LabVIEW™ FPGA Course Manual

Course Software Version 2010
October 2010 Edition
Part Number 372510D-01

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

Lesson 10
Modular Programming

Student Guide

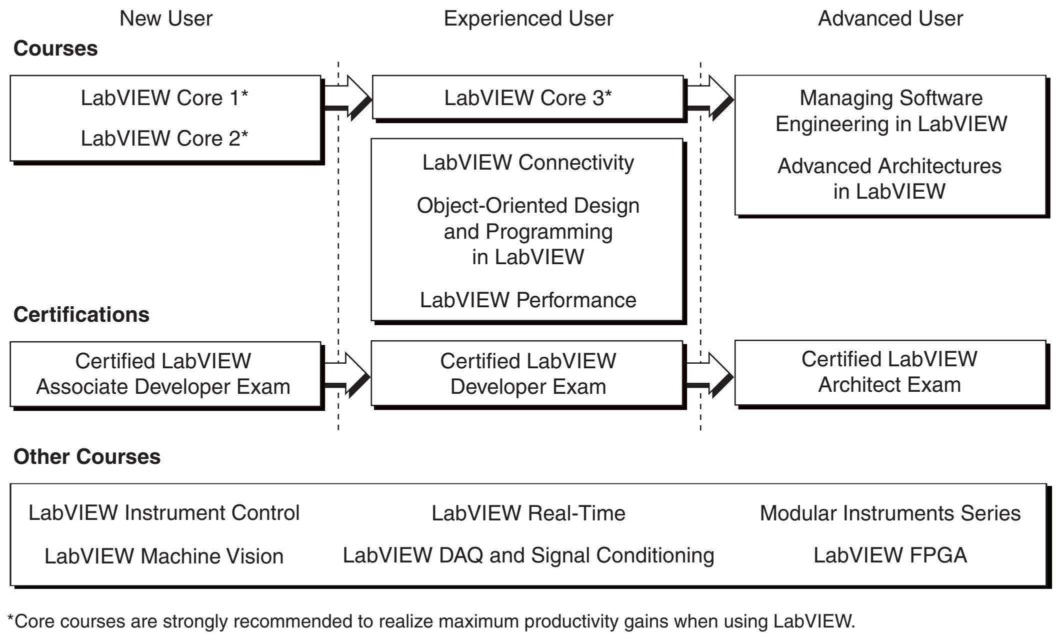
Thank you for purchasing the *LabVIEW FPGA* course kit. This course manual and the accompanying software are used in the two-day, hands-on *LabVIEW FPGA* course.

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 `lvfpga`.

A. NI Certification

The *LabVIEW FPGA* 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.


New User
Experienced User
Advanced User
Courses
LabVIEW Core 1*
LabVIEW Core 2*
LabVIEW Core 3*
LabVIEW Connectivity
Object-Oriented Design and Programming in LabVIEW
LabVIEW Performance
Managing Software Engineering in LabVIEW
Advanced Architectures in LabVIEW
Certifications
Certified LabVIEW Associate Developer Exam
Certified LabVIEW Developer Exam
Certified LabVIEW Architect Exam
Other Courses
LabVIEW Instrument Control
LabVIEW Real-Time
Modular Instruments Series
LabVIEW Machine Vision
LabVIEW DAQ and Signal Conditioning
LabVIEW FPGA
*Core courses are strongly recommended to realize maximum productivity gains when using LabVIEW.

B. Course Description

The *LabVIEW FPGA* course teaches you to extend LabVIEW to field-programmable gate array (FPGA) applications. You can use LabVIEW to create custom FPGA applications that run on NI reconfigurable I/O hardware. LabVIEW can execute block diagrams in hardware. This course assumes you have taken the *LabVIEW Core 1* course or have equivalent experience. The *LabVIEW Real-Time 1* course is recommended but not required.

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

❑ LabVIEW Full or Professional Development System 2010 or later

❑ Compatible versions of the FPGA Module, Real-Time Module, and NI-RIO software

❑ Reconfigurable I/O hardware

❑ *LabVIEW FPGA Course Exercises* manual

❑ *LabVIEW FPGA* course CD containing the following files:

Folder	Description
`Exercises`	Folder for saving VIs created during the course and for completing certain course exercises; also includes subVIs necessary for some exercises
`Solutions`	Folder containing the solutions to all the course exercises
`LabVIEW FPGA.pdf`	PDF of 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 FPGA\` 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:

- Design and implement applications using the LabVIEW FPGA Module
- Control timing and synchronization on the FPGA target
- Compile your LabVIEW FPGA VI and deploy to NI RIO hardware
- Create deterministic control and simulation solutions on the NI LabVIEW platform

This course does not present any of the following topics:

- Every built-in VI, function, or object; refer to the *LabVIEW Help* for more information about LabVIEW features not described in this course
- Developing 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 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 tip, which alerts you to advisory information.
📝	This icon denotes a note, which alerts you to important information.
⚠	This icon denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.
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.
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.
`monospace bold`	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.

1 Introduction to LabVIEW FPGA

This lesson introduces FPGA technology and LabVIEW FPGA. You will learn the components of a LabVIEW FPGA system and the types of applications that are well-suited for a LabVIEW FPGA system. You will also explore a comparison between a LabVIEW FPGA system and a traditional measurement system.

Topics

A. Introduction to FPGA Technology

Field programmable gate arrays (FPGAs) are silicon chips with unconnected logic gates. You can define the functionality of the FPGA by using software to configure the FPGA gates. You develop computing tasks in software and compile them down to a configuration file, called a bitstream, that contains information about how the gates should be wired together. In addition, FPGAs are completely reconfigurable and instantly take on a brand new "personality" when you download a different configuration of circuitry. FPGAs are often used as processing components in low- to medium-volume electronics, where the time and cost of developing and fabricating an application-specific integrated circuit (ASIC) is prohibitive. FPGAs are also found in applications that require reconfiguration in the field.

Benefits of FPGAs

FPGAs are reconfigurable through software, making them very flexible. Unlike a microprocessor or ASIC, you can modify the functionality of an FPGA. FPGAs can also execute specific tasks efficiently within silicon. Because FPGA execution is controlled by the programmable interconnects, users can implement truly parallel tasks within the FPGA that will execute simultaneously and independently of one another. Finally, because applications are implemented in hardware without an operating system, FPGAs execute applications reliably.

