can run the loop at full speed, providing no sleep at all, or you can achieve loop rates of 1,000 Hz, 500 Hz, 333.33 Hz, 250 Hz, 200 Hz, and so on. However, if your controller has at least a Pentium 3 or 4 class processor, you can use the timing functions from the Real-Time Timing palette or the Timed Loop to achieve µs wait times, which adds to the available loop rates.

You can provide sleep using the data acquisition hardware on the PXI platforms. Use the DAQmx timing VIs to harness the clock on your data acquisition hardware.

Refer to Lesson 4, *Timing Applications and Acquiring Data*, for more information about programming sleep into your deterministic loop.

Sleeping and Deterministic Loops

The time-critical priority (highest) of the LabVIEW Real-Time Module threads have a unique characteristic that is different from normal LabVIEW scheduling. If any VI running in the time-critical priority (highest) thread sleeps, then the entire thread sleeps. Other VIs running on the thread are forced to sleep and pause execution until the original VI wakes up. 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 (highest), and one of them goes to sleep, the other VIs on the same thread continue to execute. In other words, the execution system of the LabVIEW Real-Time Module does not schedule time-critical priority (highest) operations from parallel VIs or loops, when any one of them sleep in the same time-critical priority (highest) thread. All other priority threads in LabVIEW Real-Time, and all threads in normal LabVIEW, 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 you assign the time-critical priority (highest) setting to only one VI or loop. This is the only way to guarantee deterministic execution.

If more than one time-critical VI (or loop) is needed to achieve different loop rates, you can use a Timed Loop instead of a time-critical priority VI.

Avoid parallelism inside a time-critical priority VI or deterministic Timed Loop, because the code executes serially on the processor. A Wait VI or Wait Until Next Multiple VI in a time-critical priority VI or deterministic Timed Loop will execute serially with the rest of the code in the thread even if it is placed in parallel. All VIs set to time-critical VI priority execute in the time-critical priority thread. Each Timed Loop executes in its own thread.

D. Improving Speed and Determinism

The easiest way to improve determinism is to choose a faster hardware platform. If you are unable to achieve a desired loop rate, first check your hardware to be sure it is capable of reaching the required rate.

If your hardware rates are acceptable, you can improve the determinism of your application in software by avoiding shared resources, contiguous memory conflicts, and subVI overhead. Because the LabVIEW memory manager is a shared resource, using memory reduction techniques also helps to improve the determinism of an application.

As described in the *Yielding Execution in Deterministic Loops* section, you should use only one deterministic loop per core.

The following sections explain the remaining programming methods for improving determinism.

Avoid Shared Resources

In LabVIEW, two or more VIs might need to share resources. Shared resources can cause jitter and prevent applications from taking advantage of multiple CPUs. 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. You cannot access global variables simultaneously from multiple processes. Therefore, compilers automatically protect the global variable as a shared resource while one process needs to access it. Meanwhile, if a second process tries to access the global variable while it is protected, the second process must wait until the first process finishes with the global variable. Understanding shared resources and how to identify them is an important skill when programming real-time applications.

LabVIEW Real-Time shared resources include the following:

- Global variables
- LabVIEW memory manager
- Single-threaded DLLs
- Shared variables
- Non-reentrant subVIs
- Networking code (TCP/IP, UDP, VI Server)*
- File I/O*
- Semaphore VIs*

Note The operations marked with an asterisk are inherently non-deterministic. Never use them inside a time-critical priority loop if you are attempting to achieve real-time performance.

Avoid Shared Resources – Priorities

Imagine a scenario where there is a shared resource, such as a global variable, that is shared by two VIs—one set to normal priority and one set to time-critical priority.

The RTOS uses *priority inheritance* to resolve the priority inversion as quickly as possible using the following procedure:

- Allow the lower priority thread to temporarily inherit the time-critical priority setting long enough to finish using the shared resource and release the mutex.
- After releasing the mutex, the lower priority thread resumes its normal priority setting and is taken off the processor.
- The time-critical priority thread proceeds to use the resource, that is, access the global variable.

The priority inversion increases software jitter in the time-critical priority thread. The jitter induced by a mutexed global variable is small compared to the jitter induced by a mutexed 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.

Shared Resources – SubVIs

Sharing subVIs can cause priority inversions the same as global variables. You can set a VI to subroutine priority and select the **Skip Subroutine Call If Busy** option to skip that VI within time-critical code and avoid software jitter that occurs from a priority inversion.

However, if you run unrelated parallel processes that call the same VI, you can configure the VI for *reentrant execution*. Use a reentrant VI to allow multiple instances of a VI to execute in parallel with distinct and separate data storage. LabVIEW RT can call multiple instances of a reentrant VI simultaneously. Because reentrant VIs use their own data space, you cannot use them to share or communicate data between threads. You should use reentrancy only when you must simultaneously run multiple instances of a VI within unrelated processes that do not need to share data within the reentrant VI.

To make a VI reentrant, select **File»VI Properties**, select **Execution** in the VI Properties dialog box and place a checkmark in the **Reentrant execution** checkbox.

Note Use reentrant VIs carefully because each call to the VI establishes a unique data space for all controls and indicators in memory and thus uses more memory. For this reason, you cannot use reentrant VIs as functional global variables. Also, making the DAQ VIs reentrant does not solve the shared resource problem with `NIDAQ32.dll`. All DAQ VIs load the DAQ driver, `NIDAQ32.dll`. This DLL is not multithreaded safe, so it cannot be called simultaneously from multiple processes.

Shared Resources – Memory Management

When a VI allocates memory, the VI accesses the LabVIEW memory manager. The LabVIEW memory manager allocates memory for data storage. The LabVIEW memory manager is a shared resource and might be locked by a mutex for up to several milliseconds. Allocating memory within a deterministic VI can affect the determinism of the VI.

If you allow LabVIEW to dynamically allocate memory at run time, your application could suffer from software jitter for the following reasons:

- The memory manager may already be mutexed, causing a shared resource conflict.
- If the memory manager is immediately available, allocating memory is non-deterministic because there is no upper bound on the execution time of the memory allocation.

Preallocate Arrays

Avoid allocating memory within a time-critical loop. If you use arrays in deterministic loops, you can reduce jitter by preallocating the arrays before entering the loop. For example, instead of using the Build Array function within your loop to index new data into an array, use the Initialize Array function outside the loop and the Replace Array Subset function inside the loop to create the array. Because the array is preallocated outside the loop, the loop no longer needs to access the LabVIEW memory manager at every iteration.

Shared Resources – Memory Management Summary

In general, memory allocations within a deterministic loop induce jitter and affect the deterministic properties of a LabVIEW Real-Time Module application. All memory allocations must be removed to guarantee robust real-time performance. Preallocate arrays outside the loop if you want to prevent jitter. Certain LabVIEW functions allocate memory, such as the Build Array and Bundle functions. Refer to the *Memory Manager*

Structures and Functions topic in the *LabVIEW Help* for more information about functions that allocate memory.

Cast data to the proper data type in VIs running on the RT target. Each time LabVIEW performs a type conversion, LabVIEW makes a copy of the data buffer in memory to retain the new data type after the conversion. The LabVIEW memory manager must allocate memory for the copy, which can affect the determinism of time-critical VIs. Also, creating copies of the data buffer takes up memory resources on an RT target. Use the smallest data type possible when casting the data type. If you must convert the data type of an array, perform the conversion before you build the array.

Keep in mind that a function output reuses an input buffer only if the output and the input have the same data type, including the same representation, size, and dimension. Arrays must have the same structure and number of elements for function outputs to reuse the input buffer. This ability for a function to reuse buffers is called *inplaceness*. You can use the Show Buffer Allocations window to identify specific areas on the block diagram where LabVIEW allocates memory.

LabVIEW creates an extra copy in memory of every global variable you use in a VI. Reduce the number of global variables to improve the efficiency and performance of VIs. Creating copies of the global variable takes up memory resources on an RT target.

This course does not discuss all types of shared resources. Avoid Semaphore VIs, TCP/IP, UDP, VI Server, and File I/O functions within a deterministic loop. These functions are inherently non-deterministic and use shared resources. For example, Semaphore VIs 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 deterministic code due to priority inversions.

Also, all handshaking protocols are non-deterministic. Do not run GPIB, RS-232, or TCP/IP at time-critical priority. DAQ handshaking protocols, such as burst mode and 8255 emulation mode on the 653x devices, are non-deterministic, so avoid using them in deterministic loops.

Avoid Contiguous Memory Conflicts

LabVIEW handles many of the memory details that you normally handle in a conventional, text-based language. For example, functions that generate data must allocate storage for the data. When that data is no longer needed, LabVIEW deallocates the associated memory. When you add new information to an array or a string, LabVIEW allocates new memory to accommodate the new array or string. However, running out of memory is a concern with VIs running on an RT target.

You must design memory-conscious VIs for RT targets. Always preallocate space for arrays equal to the largest array size that you might encounter.

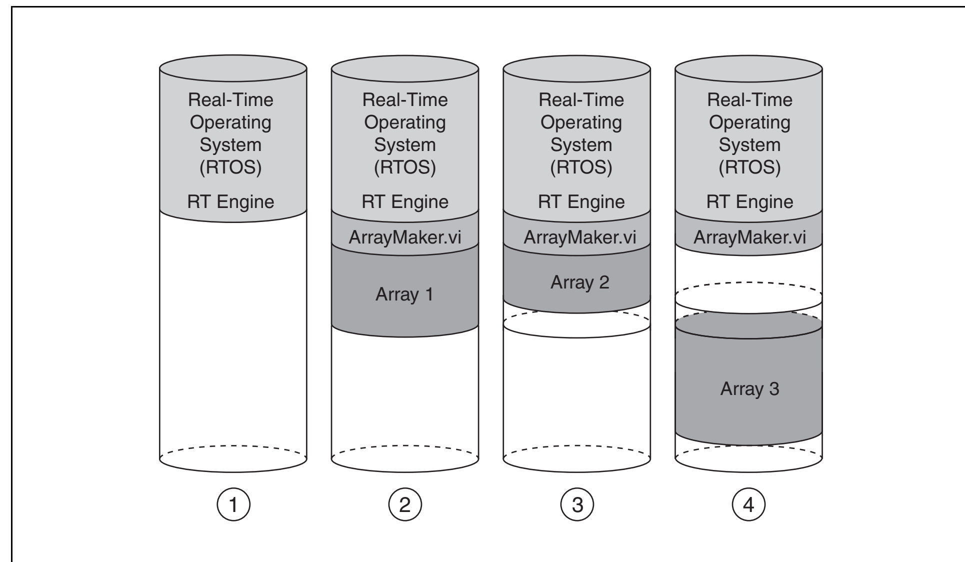


Figure 3-9. Avoid Contiguous Memory Conflicts

When you reboot or reset an RT target, the RTOS and the RT Engine load into memory as shown in diagram 1 of Figure 3-9.

The RT Engine uses available memory for running RT target VIs and storing data. In diagram 2 of Figure 3-9, `ArrayMaker.vi` creates Array 1. All elements in Array 1 must be contiguous in memory.

The RTOS reuses the same memory addresses if you stop a VI and then run it again with arrays of the same size or smaller. In diagram 3 of Figure 3-9, `ArrayMaker.vi` creates Array 2. The RTOS creates Array 2 in the reserved memory space previously occupied by Array 1. Array 2 is small enough to fit in the reserved memory space that was allocated to Array 1. The extra contiguous memory used for Array 1 remains in the reserved memory space, as shown in diagram 3.

When `ArrayMaker.vi` runs for a third time with a larger array or if another VI generates a larger array, the RT Engine must find a large enough contiguous space. In diagram 4 of Figure 3-9, `ArrayMaker.vi` must create Array 3, larger than the previous arrays, in the available memory.

Even when `ArrayMaker.vi` stops running, the RT Engine continues to run and previously reserved memory is not available. If `ArrayMaker.vi` runs a fourth time and attempts to create an array larger than Array 3, the operation fails. There is no contiguous memory area large enough to create the array because of the memory fragmentation. You can preserve memory

space by preallocating array space equal to the largest use case, as shown in the Preallocate Arrays section of this lesson.

In Place Element Structure

Use the In Place Element structure to control how the LabVIEW compiler performs common operations, such as operating on an element of an array and placing the resulting value back into the same array index, and to increase memory and VI efficiency. Many LabVIEW operations require LabVIEW to copy and maintain data values in memory, thereby decreasing execution speed and increasing memory usage. The In Place Element structure performs common LabVIEW operations without LabVIEW making multiple copies of the data values in memory. Instead, the In Place Element structure operates on data elements in the same memory location and returns those elements to the same location in the array, cluster, variant, or waveform. Because LabVIEW returns the data elements to the same location in memory, the LabVIEW compiler does not have to make extra copies of the data in memory.

The In Place Element structure is useful for updating clusters in deterministic applications. Refer to the *In Place Element Structure* topic of the *LabVIEW Help* for more information about using the structure to increase memory efficiency in VIs.