- **Flexibility**—FPGA technology offers flexibility and rapid prototyping capabilities. You can test an idea or concept and verify it in hardware without going through the long fabrication process of custom ASIC design. You can then implement incremental changes and iterate on an FPGA design within hours instead of weeks. Because they are reconfigurable, using FPGA chips enable you to keep up with future application needs. As a product or system matures you can make functional enhancements without spending time redesigning hardware or modifying the board layout.
- **Performance**—By taking advantage of hardware parallelism, FPGAs exceed the computing power of digital signal processors (DSPs) by breaking the paradigm of sequential execution and accomplishing more per clock cycle. Controlling inputs and outputs (I/O) at the hardware level provides faster response times and specialized functionality to closely match application requirements.
- **Reliability**—FPGA circuitry is truly a "hard-wired" implementation of program execution. FPGAs, which do not use operating systems, minimize reliability concerns with true parallel execution and deterministic hardware dedicated to every task. Processor-based systems often involve several layers of abstraction to help schedule tasks

and share resources among multiple processes. The driver layer controls hardware resources and the operating system manages memory and processor bandwidth. For any given processor core, only one instruction can execute at a time, and processor-based systems are continually at risk of time-critical tasks preempting one another.

- **Offload processing**—FPGAs can also be used to offload computationally intensive processing to free up the CPU for other tasks. FPGAs are ideal for processing-intensive applications such as image processing.
- **Cost**—The nonrecurring engineering (NRE) expense of custom ASIC design far exceeds that of FPGA-based hardware solutions. The large initial investment in ASICs is easy to justify for OEMs shipping thousands of chips per year, but many end-users need custom hardware functionality for the tens to hundreds of systems in development. The very nature of programmable silicon means that there is no cost for fabrication or long lead times for assembly. As system requirements often change over time, the cost of making incremental changes to FPGA designs are negligible when compared to the large expense of respinning an ASIC.

B. LabVIEW FPGA System

Although FPGAs have many benefits, there are some challenges when programming FPGAs. Typically, FPGAs are programmed using a hardware description language (HDL), such as VHDL or Verilog. These tools require a high level of proficiency to complete even the simplest tasks and can become a barrier to using FPGA technology. Additionally, these tools are often not intuitive for test and control engineers.

The basic components of a LabVIEW FPGA system are the LabVIEW FPGA Module, the NI-RIO driver, and an NI-RIO device, also known as a RIO target. The LabVIEW FPGA Module allows LabVIEW to target FPGAs on NI Reconfigurable I/O (RIO) hardware so scientists and engineers can take advantage of the performance and flexibility of FPGAs without needing to learn low-level design tools.

LabVIEW FPGA Module

Use National Instruments LabVIEW and the FPGA Module to develop custom control and measurement hardware without any prior knowledge of hardware description languages or board-level hardware design. VIs created with the LabVIEW FPGA Module can be targeted to the FPGAs on RIO hardware. VIs you build in LabVIEW define the logical interconnections of the logic gates on the FPGA. You can also create VIs for your host PC to interact with the FPGA target.

LabVIEW FPGA Targets

The LabVIEW FPGA Module can be used with a wide variety of National Instruments RIO devices. You cannot use the FPGA Module with non-National Instruments FPGA devices or with NI products that are not RIO devices. Refer to `ni.com/fpga` for a current list of National Instruments RIO devices. The following targets are compatible with the FPGA Module:

- **R Series Multifunction RIO**—R Series PCI or PXI plug-in devices provide analog and digital acquisition and control for high-performance, user-configurable timing and synchronization, as well as onboard decision making. Use NI R Series devices for complex data acquisition (DAQ) or real-time I/O applications.
- **NI CompactRIO**—CompactRIO is a modular, reconfigurable control and acquisition system designed for embedded applications that require high performance and reliability. Use the NI CompactRIO platform when you need a modular system with built-in signal conditioning and direct signal connectivity.
- **NI Single-Board RIO**—NI Single-Board RIO products are low-cost deployment solutions based on NI CompactRIO. They integrate a real-time processor, reconfigurable FPGA, and analog and digital I/O on a single board you can program with NI LabVIEW Real-Time and LabVIEW FPGA technologies.
- **NI FlexRIO**—The NI FlexRIO product family, composed of FPGA modules and I/O adapter modules, provides flexible, customizable I/O for NI LabVIEW FPGA. The FPGA and adapter modules form a high-performance, reconfigurable instrument that you program with the LabVIEW FPGA Module. With the NI FlexRIO Adapter Module Development Kit (MDK), you can design an instrument with the exact converters, buffers, clocks, and connectors your application requires.
- **NI Compact Vision System**—The Compact Vision System is a rugged machine vision package that contains a reconfigurable FPGA for implementing custom counters, timing, or motor control into your machine vision application.
- **RIO Instruments**—RIO instruments are FPGA-enabled devices. One of the latest RIO Instruments is the PXIe-5641R, which is a PC-based, intermediate frequency (IF) transceiver used in RF dynamic tests, software-defined radio (SDR), and user-defined IF applications.

How does LabVIEW FPGA work?

The FPGA Module compiles your LabVIEW VI to FPGA hardware. Behind the scenes, your graphical code is translated to text-based VHDL code. Then, Xilinx ISE compiler tools synthesize the VHDL code into a hardware circuit realization of your LabVIEW design. This process also applies timing constraints to the design and tries to map the design to the low-level hardware resources on the FPGA. Refer to Figure 1-1 for a graphical representation of this process.

The end result is a bitstream that contains the programming instructions LabVIEW downloads to the FPGA target. When you run the application, the bitstream is loaded into the FPGA chip and used to reconfigure the gate array logic. The bitstream can also be loaded into nonvolatile flash memory on the FPGA target and loaded instantaneously when power is applied to the FPGA target. There is no operating system on the FPGA chip, but you can control the FPGA chip through a host application.