Avoid SubVI Overhead

Calling a subVI from a VI running on an RT target adds a small amount of overhead to the overall application. Although the overhead is small, calling a subVI multiple times in a loop can add a significant amount of overhead. You can embed the loop in the subVI to reduce the overhead.

The overhead involved in calling a subVI increases depending on the amount of memory that must be copied by the memory manager.

You also can convert subVIs into subroutines by changing the VI priority. The LabVIEW execution system minimizes the overhead to call subroutines. Subroutines are short, frequently executed tasks that generally do not require user interaction. Subroutines cannot display front panel data and do not multitask with other VIs. Avoid using timing or dialog box functions in subroutines.

Setting VI Properties

To reduce memory requirements and increase performance of VIs, disable nonessential options in the VI Properties dialog box available by right-clicking a VI In the Project Explorer window and selecting **Properties**. Select **Execution** from the **Category** pull-down menu and remove checkmarks from the **Allow debugging** and **Auto handle menus**

at launch checkboxes. By disabling these options, VIs use less memory, compile more quickly, and perform better.

Note The LabVIEW Real-Time Module ignores the **Enable automatic error handling** option.

Use Low-Level Functions to Increase Execution Speed

LabVIEW Express VIs increase LabVIEW ease of use and improve productivity with interactive dialog boxes that minimize programming for applications. However, Express VIs may require additional performance overhead during execution and perform optional tasks. Use low-level functions to increase execution speed with finer control.

E. Sharing Data Locally on RT Target

After dividing tasks in an application into separate Timed Loops or VIs of different priorities, you might need to communicate between the loops on a block diagram of between the different VIs on the RT target. You can use the following techniques to send and receive data between VIs or loops in an application:

- Single-process shared variables with the Real-Time FIFO enabled
- Functional global variables
- RT FIFO (first in, first out buffer) functions

This course discusses the single-process shared variables with Real-Time FIFO enabled method. Refer to the *LabVIEW Real-Time 2* course for information on the other techniques.

Single-Process Shared Variables with the RT FIFO Enabled

Single-process shared variables with the RT FIFO enabled are the preferred communication method for deterministic data transfer between VIs or loops, so this course focuses on their implementation.

Use single-process shared variables to share data between two locations in a block diagram or between VIs running on an RT target. Right-click an RT target in the Project Explorer window and select **New»Variable** from the shortcut menu to open the Shared Variable Properties dialog box, which you can use to create a single-process shared variable.

The Real-Time Module adds real-time FIFO capability to the shared variable. By enabling the real-time FIFO of a shared variable, you can share data without affecting the determinism of VIs running on an RT target. From the Real-Time FIFO page of the Shared Variable Properties dialog box,

place a checkmark in the **Enable Real-Time FIFO** checkbox to enable the real-time FIFO of a shared variable.

Single-process shared variables provide a communication method that is easy to use and deterministic when you enable the Real-Time FIFO.

How Are Shared Variables Used?

Shared variables are designed to facilitate communication in LabVIEW. You can configure shared variables to perform many tasks:

- Transfer non-deterministic data between loops or VIs on a single target. In this capacity, a shared variable functions much like a global variable. This type of shared variable is called a *single-process shared variable*.
- Transfer data from a non-deterministic loop to a host. This type of shared variable is a *network-published shared variable*. Refer to Lesson 4, *Timing Applications and Acquiring Data*, for more information on communicating between a target and host.
- Transfer non-deterministic data between hosts or between a host and other computers. Shared variables implement a publisher/subscriber model that allows non real-time computers to communicate across a network. This type of shared variable is called a *network-published shared variable*.
- Transfer deterministic data between Real-Time VIs or loops (Real-Time FIFO). Shared variables can implement a Real-Time FIFO to transfer data deterministically between loops on an RT target. This type of variable is usually a *single-process shared variable with the Real-Time FIFO option enabled*.
- Transfer deterministic data between targets. With a dedicated network connection, shared variables can deterministically transfer data between two or more real-time targets over a network. This type of variable is called a *Time-Triggered shared variable*.

  For more information on the Time-Triggered shared variable, refer to the *Using Time-Triggered Networks to Communicate Deterministically Over Ethernet with the LabVIEW 8 Real-Time Module* document in the NI Developer Zone. To view the document, visit `ni.com/info` and enter `rduttl`.

Creating Shared Variables

To create a shared variable, right-click a target or library in the Project Explorer window and select **New»Variable**. All variables must exist inside a library. If you create a new variable outside a library, LabVIEW automatically creates a library. When you create a shared variable, LabVIEW displays a Shared Variable Properties dialog box in which you configure the type of shared variable and any other options such as buffering and Real-Time FIFO.

To use a shared variable on the block diagram, drag the shared variable from the Project Explorer window to the block diagram. Shared variable references on the block diagram work much like local or global variables. You can right-click a shared variable and select **Change to Read** or **Change to Write** to change the direction of the variable. Shared variables contain additional terminals along with the data. Each shared variable has error in and error out terminals, and shared variables set to read can return a timestamp indicating when the data was written. To add a **timestamp** output to a single-process shared variable, you must first place a checkmark in the **Enable timestamp** checkbox on the Variable page of the Shared Variable Properties dialog box, and then right-click the Shared Variable node and select **Show Timestamp**.

Shared Variable with the Real-Time FIFO Enabled

When you enable the Real-Time FIFO option on the Shared Variable Properties page, LabVIEW uses Real-Time FIFOs to transfer the data that is written to and read from the shared variable. You can configure the FIFO to be single or multi-element and define the size of the FIFO. When you enable the Real-Time FIFO option, a small icon appears on references to the variable to indicate that it uses Real-Time FIFOs.

Single Element FIFO

A single-element FIFO shares the most recent data value. The shared variable overwrites the data value when it receives a new data value. Use this option when you need only the most recent value. Configure the size of the array elements or the size of the waveform for the FIFO buffer if you select an array or waveform data type.

Multi-Element FIFO

A multi-element FIFO buffers the values shared by the shared variable. You can configure the number and data type of the FIFO buffer elements to match the settings from the Use Buffering section of the Variable page, or you can configure a custom size for the FIFO and the FIFO elements.

Note For both single-element and multi-element FIFOs, if the variable contains array or waveform data, you must configure the size of the FIFO elements equal to the size of the

data you want to share. If both the network buffer and the RT FIFO are enabled, the network buffer must be at least as large as one FIFO element. Sharing data smaller or larger than the length you specify causes a memory allocation that affects determinism.

Programming Shared Variable FIFOs – Initialization

Shared variable FIFOs are created the first time a variable is read from or written to. This results in a slight delay. Therefore, either initialize the variable by reading from or writing to it before your main loop or allow for a delay in the first iteration of your loop as the FIFO is created.

Programming Shared Variable FIFOs – Overflow

Multi-element RT FIFOs have a fixed memory size and a fixed number of elements, which you configure in the Shared Variable Properties dialog box. Therefore, multi-element shared variable RT FIFOs introduce the possibility for overflow and underflow errors.

An overflow error occurs when a shared variable reference attempts to write to an RT FIFO that is already full. When an overflow occurs, the shared variable returns error `-2221` and overwrites the oldest value in the FIFO with the new value. The oldest value is permanently lost.

Programming Shared Variable FIFOs – Underflow

An underflow error occurs when a shared variable reference attempts to read an empty RT FIFO. When an underflow occurs, the shared variable returns error `-2220` and returns a default value for the data item. This is different from error `-2222`, which only applies if a variable has never been written to. Also, error `-2220` applies only to multi-element FIFOs, whereas error `-2222` applies to all shared variables.

Programming Shared Variable FIFOs – Multiple Readers and Writers

LabVIEW creates a single, real-time FIFO for each single-process shared variable even if the shared variable has multiple writers or readers. To ensure data integrity, multiple writers block each other as do multiple readers. Only a single reader and a single writer can access a shared variable at the same time. However, a reader does not block a writer, and a writer does not block a reader. If a single variable has multiple readers and writers, the readers and writers alternate in accessing the variable like any other resource. Variable references waiting on another reader or writer are blocked and do not continue block diagram execution.

Because variables can block with multiple readers or writers, when using variables in a deterministic loop, ensure that a variable read by a deterministic loop cannot be read by another loop and that a variable written by a deterministic loop cannot be written to by another loop. Failure to

follow these rules can cause the deterministic loop to block and execute non-deterministically.

By enabling the real-time FIFO, you can select between two slightly different types of FIFO-enabled variables: the single-element and the multi-element buffer. One distinction between these two types of buffers is that the single-element FIFO does not report warnings on overflow or underflow conditions. A second distinction is the value that LabVIEW returns when multiple readers read an empty buffer. Multiple readers of the single-element FIFO receive the same value, and the single-element FIFO returns the same value until a writer writes to that variable again. Multiple readers of an empty multi-element FIFO each get the last value that they read from the buffer or the default value for the data type of the variable if they have not read from the variable before.

If an application requires that each reader get every data point written to a multi-element FIFO shared variable, use a separate shared variable for each reader.

RT FIFO Functions

If your application requires programmatic control of the RT FIFO, use RT FIFO functions instead of shared variables for inter-task communication. For example, you can use RT FIFO functions to:

- Programmatically create and delete RT FIFOs
- Programmatically set the number of elements in RT FIFOs
- Set timeouts and timeout behavior
- Read the number of elements remaining in RT FIFOs

RT FIFO functions and other inter-task communication methods are covered in detail in the *LabVIEW Real-Time 2* course.

Summary – Quiz

1. Which of the following are methods for improving speed and determinism in code?
   a. Avoid file I/O functions
   b. Cast all data to the proper type
   c. Use the Build Array function to dynamically build arrays
   d. Disable non-essential VI options

2. True or False? You should only use one time-critical VI or loop per CPU in a deterministic application.

3. True or False? It is good practice to use Timed Loops within VIs set to background, above normal, high, or time-critical priority.

Summary – Quiz Answers

1. Which of the following are methods for improving speed and determinism in code?

   a. Avoid file I/O functions

   b. Cast all data to the proper type

   c. Use the Build Array function to dynamically build arrays

   d. Disable non-essential VI options

2. True or False? You should only use one time-critical VI or loop per CPU in a deterministic application.

   True

3. True or False? It is good practice to use Timed Loops within VIs set to background, above normal, high, or time-critical priority.

   False

Notes

4

Timing Applications and Acquiring Data

In this lesson, you develop the deterministic loop of a target application. This typically involves control parameters, hardware input and output, and timing. This lesson focuses on software and hardware methods of timing a loop in a real-time application.

Topics

A. Timing Control Loops

B. Software Timing

C. Hardware Timing

D. Event Response – Monitoring for Events

A. Timing Control Loops

The preemptive nature of the RTOS on RT series devices can cause a deterministic loop to monopolize the processor on the device. On a single-core system, a deterministic loop might use all processor resources and not allow lower priority threads in the application to execute. Unless the deterministic loop is isolated on its own core, the deterministic loop must periodically yield processor resources to the lower-priority tasks so they can execute. By properly separating the deterministic task from lower priority non-deterministic tasks, you can reduce application jitter.

You can use software methods or hardware methods to time control loops.

B. Software Timing

LabVIEW provides multiple software timing methods. You can insert the LabVIEW Wait function, Wait Until Next Multiple function, or Wait Express VI in your code to add sleep time. Alternately, you can use a Timed Structure, such as a Timed Loop, which controls execution speed and adds other benefits. Each of these methods has a millisecond resolution when used for software timing.

Wait

The Wait Express VI causes a VI to sleep for the specified amount of time. For example, if the operating system millisecond timer value is 112 ms when the Wait Express VI executes, and the **Count (mSec)** input equals 100, then the Express VI returns when the millisecond timer value equals 142 ms.

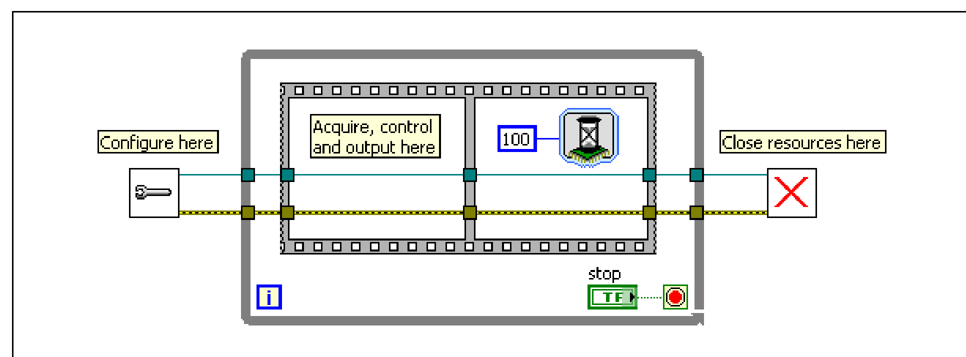


Figure 4-1. The Wait Express VI

Avoid using this Express VI in parallel with deterministic code. If the Wait Express VI executes first, the whole thread sleeps until the VI finishes, and the code in parallel does not execute until the Wait Express VI finishes. The resulting loop period is the code execution time plus the **Count (mSec)** time.

Wait Until Next Multiple

When you use the Wait Until Next Multiple Express VI, you can choose ticks, msec, or μsec resolution. The name of the **Count** input reflects the resolution you choose. When configured for ms resolution, the Wait Until Next Multiple Express VI causes a thread to sleep until the operating system ms timer value equals a multiple of the **Count (mSec)** input. For example, if the Wait Until Next Multiple Express VI executes with a **Count (mSec)** input of 100 ms and the operating system millisecond timer value is 112 ms, the VI sleeps until the millisecond timer value equals 200 ms because 200 ms is the first multiple of 100 ms after the Wait Until Next Multiple Express VI executes.

Use the Wait Until Next Multiple Express VI to synchronize a loop with the operating system millisecond timer value multiple. A loop has a period of **Count (mSec)** if the Wait Until Next Multiple Express VI executes in parallel with other code in the same loop. However, the loop does not have the period of **Count (mSec)** if the code takes longer to execute than the **Count (mSec)**.

However, avoid placing the Wait Until Next Multiple Express VI in parallel with other code because doing so can cause incorrect timing of a control system. The dataflow properties of LabVIEW programming can cause the Wait Until Next Multiple Express VI to execute before, after, or between the execution of the analog input and output. The behavior of the loop differs depending on when the Express VI executes. Instead, use a Sequence structure to control when the Express VI executes, as shown in Figure 4-2.

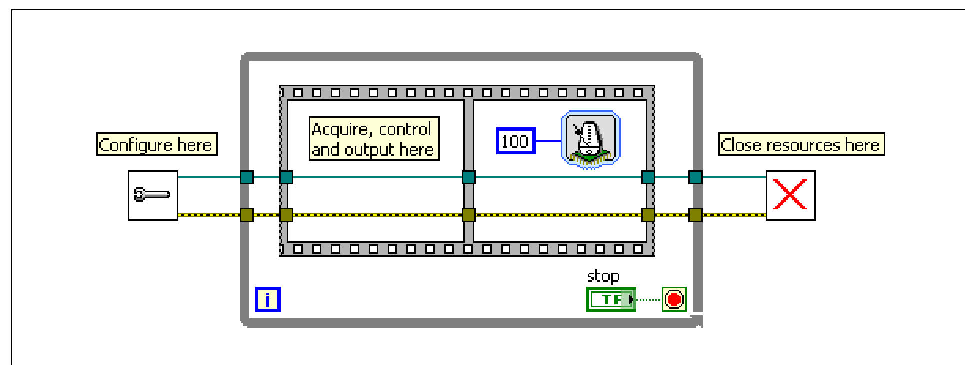


Figure 4-2. The Wait Until Next Multiple Express VI

In the Figure 4-2, the code may take a variable amount of time to finish executing, but calling the Wait Until Next Multiple Express VI afterwards enforces a loop frequency of 10 Hz (1/100 ms). The maximum achievable loop rate is 1 kHz with a wait multiple of 1 ms.

Because the Wait Until Next Multiple Express VI accepts only integers, loop rates are limited to only a few frequencies: 1000, 500, ~333, 250, 200, ~167 Hz, and so on.

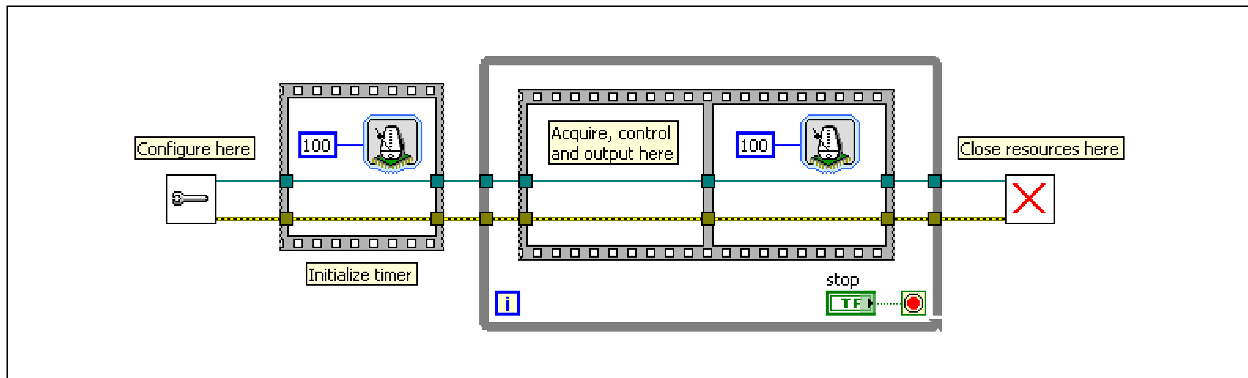


Figure 4-3. Initialized Wait Until Next Multiple Express VI

In Figure 4-3, the 100 ms timer is initialized by calling the Wait Until Next Multiple Express VI immediately before the While Loop begins. Otherwise, the loop time for the first iteration would be indeterminate. In the While Loop, placing the Wait Until Next Multiple Express VI in a sequence structure adds the delay after the code has finished. This guarantees the order of execution.

Before deciding on a **Count** value, you must ensure that the code in your loop can execute faster than the wait multiple. If the code inside the loop takes longer than the **Count** value, the loop must wait a second multiple of **Count**, because code was running when the first ms multiple arrived. In this case, the Wait Until Next Multiple Express VI is not aware that the first multiple occurred and waits until a second multiple occurs before returning.

In addition to controlling loop rates, the Wait Until Next Multiple Express VI forces time-critical VIs to sleep until the wait finishes. When a VI sleeps, it relinquishes the CPU, allowing other VIs or threads to execute. Unless the time-critical VI is isolated on its own core, sleep time is required, because the user interface and other background processes need CPU time to survive.

The Wait Until Next Multiple Express VI masks software jitter within the loop. The Express VI has some inherent jitter, which is acceptable for many real-time applications. In this example, the Wait Until Next Multiple Express VI synchronizes with each 100 ms tick of the OS clock, allowing the loop to achieve 10 Hz software-timed analog input. The timeline in Figure illustrates the 5 ms wait multiple at ΔT, the actual time required to execute the code at T_e, worst case jitter at T_j, and worst case time at T_{wc}. As long as the worst case time is smaller than the wait multiple, the actual loop

period equals the wait multiple, plus or minus the jitter incurred by the Express VI itself.

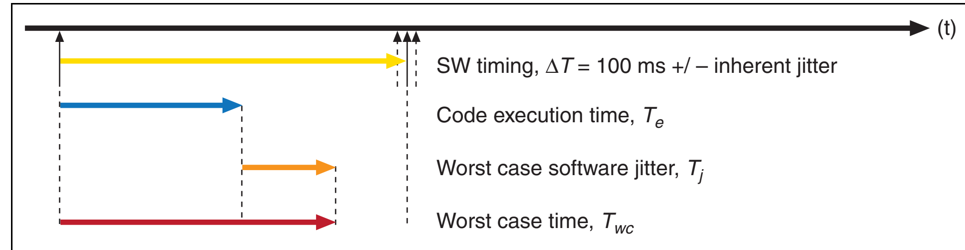


Figure 4-4. Software Timing Timeline

µs Timing

If you are targeted to an RT target that allows microsecond timing, you can use the microsecond clock for the wait Express VIs. If you are not targeted to an acceptable target, you can still select microsecond (µs) timing, but the program uses the millisecond (ms) operating system clock instead.

Using the microsecond clock allows for more loop rate options:

- With a ms clock, loop rates are *1/X* ms = 1 KHz, 500 Hz, ~333 Hz, 250 Hz, and so on
- With a µs clock, loop rates are *1/X* µs = 1 MHz, 500 KHz, ~333 KHz, 250 KHz, and so on

To use microsecond timing, double-click a Wait Express VI to open a configuration window, and set Counter Units to **µSec**.

Timed Loop

A Timed Loop executes an iteration of the loop at the period you specify. Use the Timed Loop when you want to develop VIs with multi-rate timing capabilities, feedback on loop execution, timing characteristics that change dynamically, or manual processor assignment.

Because the Timed Loop automatically imposes sleep as needed to achieve the loop rate you specify, there is no need to use a Wait or Wait Until Next Multiple Express VI to add sleep time in the loop.

Because of the preemptive nature of Timed Loops, they can monopolize processor resources. A Timed Loop might use all of the processor resources, not allowing other tasks on the block diagram to execute. You must configure the highest priority Timed Loop with a period large enough to perform the deterministic task and have idle time during every iteration to allow lower priority loops to execute.

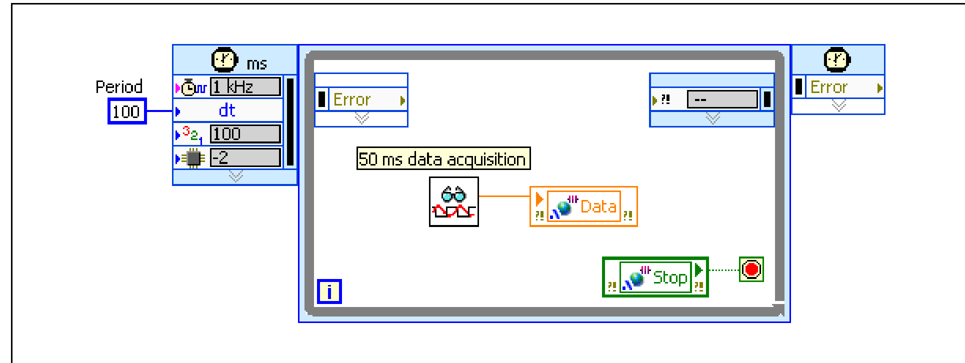


Figure 4-5. Timed Loop

Figure 4-5 contains a subVI that performs a data acquisition for 50 ms. The Timed Loop has a period of 100 ms that allows the loop to remain idle for 50 ms during each iteration. During the time when the Timed Loop remains idle, LabVIEW can execute lower priority tasks on the block diagram.

To configure the Timed Loop for microsecond timing, choose the MHz clock (μs timer) in the Loop Timing Source section of the Timed Loop configuration window if you are targeted to an appropriate target. Access the Timed Loop configuration window by double-clicking the Input node of the Timed Loop.

C. Hardware Timing

You can implement hardware timing in your real-time application by using external timing sources. The National Instruments drivers that run on RT targets support VIs or functions that can cause sleep in the current LabVIEW thread and return when the driver detects a specific event. For example, you can use NI-DAQmx and NI data acquisition hardware to time real-time applications. Refer to the specific NI driver documentation for information about VIs or functions that you can use to sleep and wait for driver events.

DAQmx

You can use NI data acquisition hardware and NI-DAQmx to achieve a sleep resolution much finer than 1 kHz. Hardware timing uses the DAQ device internal clock or an external clock to control timing. You can use the DAQmx VIs to control when a Read VI or a Write VI executes within a loop. Alternately, you can wire a DAQmx task to a Timed Loop to tie the loop rate to the hardware clock.

Use the DAQmx Timing VI to configure the sample clock, which controls the loop rate. You can configure the DAQmx task according to your application, however, the Hardware Timed Single Point option for the **sample mode** input provides the best access to the hardware clock. Notice that when the requested scan rate is too fast relative to the code execution time, you may miss ticks from the clock. In other words, if the RT target is not powerful enough to execute the code within the loop at least as fast as the scan rate, the clock rate will be slower than configured.

You can use NI data acquisition hardware with NI-DAQmx to match loop rates to match the rate of the hardware clock. With NI-DAQmx, you can use the following methods to time real-time applications:

- **Hardware-Timed Single-Point—**NI-DAQmx supports hardware-timed, single-point sample mode in which samples are acquired or generated continuously using hardware timing and no buffering. You can use hardware-timed, single-point mode for control applications that require input and/or output within a deterministic period of time. Refer to the *NI-DAQmx Single-Point Real-Time Applications* topic of the *NI-DAQmx Help* for information about using hardware-timed, single-point operations to time your deterministic application.
- **Counter Timers—**NI-DAQmx supports using hardware-timed counter input operations to drive a control loop. Use the Wait For Next Sample Clock VI to synchronize the counter operations with the counter's sample clock. Refer to the *Hardware-Timed Counter Tasks* topic of the *NI-DAQmx Help* for information about using counter input operations to time deterministic applications.
- **DAQmx Timing Sources for Timed Structures—**Timed structures can be hardware-timed and are ideal for multirate applications. By default, timed structures use the 1 kHz clock on Windows or the real-time operating system of an RT target as a timing source. You also can use an external signal on a DAQ device as the timing source of a timed structure using NI-DAQmx. Use the DAQmx Create Timing Source VI to create a timing source that can synchronize the timed structure with the hardware clock. Refer to the *Hardware-Timed Simultaneously Updated I/O Using the Timed Loop* topic of the *NI-DAQmx Help* for information about using an external signal on a DAQ device to control a timed structure.