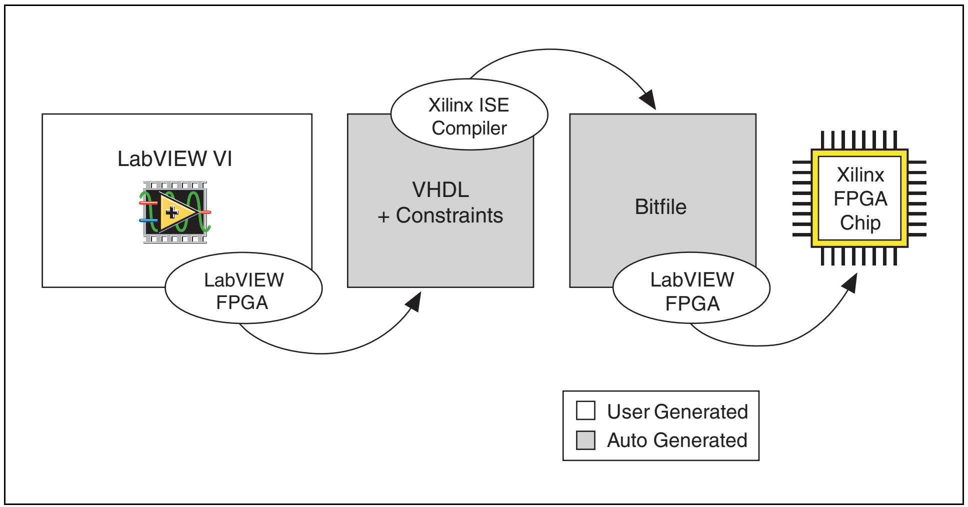


Figure 1-1. How LabVIEW FPGA Works

C. Comparison with DAQmx

To understand the paradigm of a LabVIEW FPGA system with RIO devices, you can compare it with a traditional measurement system, such as DAQmx. The NI-DAQmx driver delivers a high-performance, intuitive API with predefined counter/timer, triggering, and acquisition functions that can meet a majority of data acquisition application challenges. However, the device functionality is fixed, which means it is defined by the DAQmx API and the device hardware. Some applications may require custom functionality not available in the NI-DAQmx driver. Therefore, RIO devices help fill this gap because you have low-level hardware control of the I/O and can perform high-speed, single-point analog input and output.

To understand how an FPGA gives users more flexibility, compare the user-defined and vendor-defined components of a traditional measurement system with an FPGA system. Refer to Figure 1-2 for a representation of a DAQmx system.

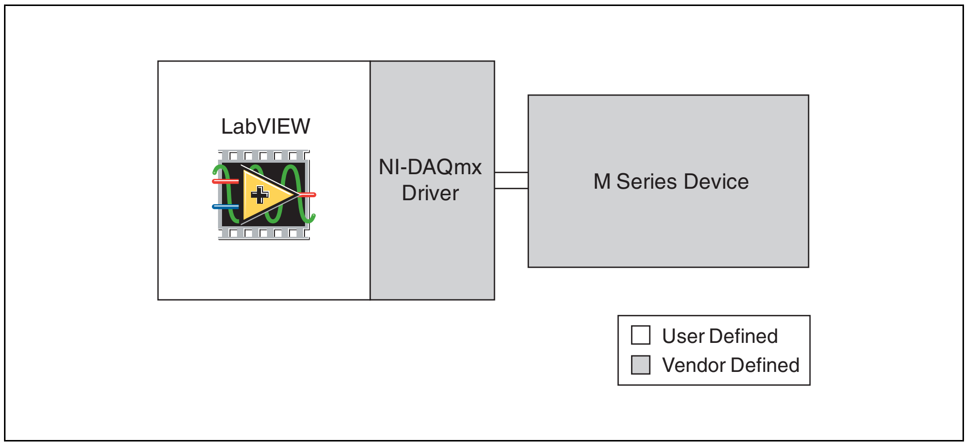


Figure 1-2. NI-DAQmx System Components

A traditional measurement and control system consists of three basic components:

- A user-defined application running in Windows or a Real-Time operating system (RTOS). This would be a VI running in the LabVIEW Development Environment.
- The vendor-defined driver software and functions to interface to your hardware. This would be the NI-DAQmx driver.
- The I/O hardware. For this specific example, we are calling out an M-series DAQ device.

When using a LabVIEW FPGA-based measurement system, the paradigm changes. Refer to Figure 1-3 for a representation of an FPGA system. Although you are still using commercial off-the-shelf (COTS) hardware from National Instruments, in this case the R Series DAQ device, you can now modify the hardware functionality of this device using the LabVIEW FPGA Module. On the RIO hardware platforms, the FPGA (instead of an ASIC) defines the device functionality. You still have the application software and a driver running in Windows or an RTOS, however, now that more functionality resides on the FPGA, the host-side driver doesn't need to be as full-featured as DAQmx. Instead, the host-side driver, called NI-RIO, only provides the interface functions to your hardware. With this paradigm, end-users define both the application running in Windows or an RTOS and on the FPGA.

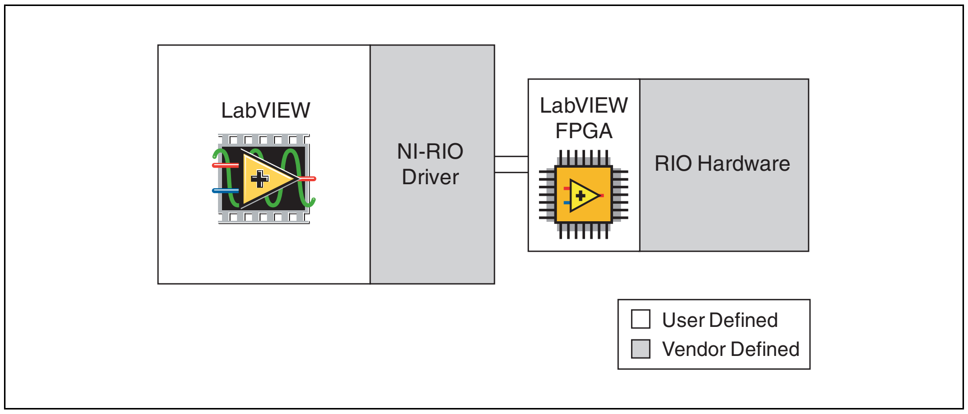


Figure 1-3. LabVIEW FPGA System Components

To understand the reliability and performance benefits of a LabVIEW FPGA system, you can compare the software and hardware layers found in a traditional measurement system with that of an FPGA system. Many test and control systems perform calculations in software. When executing in software, the calculation must be performed after multiple software calls through the application software, driver API, and operating system before interfacing with the unit under test (UUT), as shown in Figure 1-4. Even when using an RTOS and optimized algorithms, the fastest response rate you can typically achieve in software is approximately 25 µs. Additionally, a software-based system poses added vulnerability because a crash that interferes with the response of the system can occur at multiple levels.