You can create external timing sources for controlling a timed structure with NI-DAQmx. Use the DAQmx Create Timing Source VI to programmatically select an external timing source. You also can use several types of NI-DAQmx timing sources, including frequency, digital edge counters, digital change detection, and signals from task sources, to control timed structures. Use the DAQmx - Data Acquisition VIs to create the following types of NI-DAQmx timing sources to control a timed structure.

- **Frequency**—Creates a timing source that causes a timed structure to execute at a constant frequency.
- **Digital Edge Counter**—Creates a timing source that causes a timed structure to execute on rising or falling edges of a digital signal.
- **Digital Change Detection**—Creates a timing source that causes a timed structure to execute on rising or falling edges of one or more digital lines.
- **Signal from Task**—Creates a timing source that uses the signal you specify to determine when a timed structure executes.

Refer to the *NI-DAQmx Help*, available by selecting **Start»All Programs» National Instruments»NI-DAQ»NI-DAQmx Help**, for information about using NI-DAQmx VIs and functions to control timed structures.

D. Event Response – Monitoring for Events

With real-time event response, you can respond to a single event within a given amount of time. Some common events include detecting a peak in a measurement or detecting when a threshold has been reached. You can use the Point-by-Point Signal Analysis VIs to detect these types of events as shown in Figure 4-6.

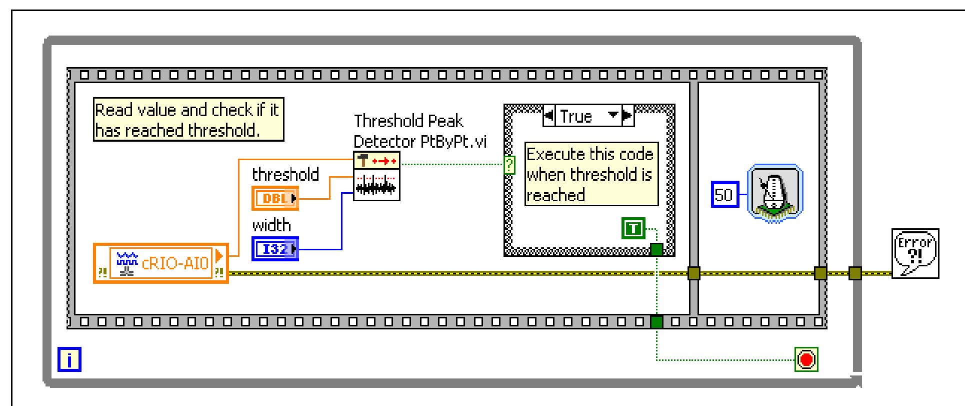


Figure 4-6. Monitoring for An Event Using A Point-by-Point VI

Event Response – Digital Change Detection

(NI-DAQmx only) Another common event response application involves watching for a digital line change, which is useful when watching for an alarm trigger. Figure 4-7 uses the DAQmx digital change functionality with the Timed Loop. The digital line is connected to the **Source Name** terminal of the Timed Loop. When the digital change happens or a timeout occurs, the Timed Loop wakes up and executes the code in the loop.

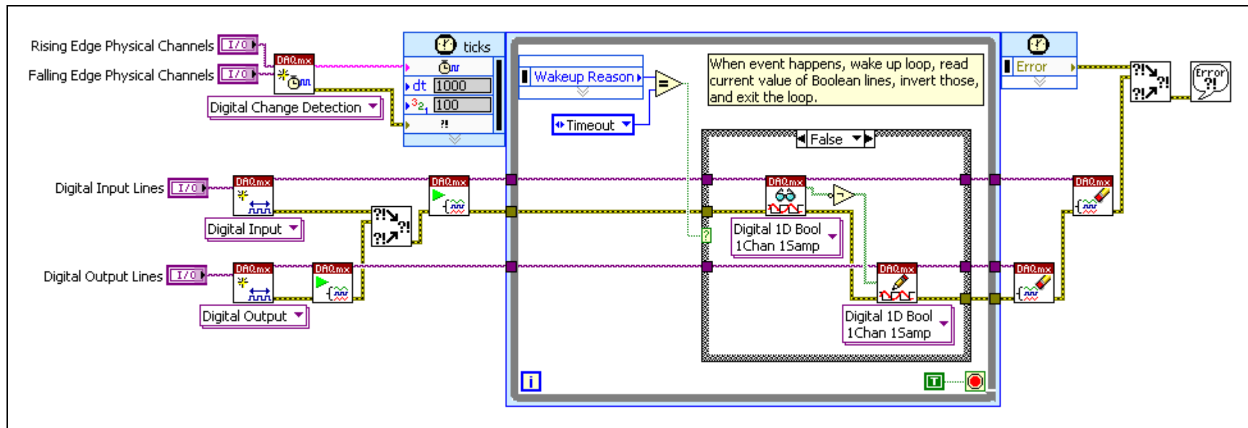


Figure 4-7. Digital Change Detection

Summary – Quiz

1. Which of the following are benefits of using timing in a control loop?
   a. Provide sleep so lower priority threads can execute
   b. Reduce application jitter
   c. Both a & b

2. True or False? It is good programming practice to use wait functions in parallel with time critical code.

3. Which of the following typically provides finer resolution?
   a. Hardware timing
   b. Software timing

4. Which of the following methods use hardware timing?
   a. Timed Loop linked to a µs clock
   b. DAQmx VIs connected to an external clock
   c. Wait Express VI with µs resolution
   d. Timed Loop linked to a ms clock

Summary – Quiz Answers

1. Which of the following are benefits of using timing in a control loop?
   a. Provide sleep so lower priority threads can execute
   b. Reduce application jitter
   c. Both a & b

2. True or False? It is good programming practice to use wait functions in parallel with time critical code.
   False

3. Which of the following typically provides finer resolution?
   a. Hardware timing
   b. Software timing

4. Which of the following methods use hardware timing?
   a. Timed Loop linked to a µs clock
   b. DAQmx VIs connected to an external clock
   c. Wait Express VI with µs resolution
   d. Timed Loop linked to a ms clock

Notes

5

Communication

The RT Engine on the RT target does not provide a user interface for applications. You can use one of two communication protocols, front panel communication or network communication, to provide a user interface on the host computer for RT target VIs.

After separating deterministic tasks from non-deterministic tasks in an application, you must use deterministic communication methods to share data. You can use deterministic communication methods to share data between locations in a VI that cannot be connected with wires, between VIs running on an RT target, and between VIs across a network running on different targets.

In this lesson, you will develop the normal priority portion of the target application, along with the host application.

Topics

A. Front Panel Communication

B. Network Communication

C. Network Communication Programming

A. Front Panel Communication

With front panel communication, the host computer and the RT target execute different parts of the same VI. On the host computer, LabVIEW displays the front panel of the VI while the RT target executes the block diagram. A user interface thread handles the communication between LabVIEW and the RT Engine.

Use front panel communication between LabVIEW on the host computer and the RT Engine to control and test VIs running on an RT target. After downloading and running the VIs, keep LabVIEW on the host computer open to display and interact with the front panel of the VI.

You also can use front panel communication to debug VIs while they run on the RT target. You can use LabVIEW debugging tools such as probes, execution highlighting, breakpoints, and single stepping to locate errors on the block diagram. Refer to Lesson 6, *Verifying Your Application*, for information about debugging applications.

Front panel communication is a good communication method to use during development, because you can quickly monitor and interface with VIs running on an RT target. Front panel communication causes sections of code that contain front panel controls and indicators to be non-deterministic. This is because LabVIEW must switch to the user interface thread, which is non-deterministic, to complete the task. Therefore, if you are using front panel communication, you should not place front panel controls and indicators in deterministic sections of code. If no front panel terminals are in the deterministic section of code, the deterministic section of code will run deterministically.

B. Network Communication

With network communication, a host VI runs on the host computer and communicates with the VI running on the RT target using specific network communication methods such as network-published shared variables, Network Stream functions, or other protocols. You might use network communication for the following reasons:

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

C. Network Communication Programming

Network-published shared variables, network streams, and other protocols are the most common methods for sharing data in deterministic applications. This section describes programming tips for communication using these methods.

Table 5-1. Network Communication Programming

Use Case	Examples	Protocol
Latest value	Display most recent I/O values of RT target on host computer	Network-published Shared Variables
Buffered values	Transfer data to host computer for file logging	Network Streams
Other protocols	Transfer data to LabVIEW and non-LabVIEW applications	TCP, UDP, serial, etc.

Network-Published Shared Variables

You can use network-published shared variables to share the latest value in a data set between VIs running on different targets across a network.

There are two ways to use network-published shared variables to transfer data from a target to the host—network-published shared variables with or without the RT FIFO option enabled.

Network-published shared variables with the RT FIFO option enabled automatically generate an invisible communication loop. A variable configured in this manner transfers data from the deterministic loop to the communication loop using an RT FIFO and transfers data from the communication loop to the host over the network. If the only function of a non-deterministic loop in your application is to transfer data to the host, using network-published shared variables with the RT FIFO enabled removes the need to program that loop and saves you effort.

Network-published shared variables without the RT FIFO option enabled allow you to transfer data from a non-deterministic loop to the host over the network. If your application requires a non-deterministic loop for reasons other than communication, such as data logging, you can transfer the data to the non-deterministic loop using a single-process shared variable RT FIFO and then transfer data from the non-deterministic loop to the host using a network-published shared variable without the RT FIFO option enabled.

By enabling the real-time FIFO of a shared variable, you can share data across a network without affecting the determinism of the VIs. However, the

transfer of the data across the network is not deterministic. Due to network latency, the most recently written data may not be available to a VI running on a machine across the network. In this case, the VI attempting to read from the network-published shared variable returns the previous value.

Initialize Your Network-Published Shared Variables

To prevent reading stale data from a previous run of your application, undeploy and redeploy shared variable libraries to clear all shared variable values. Use the Shared Variable Deployment page of the Application Properties dialog box to configure a stand-alone RT application to deploy shared variables before running the application and undeploy shared variable libraries when the application exits. Refer to Lesson 7, *Deploying Your Application*, for more information on configuring and building stand-alone real-time RT applications.

You also can safeguard your application against stale shared variable data by initializing all your shared variables before running the ongoing task loops of your application. To initialize an unbuffered network-published shared variable hosted on another computer, write the default value to the shared variable, then wait for the initialized value to propagate through the network to the Shared Variable Engine running on the host computer and back to the RT target. Figure 5-1 shows an example of unbuffered network-published shared variable initialization.

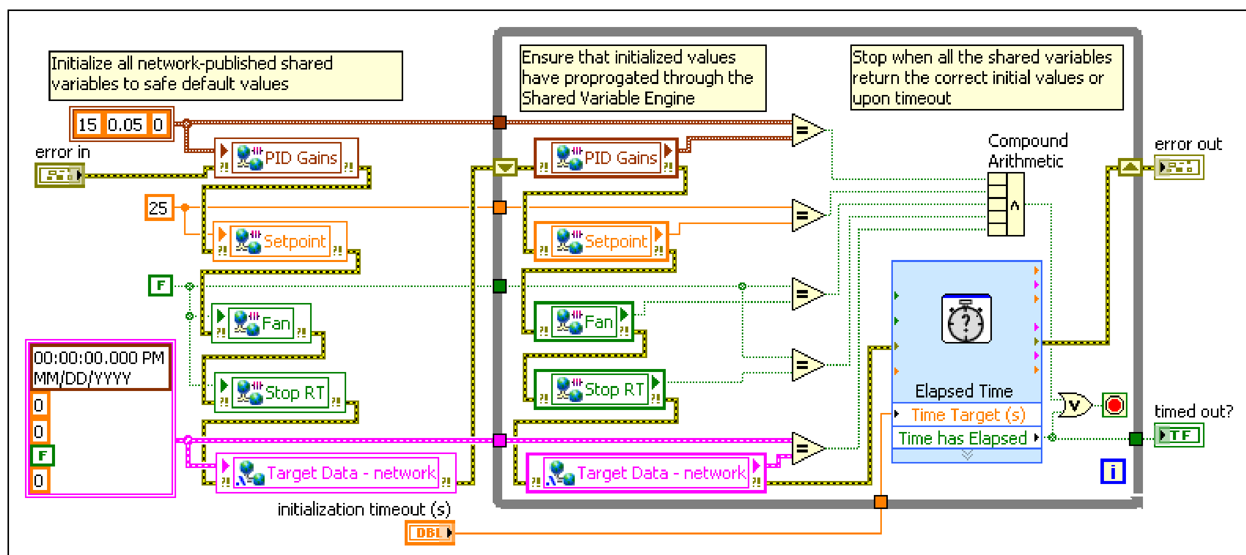


Figure 5-1. Initializing Network-Published Shared Variables

Location for Network-Published Shared Variables

You can choose to host network-published shared variables on the host or the target.

In some cases, hosting shared variables on the RT target makes sense. In other cases, it is more appropriate to host shared variables on a host PC. Before finalizing your application, ensure that you are hosting shared variables on the most appropriate device. The following table summarizes the advantages and disadvantages of hosting shared variables on an RT target.

Table 5-2. Advantages and Disadvantages of Hosting Network-Published Shared Variables on an RT Target

Target Advantages	Disadvantages
Multiple PCs can access shared variables hosted by a single RT target	Adds CPU overhead on the RT target
High uptime due to stability of the RT target	Adds memory overhead on the RT target

Scope of Network-Published Shared Variables

When a VI that references a network-published shared variable runs, all variables in the library containing the variable are deployed and published. You should consider how to group variables into libraries to avoid deploying unnecessary variables. Network-published shared variables are never automatically undeployed, even after a system reboot. They remain on the system, ready to provide or receive data.

Undeploying Shared Variables

There are many ways to remove shared variables from a system if they are no longer in use.

- Programmatically undeploy variables using the VIs in the LabVIEW DSC module.
- Programmatically undeploy variables by using the **Library»Undeploy Library** method of the Application VI Server class.
- Manually undeploy a library by right-clicking the library in the Project Explorer and selecting **Undeploy**.
- Manually undeploy a library by using the Distributed System Manager. Start the Distributed System Manager by selecting **Tools»Distributed System Manager**.

Network Stream Functions

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

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

Stream data continuously between two LabVIEW applications with network streams. A network stream is a lossless, unidirectional, one-to-one communication channel that consists of a writer and a reader endpoint. Use the Network Streams functions to stream data with network streams. Use the Network Stream Endpoint properties to view information about endpoints.

You can use network streams to stream any data type between two applications, but the following data types stream at the fastest rates:

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

Network Streams Engine

Each endpoint uses a FIFO buffer to transfer data. The Network Streams Engine (NSE) uses LogosXT to transfer data from the FIFO buffer on the writer endpoint to the FIFO buffer on the reader endpoint.

Figure 5-2 illustrates the flow of data in a network stream.

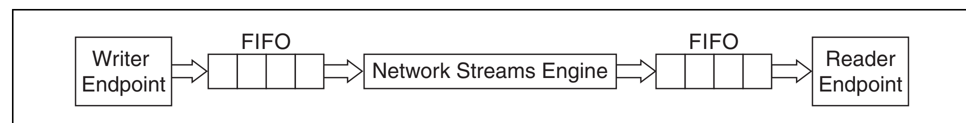


Figure 5-2. Flow Data in a Network Stream

In Figure 5-2, data flows in the following order.

1. The writer endpoint writes data to a FIFO buffer.
2. The NSE transfers data over a network to another FIFO buffer on the reader endpoint.
3. The reader endpoint reads the data from that FIFO buffer.

Determining When to Use Network Streams Instead of Shared Variables

Use shared variables to publish the latest value in a data set to many computers. Conversely, use network streams to log every point of data on one computer.

For example, assume that you are using an accelerometer to detect the vibrations of a pump that is re-pressuring natural gas in a pipeline. You are processing the vibration data on a CompactRIO target to monitor for bearing fault to ensure that the pump does not fail. However, the CompactRIO target does not have enough memory to analyze the data. Therefore, you must send the data to a desktop computer that has enough memory to store, analyze, and display that data.

Because shared variables are optimized for publishing the latest value of data only, they could miss a critical data point. However, network streams would stream every point of data to the desktop computer so you could monitor the condition of the engine.

Note Network streams can induce jitter in deterministic loops. Therefore, if you want to stream data from a deterministic loop with network streams, National Instruments recommends that you first share the data with a non-deterministic loop. Then, use network streams to stream the data to another device.

Network Stream Endpoints

Endpoints are the parts of applications that exchange data. Every network stream must have a writer endpoint and a reader endpoint. Writer endpoints write data to the stream. Reader endpoints read data from the stream. LabVIEW identifies each endpoint with an endpoint URL, which you use to establish connections between endpoints. When two endpoints connect, they create a network stream. You use the Network Streams functions to create endpoints and stream data between them.

When you specify a name for an endpoint with the writer name terminal of the Create Network Stream Writer Endpoint function or the reader name terminal of the Create Network Stream Reader Endpoint function, LabVIEW uses that name to create a URL.

To create a valid network stream, use endpoint URLs to prompt a writer and a reader endpoint to connect to each other. Perform this task by wiring the URL of a remote endpoint to the reader url input on the Create Network Stream Writer Endpoint function or the writer url input on the Create Network Stream Reader Endpoint function.

The URL you must specify in these terminals varies depending on the network location of the remote endpoint.

Note Endpoint URLs are not case sensitive. However, when you specify an endpoint URL, you must replace any reserved characters that you use with the corresponding escape codes to prevent parsing errors.

Organizing Network Stream Endpoint Names

You can organize endpoints by adding segments to their names that describe the data they stream, the computer on which they reside, or other characteristics. When you have multiple endpoints on different computers, organizing endpoints in this way helps ensure that you connect each writer endpoint to its corresponding reader endpoint.

Complete the following steps to organize endpoints by name.

1. Wire the writer name terminal of the Create Network Stream Writer Endpoint function or the reader name terminal of the Create Network Stream Reader Endpoint function.
2. Place a slash between each segment of the name that you specify in these terminals.

For example, assume you have three writer endpoints within the same application on a real-time (RT) target. One measures temperature, and the other two measure voltage signals. These endpoints connect to three corresponding reader endpoints in an application on a desktop computer. Figure 5-3 shows examples of names that you could assign to each endpoint. The arrows indicate the reader endpoint to which each writer endpoint connects.

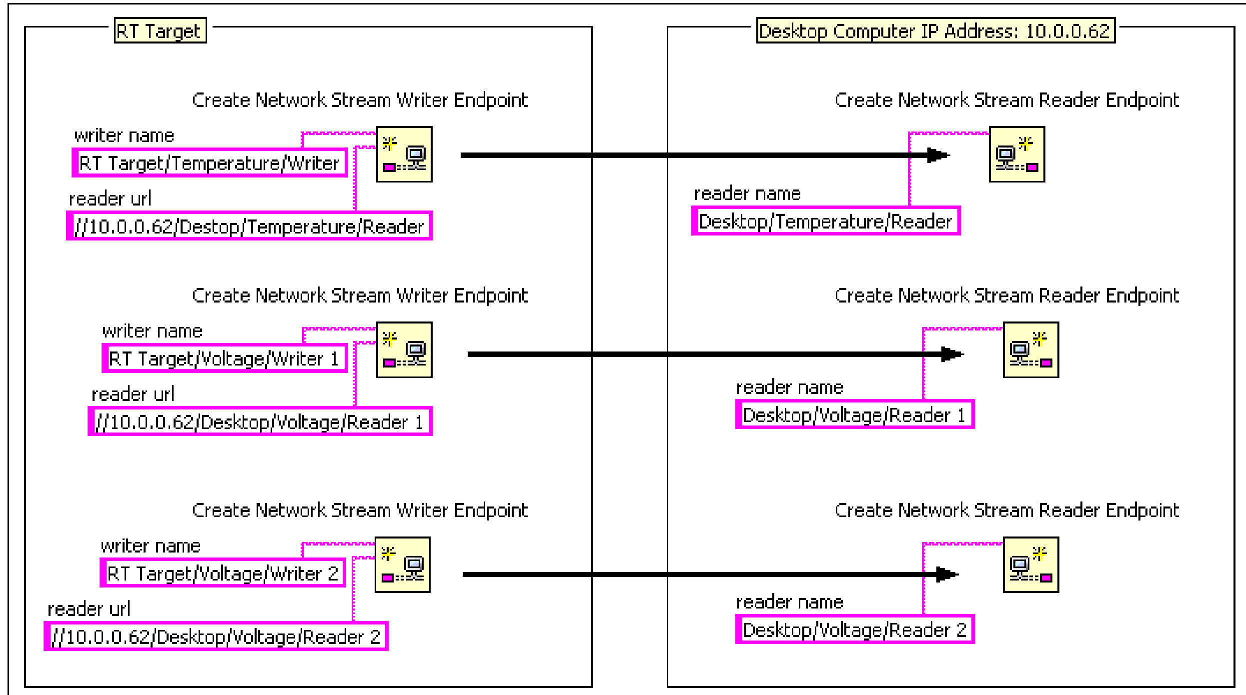


Figure 5-3. Organizing Network Stream Endpoint Names

In Figure 5-3, the name of each endpoint describes the target the endpoint resides on, the type of data the endpoint streams, and whether the endpoint reads or writes that data. This naming scheme shows which endpoints correspond with each other.

Note The name of one endpoint cannot be the partial name of another endpoint within the same application. For instance, in the above example, if you name one of the writer endpoints `RT Target/Voltage` and the other `RT Target/Voltage/Writer 1`, these endpoints will return an error instead of creating network streams.

Other Protocols

You can use other protocols to communicate with LabVIEW and non-LabVIEW applications. For example, you may need to transfer data to a third party device that uses the Transmission Communication Protocol (TCP) for communication. You can use the TCP functions in LabVIEW for TCP communication. Other protocols include User Datagram Protocol (UDP), serial, and more.

TCP Communication

TCP is an industry-standard protocol for communicating over networks. VIs running on the host computer can communicate with RT target VIs using the TCP VIs and functions. However, TCP is non-deterministic, and using TCP communication inside a deterministic loop can affect the determinism of the deterministic loop.

You can use the TCP functions in LabVIEW for TCP communication.

UDP Communication

UDP is a network transmission protocol for transferring data between two locations on a network. UDP is not a connection-based protocol, so the transmitting and receiving computers do not establish a network connection. Because there is no network connection, there is little overhead when transmitting data. However, UDP is non-deterministic, and using UDP communication inside a deterministic loop can affect the determinism of the deterministic loop.

When using the UDP VI and functions to send data, the receiving computer must have a read port open before the transmitting computer sends the data. Use the UDP Open function to open a write port and specify the IP address and port of the receiving computer. The data transfer occurs in byte streams of varying lengths called *datagrams*. Datagrams arrive at the listening port and the receiving computer buffers and then reads the data.

You can transfer data bidirectionally with UDP. With bidirectional data transfers, both computers specify a read and write port and transmit data back and forth using the specified ports. You can use bidirectional UDP data transfers to send and receive data from the network communication VI on the RT target.

UDP has the ability to perform fast data transmissions deterministically. However, UDP cannot guarantee that all datagrams arrive at the receiving computer. Because UDP is not connection based, you cannot verify the arrival of datagrams. You must ensure that network congestion does not affect the transmission of datagrams. Also, you must read data stored in the data buffer of the receiving computer fast enough to prevent overflow and loss of data.

You can use the UDP functions in LabVIEW for UDP communication.

Table 5-3. Network Communication Methods Comparison

Protocol	Speed	Deterministic Read/Write	Deterministic Data Transfer	Advantages	Caveats	Common Use
Network-published Shared Variable	Fast	With RT FIFO enabled	No	Ease of programming	LabVIEW Only	Latest value, host interface
Network Streams	Faster	No	No	Built-in functions	LabVIEW Only	Data streaming
TCP	Fastest	No	No	High transfer rates	String data	Data streaming
UDP	Fastest	No	No	High transfer rates	String data, lossy	Broadcast latest values

Summary – Quiz

1. Match the following terms with their definitions:

TCP	A. Fast, lossy communication protocol with minimal error checking
UDP	B. Commonly used protocol, fast and lossless
Network-published shared variables	C. Can transfer data directly from a time-critical loop

2. True or False? When using front panel communication, a time-critical loop that contains front panel controls and indicators is deterministic.

3. True or False? You should never use TCP and UDP functions inside time-critical code.

4. Which of the following should you use to communicate from the time-critical loop directly to the host VI?
   a. Single-process shared variable with RT FIFO disabled
   b. Single-process shared variable with RT FIFO enabled
   c. Network-published shared variable with RT FIFO disabled
   d. Network-published shared variable with RT FIFO enabled

Summary – Quiz Answers

1. Match the following terms with their definitions:

TCP	**B. Commonly used protocol, fast and lossless**
UDP	**A. Fast, lossy communication protocol with minimal error checking**
Network-published shared variables	**C. Can transfer data directly from a time-critical loop**

2. True or False? When using front panel communication, a time-critical loop that contains front panel controls and indicators is deterministic.

   False

3. True or False? You should never use TCP and UDP functions inside time-critical code.

   True

4. Which of the following should you use to communicate from the time-critical loop directly to the host VI?

   a. Single-process shared variable with RT FIFO disabled

   b. Single-process shared variable with RT FIFO enabled

   c. Network-published shared variable with RT FIFO disabled

   d. Network-published shared variable with RT FIFO enabled

Notes

6 Verifying Your Application

In this lesson, you learn how to debug an RT application and how to monitor the performance and memory usage of an RT target.

Topics

A. Verifying Correct Application Behavior
B. Verifying Performance and Memory Usage

A. Verifying Correct Application Behavior

When verifying your application, you must ensure that the application behaves as expected. When you discover problems with your code, you can use the LabVIEW debugging tools, such as execution highlighting and single-stepping, while the host computer is connected to an RT target to step through LabVIEW code to find the source of the unexpected behavior.