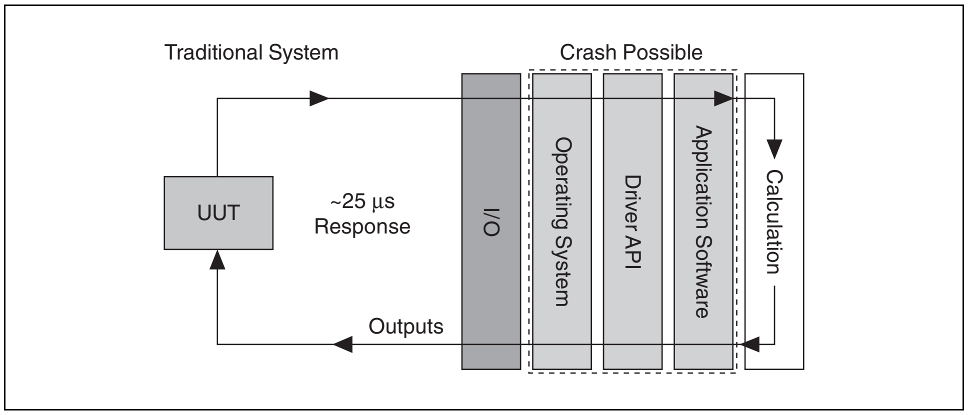


Figure 1-4. DAQmx Software and Hardware Layers

When executing calculations in hardware, as with LabVIEW FPGA, you can remove software from the required response to the UUT. In the configuration shown in Figure 1-5, the LabVIEW FPGA system can respond to digital signals within a single clock cycle. With a default clock rate of 40 MHz, the LabVIEW FPGA system can respond to a digital signal within 25 ns. In some cases, you can compile LabVIEW FPGA code at higher rates.

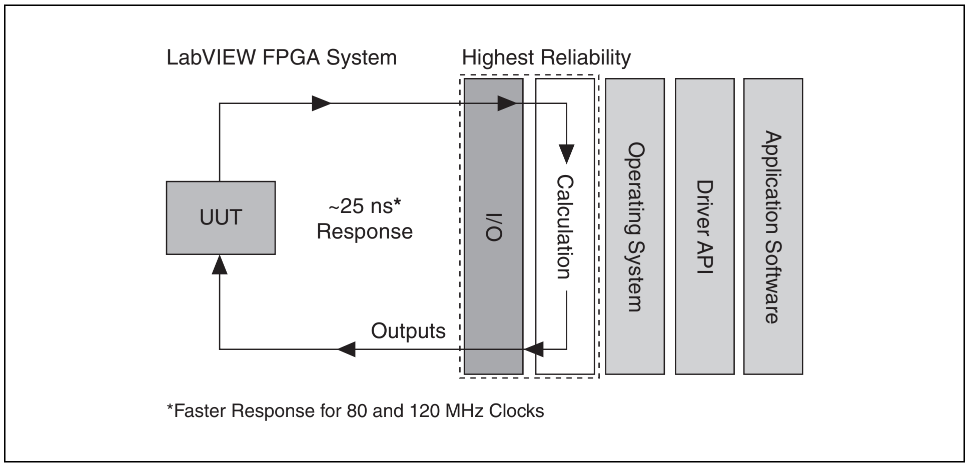


Figure 1-5. FPGA Software and Hardware Layers

Performing calculations in hardware provides the highest reliability possible because a crash at any software layer will not affect the execution of your code.

D. LabVIEW FPGA Applications

The LabVIEW FPGA system with RIO technology is ideal for applications such as the following:

- **On-Board Processing**—Acquire signals and implement inline processing for data reduction, filtering, or other digital signal processing (DSP).
- **High-Speed Control**—Use single-point I/O with a proportional-integral-derivative (PID) or other control algorithm to implement onboard control with loop rates up to hundreds of KHz.
- **Digital Communications and Protocols**—Interface with standard and/or custom digital protocols.
- **Off-Load CPU Processing**—Communicate with direct memory access (DMA) by outsourcing the many processing algorithms that are well-suited for FPGA hardware but CPU intensive on processor-based systems.
- **Complex Timing and Synchronization**—Implement complex triggering schemes, multi-rate acquisition, or supply clock signals to synchronize external devices.
- **Hardware-in-the-Loop (HIL) Testing**—Simulate sensors, emulate protocols, and implement important parts of HIL systems that are difficult to implement on processor-based systems.

Self-Review: Quiz

1. Which of the following are required software components of a LabVIEW FPGA system?
   a. NI-RIO driver
   b. DAQmx driver
   c. LabVIEW Development System
   d. LabVIEW Real-Time Module
   e. LabVIEW FPGA Module

2. What OS runs on the FPGA?
   a. LabVIEW FPGA
   b. MicroLinux
   c. Unix
   d. PharLap
   e. None of the above

3. The LabVIEW FPGA module allows you to program an FPGA without knowing VHDL or Verilog.
   a. True
   b. False

4. Which of the following might be typical reasons for using an R Series Multifunction RIO device over a DAQ board programmed with DAQmx?
   a. Need more counters than available on a DAQ board
   b. Need custom trigger logic
   c. Need more analog input channels than available on a DAQ board
   d. Need to simultaneously acquire data on different channels at different rates
   e. Need a high-level/easy-to-use driver

Self-Review: Quiz Answers

1. Which of the following are required software components of a LabVIEW FPGA system?

   a. NI-RIO driver

   b. DAQmx driver

   c. LabVIEW Development System

   d. LabVIEW Real-Time Module

   e. LabVIEW FPGA Module

   DAQmx is not used to program RIO devices. The LabVIEW Real-Time module is only needed if you are also using a Real-Time target such as CompactRIO. It is not needed when using an R Series device. You will learn when the LabVIEW Real-Time module is required in the next lesson.

2. What OS runs on the FPGA?

   a. LabVIEW FPGA

   b. MicroLinux

   c. Unix

   d. PharLap

   e. None of the above

   FPGAs do not use an operating system. This is why you can achieve parallelism offered by FPGAs.

3. The LabVIEW FPGA module allows you to program an FPGA without knowing VHDL or Verilog.

   a. True

   b. False

   Although there are ways of incorporating HDL into a LabVIEW application, you do not need HDL knowledge to program a LabVIEW FPGA system.

4. Which of the following might be typical reasons for using an R Series Multifunction RIO device over a DAQ board programmed with DAQmx?

   a. Need more counters than available on a DAQ board. With an FPGA, you can turn a digital line into a counter. You are not limited to counters available in hardware.

   b. Need custom trigger logic. With an FPGA, you can implement custom triggers that are not available through DAQmx or supported on the device.

   c. Need more analog input channels than available on a DAQ board. You cannot increase the number of analog or digital channels by just going to an FPGA device. I/O are fixed for most RIO devices. With FlexRIO you do have some flexibility, but typically increasing number of analog input channels isn't a reason to convert your application to use FPGA technology.

   d. Need to simultaneously acquire data on different channels at different rates. You can implement an application where each channel is acquiring data at different, specified rates.

   e. Need a high-level/easy-to-use driver. DAQmx provides a high-level API and will typically be easier to program than an FPGA.