Note You must place a checkmark in the **Allow debugging** checkbox of the Execution page of the VI Properties dialog box to use the LabVIEW debugging tools to debug a VI.

The only feature not supported by the LabVIEW Real-Time Module is the Call Chain ring, which appears in the toolbar of a subVI block diagram window while single-stepping.

Note Do not use the LabVIEW debugging tools to debug execution timing, because all debugging tools affect the timing of an application.

The following pages review standard LabVIEW debugging techniques. Refer to the *Debugging Techniques* topic of the *LabVIEW Help* for more information about debugging in LabVIEW.

Standard Debugging Techniques

When your VI is not executable, the **Run** button on the toolbar appears with a broken arrow. To list errors and warnings, click the **Run** button to open the Error list window. Double-click the error message to locate the error on the block diagram.

Use error handling to debug and manage errors in VIs. The LabVIEW error handler VIs return error codes when an error occurs in a VI. Error codes reveal the specific problem the VI encountered. When you configure an RT target, LabVIEW automatically copies the error code files used by the error handler VIs to the target.

You can use custom error codes with VIs that run on an RT target. Create error files using the Error Code File Editor by selecting **Tools»Advanced» Edit Error Codes**. If you use custom errors with LabVIEW, you must rename the files to use a `.err` extension and then place the error files in the `<ni-rt>\system\user.lib\errors` directory or the `<ni-rt>\system\errors` directory on the RT target. Use the FTP client in MAX or any other FTP client to transfer the error file to the networked device. Refer to the *Defining Custom Error Codes* topic *LabVIEW Help* for information about defining custom error codes.

Execution highlighting animates the block diagram and traces the flow of the data, allowing you to view intermediate values. To implement execution highlighting, click **Highlight Execution** on the toolbar.

Use a probe to view values passing through a wire segment. Click a wire with the Probe tool or right-click the wire to set a probe.

A breakpoint sets a pause at a location on the diagram. Click wires or objects with the Breakpoint tool or right-click a wire to set breakpoints.

Use a conditional probe to set conditions for when to pause at the probe. For example, you may want a breakpoint to occur only when the value in the wire drops below zero. A conditional probe can accomplish this task.

Use single stepping to execute the diagram node by node. You can access single stepping from the single step buttons on the block diagram toolbar.

Click **Step Into** or **Step Over** to begin single stepping.

- **Step Into**—Steps into a node. If the node contains a subVI, LabVIEW opens the subVI and enables single stepping through the subVI.
- **Step Over**—Executes the next node, but visually does not single step through the nodes.
- **Step Out**—Steps out of a node, if the block diagram has completed execution; click **Step Out** to terminate single stepping mode.

B. Verifying Performance and Memory Usage

You can use one of the following methods to verify the performance and memory usage of an application:

- Profile Performance and Memory tool
- Distributed System Manager
- RT Utility VIs
- Real-Time Execution Trace Toolkit

Profile Performance and Memory Tool

The Profile Performance and Memory window is a powerful tool for statistically analyzing how an application uses execution time and memory. You can use the Profile Performance and Memory window to display information for all VIs and subVIs in memory. This information can help you optimize the performance of your VIs by identifying potential bottlenecks. For example, if you notice that a particular subVI takes a long time to execute, you can improve the performance of that VI. The Profile Performance and Memory window displays the performance information for all VIs in memory in an interactive tabular format. From the Profile

Performance and Memory window, you can select the type of information to gather and sort the information by category. You also can monitor subVI performance within different VIs. Select **Tools»Profile»Performance and Memory** to display the Profile Performance and Memory window.

You must place a checkmark in the **Profile Memory Usage** checkbox before starting a profiling session. Collecting information about VI memory use adds a significant amount of overhead to VI execution, which affects the accuracy of any timing statistics gathered during the profiling session. Therefore, perform memory profiling separate from time profiling to return an accurate profile.

Many of the options in the Profile window become available only after you begin a profiling session. During a profiling session, you can take a snapshot of the available data and save it to an ASCII spreadsheet file. The timing measurements accumulate each time you run a VI.

You can verify memory usage with the Profile Performance and Memory window. However, you also can verify and determine memory usage with the Distributed System Manager and RT Utility VIs.

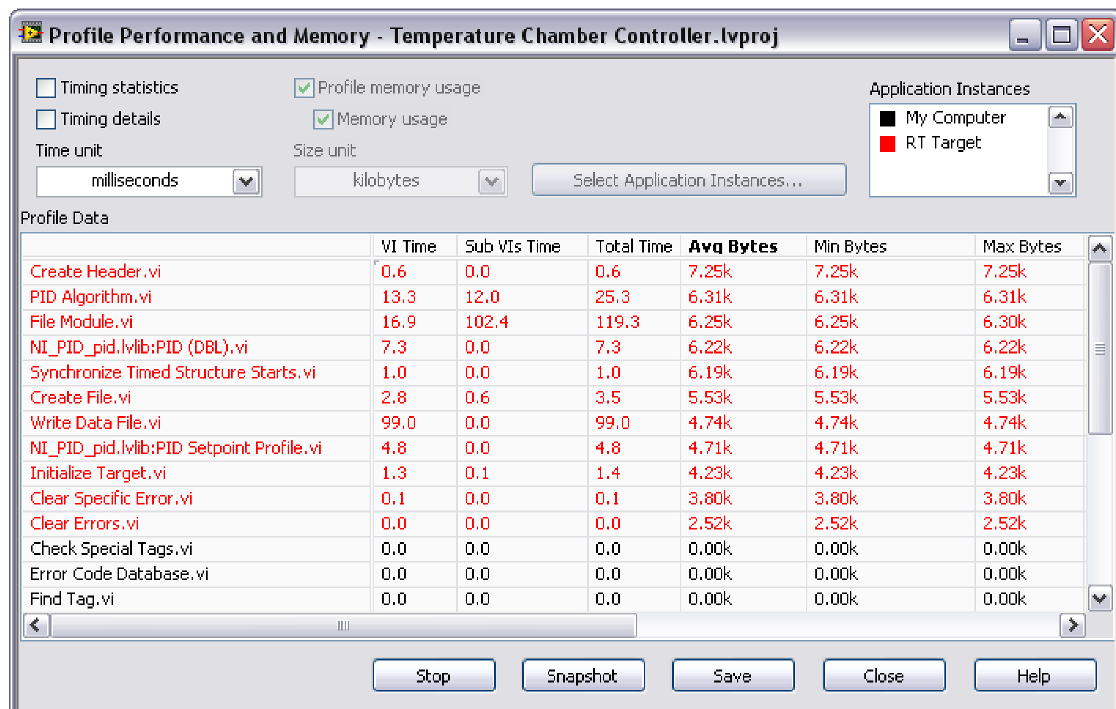


Figure 6-1. Profile Performance and Memory Window

Distributed System Manager

The Distributed System Manager displays RT target resources in addition to shared variable and I/O variable data. The Distributed System Manager displays details about VIs running on an RT target and provides a dynamic display of the memory and CPU resources of the target. You can stop VIs and start idle VIs on the RT target using the Distributed System Manager. Select **Tools»Distributed System Manager** to open the Real-Time System Manager.

The Distributed System Manager is installed by default when you install LabVIEW Real-Time. The Distributed System Manager interface runs on the host and displays the RT system usage. If the target has no available CPU resources to run report RT system usage, then the target does not report back any information.

Memory and CPU Usage

In the Distributed System Manager, click the CPU/Memory tab of an RT target to monitor CPU and memory usage on the target.

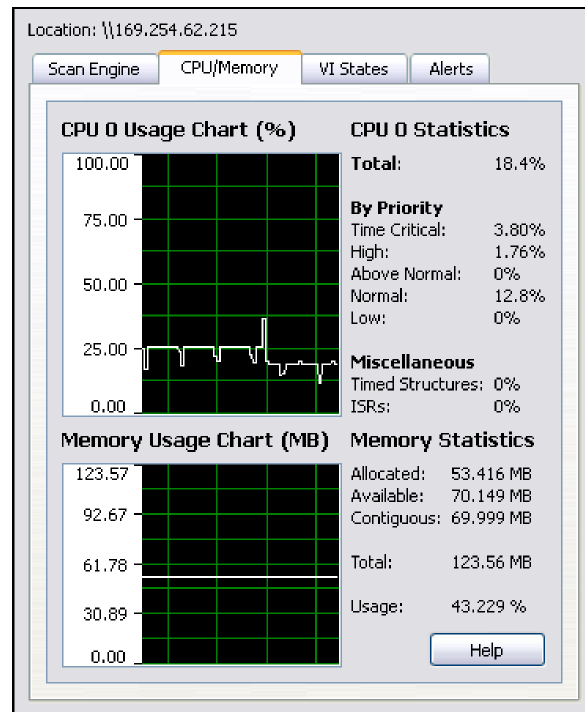


Figure 6-2. Distributed System Manager CPU/Memory Tab

The Distributed System Manager displays a CPU chart for each CPU in the system. Each graph tracks the utilization of the CPU as a percentage of capacity, over time. The Distributed System Manager also displays the percentage of total CPU utilization devoted to each priority level, as well as

the percentage devoted to interrupt service routines (ISRs) and Timed Structures. The Distributed System Manager also displays a Memory chart that tracks memory usage, in megabytes, over time.

You can monitor the memory usage to determine if memory leaks occur over time. You can monitor the CPU usage statistics to determine if higher priority loops are yielding enough time for lower priority loops to execute.

This view includes the following components:

- CPU N Usage Chart (%)—Tracks utilization of CPU N over time, as a percentage of capacity.
- Memory Usage Chart (MB)—Tracks target memory usage over time.
- CPU N Statistics—Includes current CPU usage statistics including total CPU usage, CPU usage by priority level, and CPU usage dedicated to Timed Structures and interrupt service routines (ISRs). Each value represents a percentage of total capacity.
- Memory Statistics—Displays current memory usage data.
  - Allocated—The amount of memory, in MB, currently allocated on the target.
  - Available—The amount of memory, in MB, currently available on the target.
  - Contiguous—The largest contiguous block of available memory on the target, in MB.
  - Total—The total amount of memory available to the operating system, in MB.
  - Usage—The percentage of total memory currently allocated on the target.

VI States

In the Distributed System Manager, click the VI States tab of an RT target to start, stop, and monitor VIs on the RT target.

This view includes the following components:

- Port—Specifies the TCP/IP port at which the VI server listens for requests on the RT target. Right-click the RT target in the Project Explorer window and select Properties from the shortcut menu to display the RT Target Properties dialog box. Select VI Server from the Category list to display the VI Server page, where you can configure the TCP/IP port for the VI Server.
- Update Interval—Sets the minimum time that the server waits between updates. The actual update interval might exceed the time you specify if higher priority tasks are running on the RT target.

- Show Top Level VIs Only—Filters all subVIs from the list of RT target VIs.
- Start Monitoring—Starts monitoring VIs on the RT target.
- Stop Monitoring—Stops monitoring VIs on the RT target.
- Start VI—Starts an idle VI that you select from the list of RT target VIs.
- Stop VI—Stops a VI that you select from the list of RT target VIs.

The VI States tab uses VI Server to communicate to the RT target. You must configure the RT target to allow VI Server access on a TCP/IP port. Configure VI Server access on the RT target opening the Real-Time Target Properties window from the Project Explorer. Then you must configure the Distributed System Manager to connect to the corresponding port on the RT target.

Alerts

Use the Alerts tab of the Distributed System Manager to log and save system information for viewing later. This can be very important for monitoring system history. If an RT system has problems, you can return to the log to see what the system properties were at the time the problem occurred.

This view includes the following components:

- Log Alert when memory usage is above—Enables alert logging when memory usage on the RT target exceeds the value you specify.
- Log Alert when CPU usage is above—Enables alert logging when the RT target CPU usage exceeds the value you specify.
- Log Alert when VI changes state—Enables alert logging when a VI listed in the VI States view changes state.
- Recent Alerts—Lists recently-logged alerts.
- Clear—Removes all entries from the Recent Alerts list.
- Save—Saves the Recent Alerts list as a text file on the local computer.

RT Utility VIs

Use the following RT Utility VIs in the RT target VI to programmatically monitor CPU load and memory usage data on the RT target.

- **RT Get CPU Loads VI**—Monitors the distribution of load on the CPUs in the system. For each CPU in the system, this VI returns the total load as a percentage of capacity. This VI also returns the percentage of CPU time devoted to each priority level, the percentage of idle CPU time, and the percentage of CPU time devoted to Timed Structures and interrupt service routines (ISRs).
- **RT Get Memory Usage VI**—Monitors memory usage on the RT target. This VI returns the following memory usage data:
  - Total (bytes)—amount of memory installed on the target, in bytes
  - Available (bytes)—amount of memory available on the target, in bytes.
  - Largest contiguous (bytes)—size of the largest contiguous block of available memory on the target, in bytes.

Real-Time Execution Trace Toolkit

The Real-Time Execution Trace Toolkit is a set of real-time event and execution tracing tools that allows you to capture and display the timing and execution data of VI and thread events for applications running on an RT target.

With minimal modifications to your embedded code, these tools graphically display multithreaded code execution while highlighting thread swaps, mutexes, and memory allocation. Using this information, you can optimize the real-time code for faster control loops and more deterministic performance.

This tool and topic is covered in the *LabVIEW Real-Time 2* course.

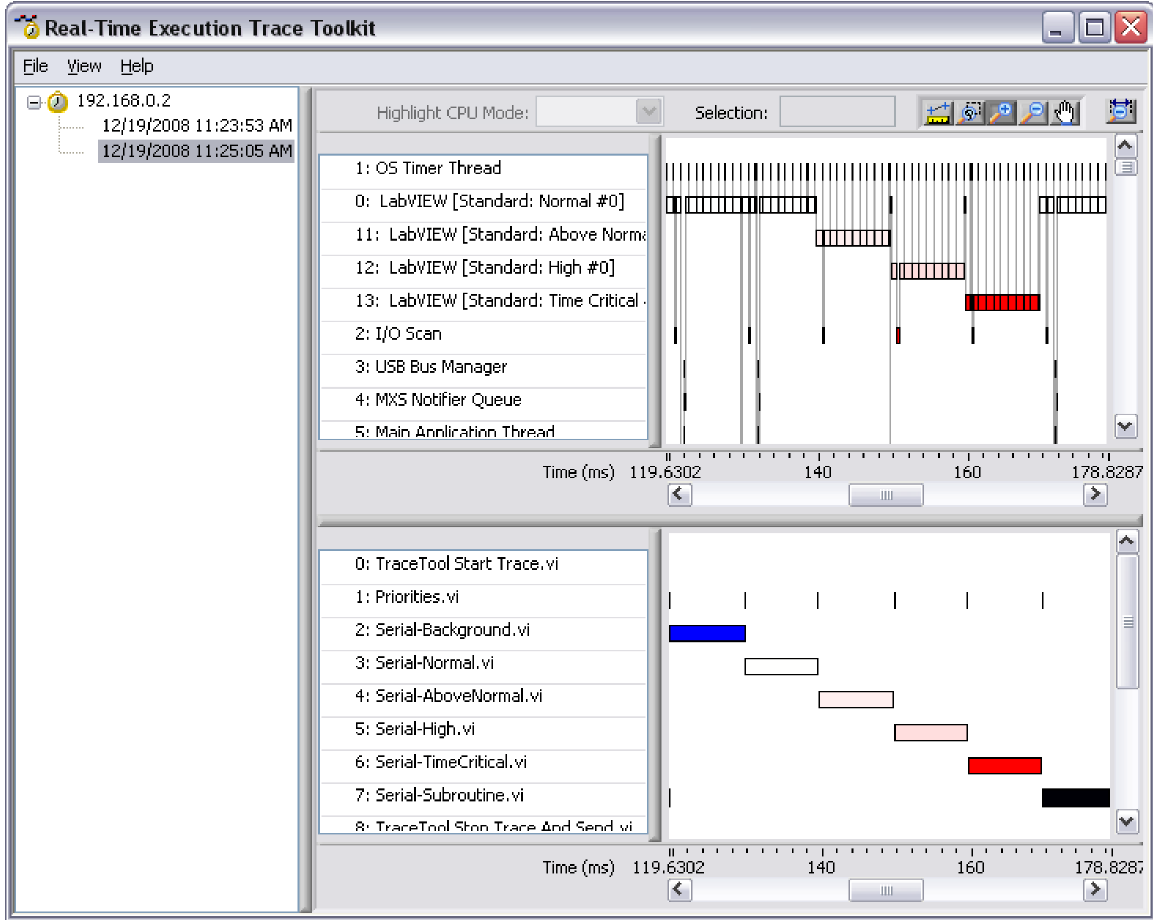


Figure 6-3. Real-Time Execution Trace Toolkit

Summary – Quiz

1. True or False? You must enable the **Allow debugging** option in VI Properties in order to use the **Execution Highlighting** and single-stepping tools on a deterministic VI.

2. Match the tool with what it can test:

Distributed System Manager	A. CPU Usage
Performance and Memory Profiler	B. Memory Usage
Probes	C. Timing Behavior
Execution Highlighting	D. Application Behavior

Summary – Quiz Answers

1. True or False? You must enable the **Allow debugging** option in VI Properties in order to use the **Execution Highlighting** and single-stepping tools on a deterministic VI.

   True

2. Match the tool with what it can test:

Distributed System Manager **(A, B)**	A. CPU Usage
Performance and Memory Profiler **(B, C)**	B. Memory Usage
Probes **(D)**	C. Timing Behavior
Execution Highlighting **(D)**	D. Application Behavior

Notes

7

Deploying Your Application

After developing a LabVIEW Real-Time Module application, you often want to deploy the application so that it becomes the primary application for the controller. In this lesson, you will learn how to create an executable from an application, embed the executable on the target, launch the executable, and communicate with the executable.

Topics

A. Introduction to Deployment

B. Creating a Build Specification

C. Communicating with Deployed Applications

D. System Replication

A. Introduction to Deployment

When you complete the development work on a project, you may want to deploy the project. Use the LabVIEW Application Builder, included with the LabVIEW Professional Development System, to create stand-alone LabVIEW Real-Time Module applications. You can embed a stand-alone application on an RT target and launch the application automatically when you boot the target.

Preparing Your Application for Deployment

Before deploying your application, you must prepare the application for deployment. This involves reviewing the code for any unsupported functions.

Some of the LabVIEW features that are unavailable when you target a specific RT target include functions that modify front panel objects and functions specific to other operating systems, such as ActiveX.

Note If you attempt to download and run a VI that has unsupported functionality on an RT target, the VI still executes. However, the unsupported functions do not work and return standard LabVIEW error codes.

Avoid Modifying Front Panel Objects

When a VI or stand-alone application runs on an RT target and there is no front panel connection with LabVIEW on the host computer, you cannot execute VIs that modify a front panel. For example, you cannot change or read the properties of front panel objects with Property Nodes, because no front panel exists. You must establish a front panel connection with the RT target or open a remote front panel connection to read any front panel properties or for any front panel Property Node changes to reflect on the front panel objects. Refer to Lesson 5, *Communication*, for more information about front panel connections.

The following features do not work on an RT target with no front panel connection:

- Front panel Property Nodes and control references
- Dialog functions
- VI Server front panel functions

Avoid OS-Specific Technologies

VIs on the RT target cannot run VIs that use Windows-only technology. The following features do not work on an RT target:

- ActiveX VIs
- NET VIs
- VIs that use NI-IVI drivers
- Windows Registry Access VIs
- TestStand VIs (ActiveX-based)
- Report Generation Toolkit VIs
- Call Library Nodes that access an operating system API other than RTOS
- Graphics and Sound VIs
- Database Connectivity Toolset
- XML DOM Parser and G Web Server for CGI Support

Refer to the *Unsupported LabVIEW Features (RT Module on VxWorks Targets)* and *Unsupported LabVIEW Features (RT Module on ETS Targets)* topics in the *LabVIEW Help* for more information about technologies to avoid on RT targets.

Running a Stand-Alone Real-Time Application (RT Module)

You can create a stand-alone real-time application using the LabVIEW Application Builder and set the application to run when you power on an RT target. You can create multiple build specifications that configure the settings of stand-alone real-time applications under an RT target. However, you can set only one stand-alone real-time application as startup for an RT target. When you set a build specification as the startup application for an RT target, LabVIEW displays a green border around the icon for the build specification in the Project Explorer window.

To run a built stand-alone real-time application, right-click the build specification in the Project Explorer window and select **Run as startup** from the shortcut menu. The **Run as startup** shortcut menu item sets the application as the startup application, deploys the application to the target, and prompts you to reboot the RT target.

You also can complete the following steps to run a stand-alone real-time application as the startup application on an RT target:

1. Build the stand-alone real-time application.
2. Right-click the build-specification and select **Set as startup** from the shortcut menu to enable the application to run when you reboot or power on the RT target.
3. Right-click the build specification and select **Deploy** from the shortcut menu to deploy the application to the RT target.
4. Right-click the RT target and select **Utilities»Reboot** to reboot the RT target and run the stand-alone real-time application.

Note You cannot run a stand-alone real-time application without setting the application as startup and rebooting the RT target.

B. Creating a Build Specification

An RT build specification saves stand-alone applications on the host computer or embeds applications on an RT target. You create a build specification through the RT target in the Project Explorer. You can launch a built application from the Project Explorer or outside of LabVIEW. Refer to the *Automatic Start on Target* section of this lesson for information about automatically launching stand-alone applications.

Complete the following steps to build and deploy a stand-alone real-time application.

1. Create a build specification under the RT target in the Project Explorer.
2. Save the project to save the build specification settings.
3. (Optional) Right-click the build specification and select the **Set as Startup** option to configure the target to run the application automatically when the target reboots.
4. Right-click the build specification and select **Deploy** to build the application and download it to the RT target.
5. Reboot the target.

Configuring Settings – Information

The **Information** category contains the **Build specification name**, **Executable filename**, and **Target destination directory** for both the target and host.

Configuring Settings – Source Files

Use the **Source Files** category to set startup VIs and include additional VIs or support files. You do not need to specifically include VIs called as subVIs from the Startup VIs unless the subVIs are called dynamically.

Use the **Destinations** and **Source File Settings** categories to control where files are created and to set the visual properties of each VI. These categories are rarely used when building RT applications.

Configuring Settings – Advanced

Use the **Advanced** category to enable debugging in the executable, copy error code files, and use an alias file. Enable debugging in an executable to connect to the executable and debug it as it runs. However, debugging requires additional resources on the target and slows the executable considerably.

Configuring Settings – Preview

The Preview category shows you the destination of the files to be created when you deploy the build.

Automatic Start on Target

Selecting the **Set as Startup** option configures the target to run the application automatically when you power on or reboot the RT target. Use this option to create headless systems that start automatically without any interaction from the host or the development environment. You can select **Unset as Startup** and then select **Deploy** to disable automatic startup. Connecting to the target from any project other than the one containing the startup VI also disables automatic startup.

Right-click the build specification and select **Set as startup** to set the real-time application to begin execution when you start or reboot the RT target.

Right-click a build specification and select **Deploy** from the shortcut menu to deploy the stand-alone real-time application to the RT target. You must redeploy the real-time application when you rebuild the application or change the properties of the application for the changes to take effect on the RT target.

Note You also can set a VI as the startup VI for an RT target without creating a stand-alone real-time application if you do not have access to the Application Builder. Refer to `ni.com/info` and enter the Info Code `rdcsvi` for information about setting VIs as startup VIs for RT targets.

Deploying the Build Specification

Select **Build** to build the application and **Deploy** to download the executable to the target. In most cases, selecting **Deploy** also builds the application.

Removing Executables

At some point, you may want to remove an executable you stored on your RT target. The easiest way to do this is to use FTP to access the module and delete the file. You can use any FTP client software including Internet Explorer. After you access the target, you can see all of the files stored on the target.

Compact FieldPoint—At this point, if you try to remove the file, you get a message that you do not have permission. You must set Compact FieldPoint to safe mode before you can remove files from the flash. To put the module in safe mode, flip the DIP switch labeled `safe mode` and cycle power to the module. You must cycle power to the module because this switch, along with the `Disable VI` and `Reset` switches, is read only at power-up. If you have the VI set to launch on boot up, you also must flip the `Disable VI` switch. After the module comes back online, you can delete the `startup.exe` file. The file is located in `NI-RT\Startup`.

C. Communicating with Deployed Applications

You can create VIs on a host computer, as described in Lesson 5, *Communication*, to communicate with a stand-alone application running on an RT target. You also can debug a deployed executable using standard debugging functions.

Debugging Executables

You can debug deployed executables as long as you enable the debugging option when you build the executable. Notice that debuggable applications use more memory and processor resources than non-debuggable applications. To debug a stand-alone executable, perform the following steps:

- Enable debugging on the Advanced page of the Application Builder when you build the executable.
- Select **Operate»Debug Application or Shared Library**.
- Enter the IP address of the target and click **Refresh**.
- Select the application to debug and click **Connect**.
- Debug the application normally.

D. System Replication

You can use RT target disk images to backup, restore, and replicate RT targets. An RT target disk image is a copy of the file contents of the primary RT target hard drive.

There are two different methods you can use to create and apply RT target disk images. For networked RT targets, you can use the programmatic method, which involves using the Real-Time Utilities VIs on host computer. For USB-enabled RT targets, you can use the USB method, which involves booting from an RT Desktop PC Utility USB drive and using the **National Instruments Real-Time Desktop PC Utility Collection** menu.

Note You cannot use the programmatic method to load an image created with the USB method. You also cannot use the USB method to load an image created with the programmatic method.