Notes

Notes

2 LabVIEW FPGA Basics

In this lesson, you learn about the two major FPGA system architectures: FPGA–Windows and FPGA–Real-Time. You also learn about R Series devices and CompactRIO, two of the different RIO platforms. You then learn to configure your RIO hardware in Measurement and Automation Explorer (MAX) and create a LabVIEW FPGA project.

Topics

A. Evaluate System Requirements

Figure 2-1 shows the basic steps of developing a LabVIEW FPGA application.

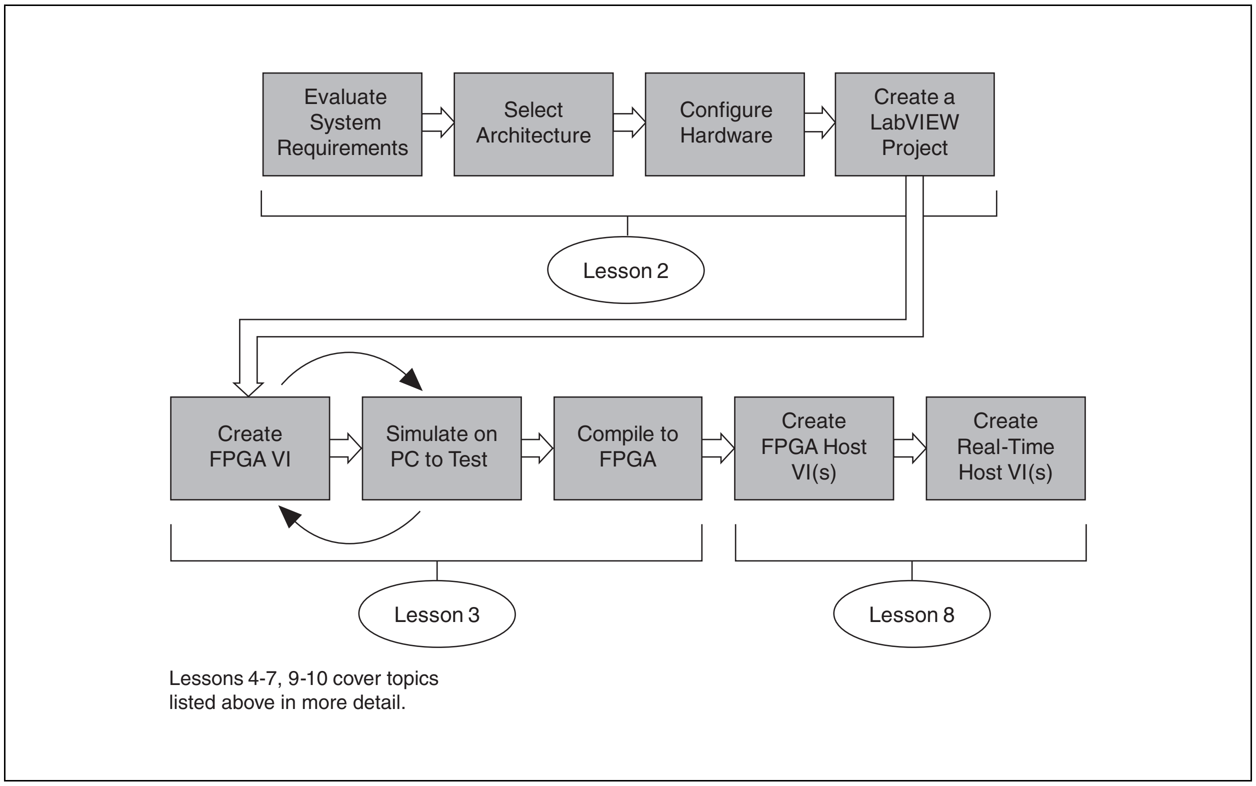


Figure 2-1. FPGA Development Flow

The first step in any development project is to evaluate the requirements of the system. Create a list of all of the tasks the system should perform and any restrictions on the system, such as development time, cost, and size. If you need an FPGA system, the first consideration is the system architecture.

- Will the system use a Windows-based operating system or a Real-Time operating system?

After that, you must select the packaging, or form-factor, for your system.

- Will you use a desktop PC or a PXI system?
- Do you need a plug-in board?
- Do you need an embedded system such as CompactRIO?

After you decide on the form factor, there are other considerations.

- What type of I/O channels do you need and how many do you need?
- Does your system require minimal use of the FPGA or extensive FPGA coding?

The amount of FPGA coding affects the size of the FPGA needed for the application. Refer to `ni.com/fpga` for help with your purchasing decisions.

Throughout this course, the lessons illustrate concepts for the two fundamental FPGA system architectures—FPGA–Windows and FPGA–Real-Time. To illustrate the differences based on both architecture and form factor, course exercises are based around two systems: a PCI R Series board and a CompactRIO system. Most concepts apply to both systems. However, in areas where the systems differ, such as configuration, the course material covers both.

B. FPGA System Architectures

In this section, you will learn about the two fundamental FPGA system architectures: FPGA–Windows and FPGA–Real-Time.

LabVIEW FPGA for Windows

Figure 2-2 illustrates an FPGA system where the host device is a PXIe, PXI, or desktop PC running the Windows operating system. The RIO device is a PXIe, PXI, or PCI plug-in device. This system architecture uses the following VIs:

- LabVIEW FPGA VI—Executes on the FPGA of the PXIe, PXI, or PCI RIO device.
- Host VI—Executes on the host PC, controls the FPGA, and provides a user interface to the system.

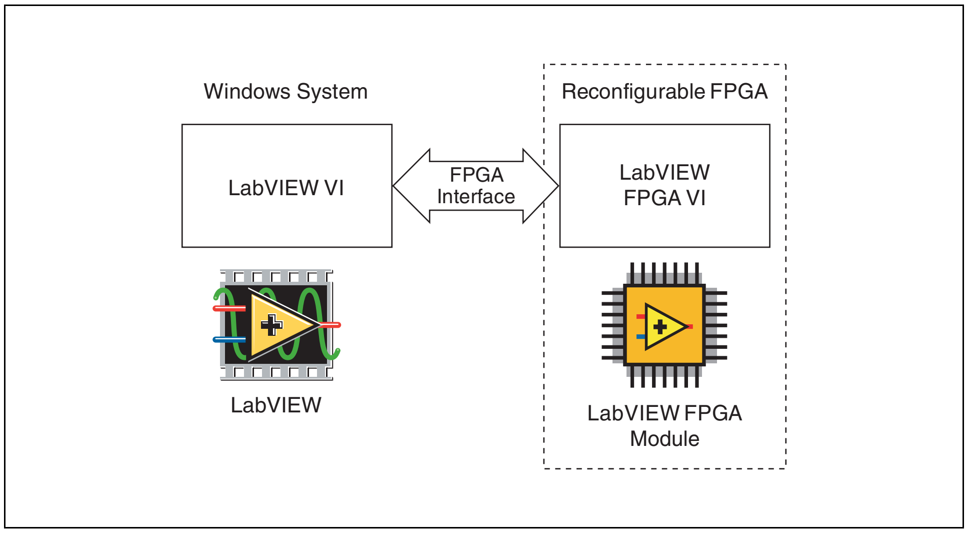


Figure 2-2. LabVIEW FPGA Architecture for Windows

LabVIEW FPGA for Real-Time

Figure 2-3 illustrates an FPGA system where the FPGA is running within a LabVIEW real-time system. The real-time system is typically on a PXIe, PXI, or CompactRIO controller. This system architecture uses the following VIs:

- LabVIEW FPGA VI—Executes on the FPGA of the device.
- LabVIEW Real-Time VI—Executes on a dedicated processor running a real-time operating system and adds deterministic, floating-point processing and control algorithms to the system.
- Host VI—Executes on the host PC and contains the user interface for the real-time system.

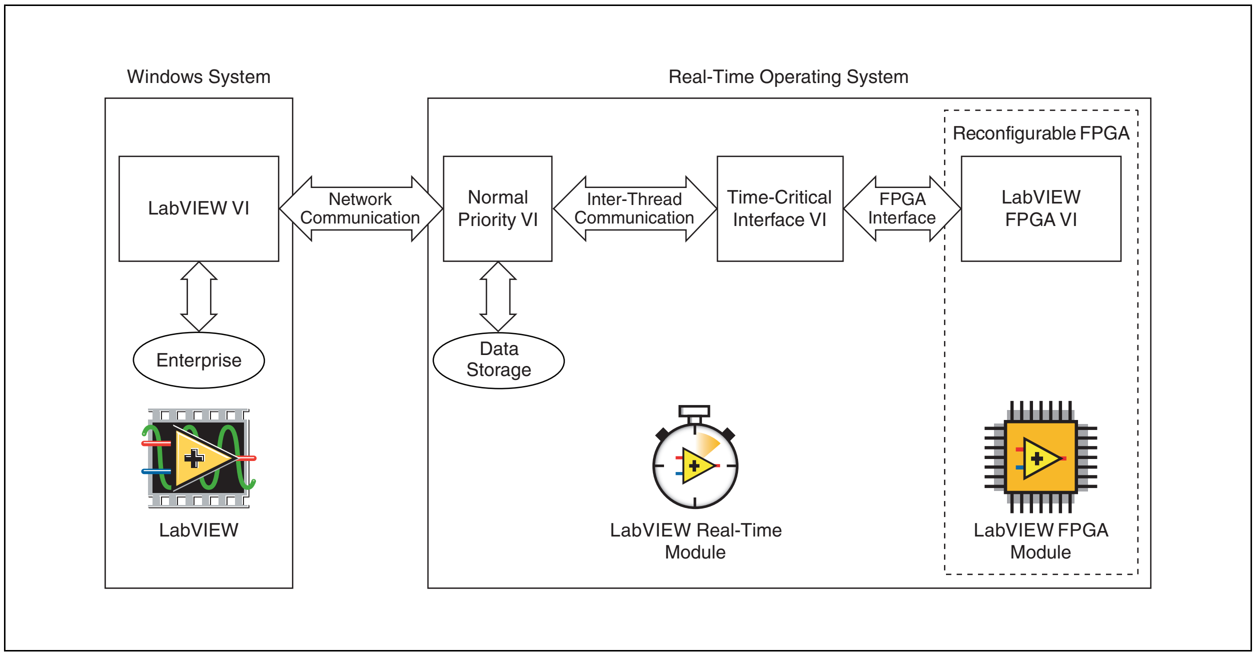


Figure 2-3. LabVIEW FPGA Architecture for Real-Time

Refer to the *LabVIEW Real-Time 1* course or visit `ni.com/realtime` for more information about developing well-architected LabVIEW Real-Time applications.

C. Reconfigurable I/O (RIO) Platforms

After you select your architecture, you must specify a RIO device and accessories. Refer to `ni.com/fpga` for a complete list of all available RIO devices and accessories. Two common RIO platforms are R Series Multifunction RIO and CompactRIO.

R Series Multifunction RIO

The R Series Multifunction RIO devices come in either PCI or PXI form factors. PCI cards require desktop PCs running a Windows operating system. PXI systems can run Windows or real-time operating systems.

You can customize your own data acquisition using the NI R Series Multifunction RIO devices, which integrate FPGA technology with analog and digital I/O. Use these devices for high-speed applications that require precise timing and control.

Digital R Series devices have 160 digital lines that can be configured for inputs, outputs, counters, or custom logic at rates up to 40 MHz.

Figure 2-4. R Series Multifunction RIO Device

CompactRIO

CompactRIO is a small, rugged, modular embedded control and data acquisition system. Consider the various components displayed in Figure 2-5 when designing your CompactRIO system.