Table 7-1 summarizes the supported backup, restoration, and replication methods for an RT target based on whether the target is connected to the network and whether the target is USB-enabled.

Table 7-1. Supported System Replication methods for an RT Target

Connected to Network?	USB-Enabled?	Supported Methods
Yes	Yes	Both
Yes	No	Programmatic
No	Yes	USB
No	No	Neither

Backing Up RT Targets

After you install the necessary software components and drivers on an RT target and create a stand-alone RT application on the target, you can create a disk image of the target to serve as a backup image or as a prototype image for RT target replication. You can use either the programmatic method or the USB method to create an RT target disk image.

Creating RT Target Disk Images Programmatically

You can use the RT Create Disk Image VI on a Windows host computer to create an RT target disk image based on an RT target connected to the network, as shown in the Figure 7-1.

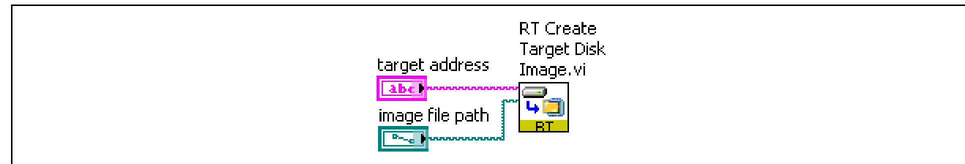


Figure 7-1. Create RT Target Disk Image Programmatically

Creating RT Target Disk Images with a USB Drive

Complete the following steps to create and store an RT target disk image on an RT Desktop PC Utility USB Drive.

1. Connect a keyboard and monitor to the RT target.
2. Insert an RT Desktop PC Utility USB Drive into the RT target. Refer to the *Measurement & Automation Explorer Help* for information about creating an RT Desktop PC Utility USB Drive.
3. Restart the target to boot from the USB drive.
4. Select **Backup, Restore, or Replicate the Real-Time System** from the National Instruments Real-Time Desktop PC Utility Collection menu.
5. Select **Backup system to default folder** to clear the default disk image folder on the USB drive and create the new disk image in the default disk image directory. You also can select **Backup system to unique folder** to create the disk image in a unique folder on the USB drive.

Restoring and Replicating RT Targets

You can apply RT target disk images to restore and replicate RT targets. You can use either the programmatic method or the USB method to apply an RT target disk image.

To restore a previously backed-up RT target in the event of a hard drive failure, apply the backup disk image to the target. To replicate an RT target, apply an RT target disk image created from the original target to other RT targets of the same model code. You can use the RT Get Target Information VI to find the model code of an RT target.

Applying an RT target disk image copies files to the hard drive of the target but does not deploy anything to memory on the target. If you need a target to run an application on restart immediately after applying a disk image, you must create a stand-alone application on the original target. To ensure that a target runs as expected after applying the disk image, you must test the original target before creating the disk image.

Note Before attempting to apply an RT target disk image, ensure that the RT target can boot the real-time operating system. Before attempting to apply an RT target disk image to an unconfigured RT Desktop PC target, follow the instructions in the Using Desktop PCs as RT Targets with the LabVIEW Real-Time Module manual in the `<labview>\manuals` directory.

Using the Programmatic Method to Apply an RT Target Disk Image

You can use the RT Apply Disk Image VI on a host computer to load an RT target disk image onto any networked RT target with the same model code as the original target from which the disk image was created. You can use the RT Get Target Information VI to find the model code of an RT target. For example, the VI shown in Figure 7-2 uses the RT Get Target Information VI to obtain the model code of a target and then uses the RT Apply Target Disk Image VI inside a Case structure to apply an appropriate RT target disk image based on the model code.

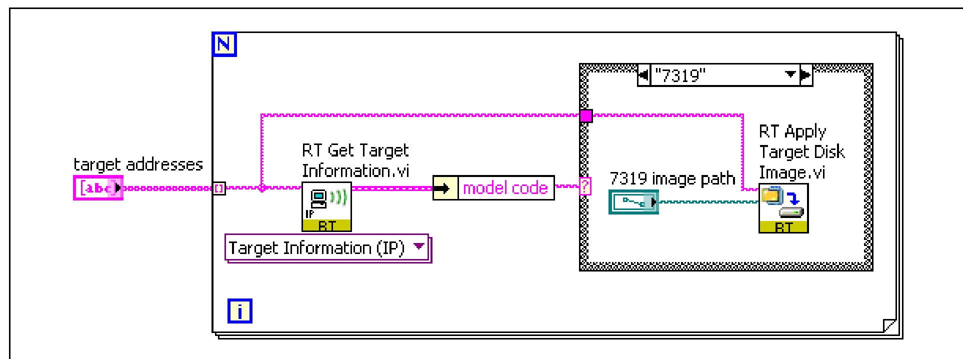


Figure 7-2. Apply an RT Disk Image Programmatically

Using the USB Method to Apply an RT Target Disk Image

Complete the following steps to load an RT target disk image from an RT Desktop PC Utility USB drive.

1. Connect a keyboard and monitor to the RT target.
2. Insert the RT Desktop PC Utility USB drive into the RT target and restart the target to boot from the USB drive.
3. Select **Backup, Restore, or Replicate the Real-Time System** from the National Instruments Real-Time Desktop PC Utility Collection menu.
4. Select **Restore or Replicate a Real-Time System** from the National Instruments Real-Time Desktop PC Utility Collection menu.
5. Select **Restore system from default folder** to load the disk image stored in the default disk image folder on the USB drive. You also can select **Select an image to restore** to load an image from a unique directory on the USB drive.

Summary – Quiz

1. True or False? Front panel property nodes are supported in a stand-alone real-time application.

2. True or False? If you are replicating a disk image onto an RT target, the RT target does not need to be the same model as the target used to create the disk image.

Summary – Quiz Answers

1. True or False? Front panel property nodes are supported in a stand-alone real-time application.

   False

2. True or False? If you are replicating a disk image onto an RT target, the RT target does not need to be the same model as the target used to create the disk image.

   False

Notes

A

Additional Information about LabVIEW Real-Time

This appendix contains useful information for LabVIEW Real-Time.

Topics

Using LabWindows/CVI DLLs in LabVIEW Real-Time

Using LabWindows/CVI DLLs in LabVIEW Real-Time

LabWindows™/CVI™ extends the functionality of LabVIEW Real-Time in two ways. First, it allows the use of ANSI C code on LabVIEW Real-Time (RT) targets. Second, LabWindows/CVI offers programmatic access to the shared memory of National Instruments RT plug-in devices, enabling you to use the LabWindows/CVI environment to develop host applications for these devices.

Benefits of Using LabWindows/CVI DLLs in Real-Time

LabWindows/CVI can compile code into a LabVIEW Real-Time compatible DLL that can be called and executed by the LabVIEW Real-Time environment. This feature reduces development time for real-time applications in three ways:

- Engineers and scientists with large amounts of existing ANSI C code greatly reduce their development time for their LabVIEW Real-Time applications through code reuse.
- Engineers and scientists can develop portions of their LabVIEW Real-Time applications in ANSI C.
- Engineers and scientists can take advantage of the LabWindows/CVI development environment to create LabVIEW Real-Time VISA drivers for non-NI PXI/PCI hardware. In doing this, third party hardware can be incorporated into RT applications.

LabWindows/CVI Functions Supported by Real-Time

LabVIEW Real-Time hardware devices include an embedded real-time operating system. The RTOS is different than traditional operating systems, such as Windows, and supports a slightly different set of functions. When creating a DLL in LabWindows/CVI, you can specify its use for LabVIEW Real-Time. This causes the LabWindows/CVI compiler to automatically verify that all the function calls made from that DLL are supported by the RTOS. To make this specification, select the **LabVIEW Real-Time Only** option as the run-time support in the Target Settings dialog box.

For further instructions, refer to the *Using LabWindows/CVI with LabVIEW Real-Time* tutorial available by browsing to `ni.com/info` and entering `rdul38`.

The following LabWindows/CVI libraries are supported for use with the LabVIEW Real-Time Support Engine:

- Analysis or Advanced Analysis Library
- ANSI C Library
- Formatting and I/O Library

- Internet Library
- Real-Time Utility Library
- TCP Support Library
- TDM Streaming Library
- UDP Support Library
- User Interface Library
- Utility Library

Note Not all of the functions in the preceding libraries are supported. For a complete list of these exported functions, refer to the *Using LabWindows/CVI Libraries in RT Applications* topic in the *NI LabWindows/CVI Help*.

In addition to the libraries listed previously you also link a **Real-time only** project to the following libraries, which may require you to install additional components on the RT target:

- RS-232 Library
- VISA Library
- NI-DAQmx Library
- Traditional NI-DAQ Library
- NI-DMM
- NI-Scope
- NI-FGEN
- NI-Switch
- NI-HSDIO
- NI-CAN

Note For a more detailed list of what functions and libraries are supported, refer to the *LabWindows/CVI Libraries in RT Applications* topic in the *NI LabWindows/CVI Help*.

TCP Library Support

LabWindows/CVI 8.x includes TCP Library function support for LabVIEW Real-Time application development. This feature allows DLLs on RT targets to share data directly with nodes on the network, eliminating the need to return to the LabVIEW code to pass data to another node. This can be done through the ProcessTCPEvents function. TCP function calls, such as TCP write or TCP read, trigger events when they are completed. In previous versions of LabWindows/CVI these events were captured by messaging, which is not supported under an RTOS. The Process TCP Events function uses polling to capture TCP events.

Use the Process TCP Events function to publish data as it is received. For example, you may have a DLL that contains a function with multiple data acquisition loops and you would like to publish the data as you acquire it. However, because DLLs pass data back to the calling application only when the function has finished executing, you would need to be able to send the data from within the DLL.

Without TCP polling functions, the only way to send the data across the network as you received it would be to implement a loop in LabVIEW and call a function in LabWindows/CVI DLL that acquired data only a single time. After you did that, each time the data was acquired, the function would end and the data would be passed back to the LabVIEW Real-Time application. After the data was back in LabVIEW, you could use the LabVIEW TCP functions to publish the data. However this requires the DLL to be called, loaded, and unloaded each time the LabVIEW loop executes. It also undermines performance. With the added functionality of the TCP function calls, the data can be sent from within the DLL in each loop, allowing the DLL to be called only once, and therefore saving time and memory.

B

Instructor's Notes

This appendix contains information that the instructor needs to properly set up and teach this course.

Course Setup

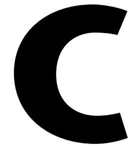
1. Confirm that all host computers are setup for static IP addresses. Course instructions are simplified if all computers use `192.168.0.1` for the IP address.

   Do not use DHCP in this course; classrooms do not have enough IP addresses allocated.

2. Place the following equipment at each student station:
   - One CompactRIO system
   - One temperature chamber
   - One cross-over cable (grey or orange)
   - One thermocouple cable
   - One six-pin cable
   - One PS-3 power supply
   - Two power cables

   The student connects the hardware in the configuration exercise. You do not need to connect/test this equipment.

3. Place the following equipment at the instructor station.
   - One CompactRIO system
   - One PXI system
   - One temperature chamber
   - One cross-over cable (grey or orange)
   - One thermocouple cable
   - One six-pin cable
   - One PS-3 power supply
   - Three power cables



Additional Information and Resources

This appendix contains additional information about National Instruments technical support options and LabVIEW Real-Time resources.

National Instruments Technical Support Options

Visit the following sections of the award-winning National Instruments Web site at `ni.com` for technical support and professional services:

- **Support—**Technical support at `ni.com/support` includes the following resources:
  - **Self-Help Technical Resources—**For answers and solutions, visit `ni.com/support` for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on. Registered users also receive access to the NI Discussion Forums at `ni.com/forums`. NI Applications Engineers make sure every question submitted online receives an answer.
  - **Standard Service Program Membership—**This program entitles members to direct access to NI Applications Engineers via phone and email for one-to-one technical support as well as exclusive access to on demand training modules via the Services Resource Center. NI offers complementary membership for a full year after purchase, after which you may renew to continue your benefits.

    For information about other technical support options in your area, visit `ni.com/services` or contact your local office at `ni.com/contact`.
- **System Integration—**If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. The NI Alliance Partners joins system integrators, consultants, and hardware vendors to provide comprehensive service and expertise to customers. The program ensures qualified, specialized assistance for application and system development. To learn more, call your local NI office or visit `ni.com/alliance`.

If you searched `ni.com` and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of `ni.com/niglobal` to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Other National Instruments Training Courses

National Instruments offers several training courses for LabVIEW users. These courses continue the training you received here and expand it to other areas. Visit `ni.com/training` to purchase course materials or sign up for instructor-led, hands-on courses at locations around the world.

National Instruments Certification

Earning an NI certification acknowledges your expertise in working with NI products and technologies. The measurement and automation industry, your employer, clients, and peers recognize your NI certification credential as a symbol of the skills and knowledge you have gained through experience. Visit `ni.com/training` for more information about the NI certification program.