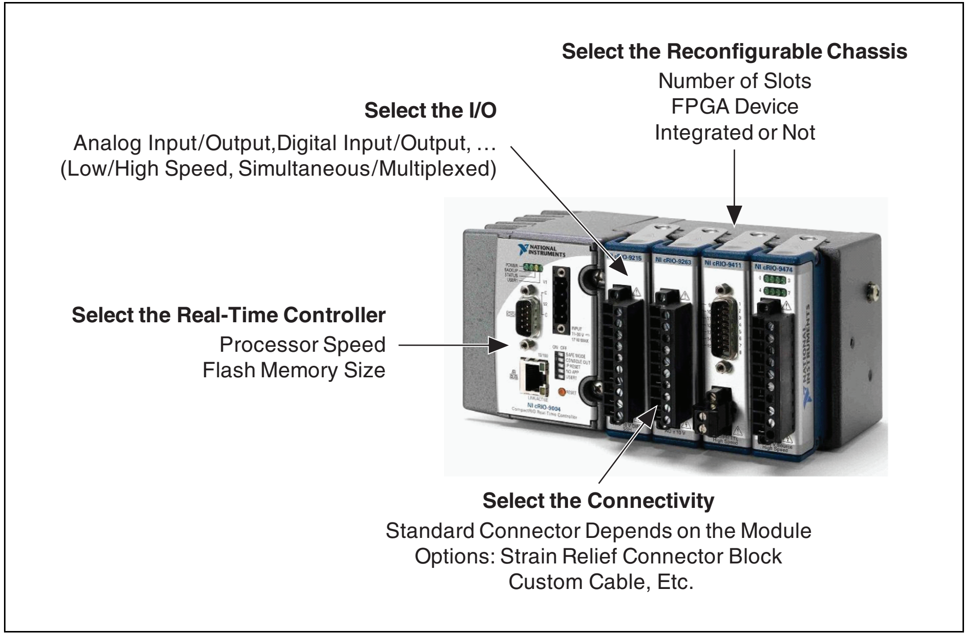

Select the Reconfigurable Chassis
Number of Slots
FPGA Device
Integrated or Not
Select the I/O
Analog Input/Output,Digital Input/Output, …
(Low/High Speed, Simultaneous/Multiplexed)
Select the Real-Time Controller
Processor Speed
Flash Memory Size
Select the Connectivity
Standard Connector Depends on the Module
Options: Strain Relief Connector Block
Custom Cable, Etc.

Figure 2-5. CompactRIO Components

The CompactRIO controller is an embedded real-time processor. It communicates with the I/O modules, performs embedded control and logging, and communicates through Ethernet and serial protocols. When selecting a controller, consider processor speed and flash memory size. The controller module communicates with the I/O modules through the FPGA on the chassis. Some controllers are integrated with the chassis, while others are separate components. There are 4- and 8-slot chassis available. Chassis and FPGA device size are dependent on how many CompactRIO modules you use, how much I/O you need, and the additional requirements you have for the FPGA.

The broad selection of I/O modules include Analog I/O with various signal conditioning, Thermocouple, RTD, Bridge Sensors, Digital I/O, Counters, Pulse Generation, and so on. Refer to `ni.com/compactrio` for more information about the CompactRIO platform.

The CompactRIO family includes other form factors. For example, the CompactRIO Integrated Controller is a lower cost solution than systems with separate controllers and chassis. This integrated solution is ideal for volume usage. On the other hand, with separate controllers and chassis, you can upgrade your processor speed, FPGA size, and/or number of slots independently of each other. The non-integrated solution might be a better choice while developing your application. When you want to replicate it, the integrated system might be a better choice.

NI Single-Board RIO is a low-cost embedded deployment solution based on NI CompactRIO platform. It is intended for high volume/OEM usage. You can program it as you do other CompactRIO devices, and you can customize the enclosure.

D. System Configuration

To set up and configure your FPGA system, complete the following steps:

1. Install the appropriate software and driver.
2. Set up your FPGA hardware and host computer.
3. Configure your target.

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

Software Installation

Before you configure your hardware, you must install the application software and driver software for your system.

FPGA–Windows

If you are not using a real-time system, you install the following products in order:

1. LabVIEW Development System (Full or Professional)
2. LabVIEW FPGA Module
3. NI-RIO drivers

FPGA–Real-Time

If you are using a real-time system, install the following products in order:

1. LabVIEW Development System (Full or Professional)
2. LabVIEW Real-Time Module
3. LabVIEW FPGA Module
4. NI-RIO drivers

Note You can install the LabVIEW Real-Time Module and the LabVIEW FPGA Module as part of the LabVIEW Platform DVD. You can install NI-RIO as part of the Driver CD.

Hardware Set-Up and Configuration

After you install the software, assemble and install your FPGA device, cables and accessories. You then configure your hardware in MAX.

FPGA–Windows

To confirm that your device is installed and recognized by the host computer, complete the following steps:

1. Select **Start»All Programs»National Instruments»Measurement & Automation** to open MAX.
2. Expand My System in the MAX configuration tree. Expand Devices and Interfaces.
3. Verify that the device appears under **Devices and Interfaces»RIO Devices**.

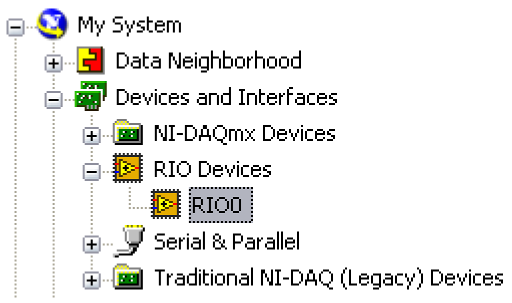


Figure 2-6. Hierarchy for an FPGA Target in MAX

FPGA–Real-Time

If you are using a real-time system, the FPGA device is on a remote system. To confirm a connection and configure the hardware, you will complete the following steps, which are described in more detail in the following sections:

1. Selecting a host-target network set-up
2. Using MAX to detect the remote system
3. Configuring the target network settings
4. Viewing devices and interfaces
5. Adding or removing software on the target

Selecting a Host-Target Network Set-Up

You must establish communication between your host computer and your remote system using an Ethernet connection. You have two options for connecting to your remote system:

- Place the host computer and remote system on a local area network. Use a standard Ethernet cable to connect your remote system to the network. Both the host and remote system must be on the same subnet.
- Connect your host computer to your remote system using a crossover cable.

Using MAX to Detect the Remote System

Select **Start»All Programs»National Instruments»Measurement & Automation** to open MAX. Expand **Remote Systems** in the 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 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. 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.

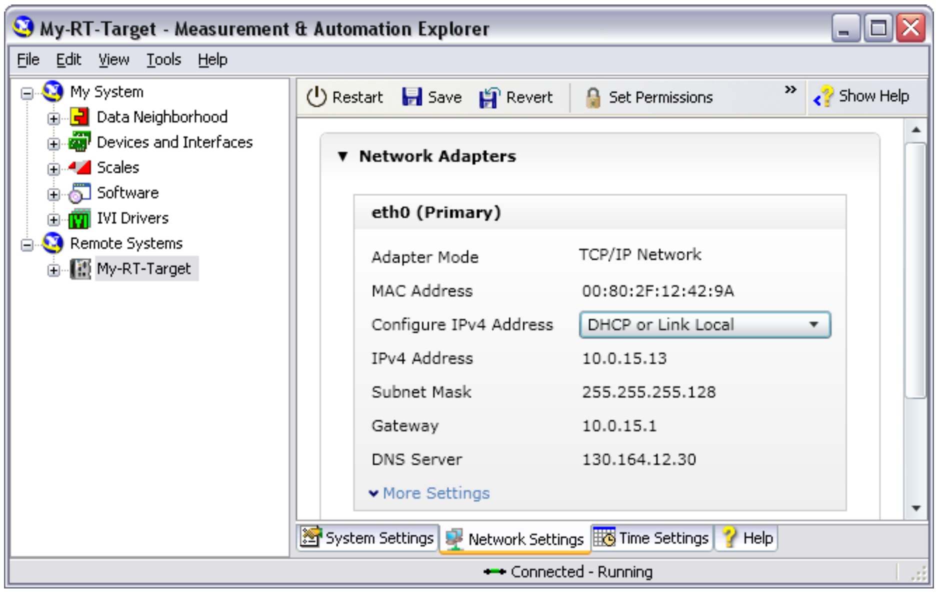
Configuring the Target Network Settings

Use the Network Settings tab of the configuration view to set the network IP address and assign a host name to your remote system. You must set the network IP address if your device is unconfigured or if you need to modify the IP settings. You can obtain the IP address automatically or specify it manually.

Obtaining an IP Address Automatically

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.



eth0 (Primary)

Adapter Mode	TCP/IP Network
MAC Address	00:80:2F:12:42:9A
Configure IPv4 Address	DHCP or Link Local
IPv4 Address	10.0.15.13
Subnet Mask	255.255.255.128
Gateway	10.0.15.1
DNS Server	130.164.12.30

Figure 2-7. 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.

Manually Specifying an IP Address

If you are unable to automatically assign an IP address, select **Use the following IP address**. You can specify a static IP address or click on the **Suggest Values** button to launch the IP Address Suggestion dialog box. The IP address of the device shown in Enter IP Address of Device will have numbers that match the first three numbers of the host computer IP address. The forth number must be set to a number that differs from the host computer. The dialog recommends a number, but you can change it to any number between 0 and 255. Select **OK**.

After verifying the IP settings on the Network Settings tab, click **Apply**. You must restart the remote system for any changes to take effect.

Note If you are using a crossover cable, your Windows system will default to a link-local address. Since the remote target needs to be on the same subnet as the host computer, the suggested IP address will be in the range of `169.254.x.x`.

You have to specify the following items to configure the IP for your target:

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 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.

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`.

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.

Viewing Devices and Interfaces

Once you establish a network connection with your remote device, expand Devices and Interfaces in the Configuration tree to view your detected FPGA target. By default your FPGA target will appear as a numbered RIO device, such as RIO0.

Adding or Removing Software on the Target

You should verify that you have the correct software installed on the RT target.

You can also update or install the LabVIEW Real-Time engine or other driver software on the remote target.

If your RT target has the LabVIEW Real-Time engine 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.

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 **Add/Remove Software** to launch the LabVIEW Real-Time Software Wizard.
3. Use the LabVIEW Real-Time Software Wizard to add, remove, or update the software on your remote target. For most FPGA applications, the minimal installation option selection is sufficient. If your real-time application needs additional features, you may need additional functionality available in other installation options.

E. Creating a LabVIEW FPGA Project

You must create a LabVIEW project before you can create an application using an FPGA target. You then can add an FPGA target to the project and create FPGA VIs.

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 a Real-Time or FPGA target.

Each project can have multiple targets representing the host computer as well as real-time and FPGA systems. 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.

Add 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 into the auto-populated folder. Items in the virtual folder that do not exist in the directory on disk move to the target.

Creating an FPGA Project

The process for creating a LabVIEW FPGA Project differs depending on your FPGA target. For R Series devices and other FPGA targets installed on your PC, you only add the FPGA target to your project. For CompactRIO and other Real-Time targets, you add your target, chassis, and modules. If the device is connected on the network, LabVIEW can autodetect the chassis and modules.

You also can create a new project with the FPGA Project Wizard. Refer to the *FPGA Project Wizard* topic the *LabVIEW Help* for more information about creating projects with the FPGA Project Wizard.

Note The FPGA Project Wizard differs from the FPGA Wizard.

FPGA–Windows

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. Right-click the My Computer target and select **New»Targets and Devices** to display the Add Targets and Devices dialog box.
3. Select the type of FPGA 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 FPGA targets:
   - Existing target or device—Select this option for online devices that are installed and detected in MAX.
   - New target or device—Select this option for offline devices.
4. Select a target and click **OK**.
5. An item representing the FPGA target appears in the Project Explorer window.
6. Select **File»Save** to save the project.

FPGA–Real-Time

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. 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 CompactRIO systems and PXI systems.
3. 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. Select this option for online devices that are configured in MAX.
   - New target or device. Select this option for offline devices.
4. Select a target and click **OK**.
5. If you are connecting with a CompactRIO device, select **LabVIEW FPGA Interface** in the Select Programming Mode dialog box.

Note If you are connecting with an online CompactRIO device, you can optionally have LabVIEW discover your C Series Modules.

6. An item representing the RT target appears in the Project Explorer window.
7. Select **File»Save** to save the project.

Verifying the Target Connection in a Project

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 LabVIEW is successful in connecting to your target, a bright green dot appears next to the target name in the Project Explorer. If you do not manually connect to the target, LabVIEW connects automatically when you run a VI on the target.

Note You can change your project to use another target IP address by right-clicking the target and selecting **Properties**. If you are using a DHCP server, you can specify the DNS name for your RT target. For a CompactRIO controller, the DNS name is the name you specified in MAX.

Self-Review: Quiz

1. Which of the following operations are performed in MAX?
   a. Install software on an R Series board
   b. Install software on a CompactRIO target
   c. Verify connection between the host and a real-time target
   d. Assign the IP address of your host computer

2. Referring to Figure 2-8, under which node do you add a CompactRIO target?
   a. Real-Time.lvproj
   b. My Computer
   c. Dependencies
   d. Build Specifications

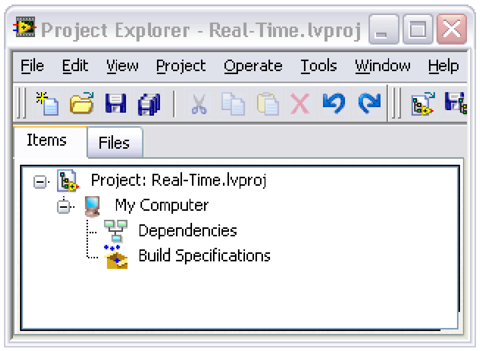
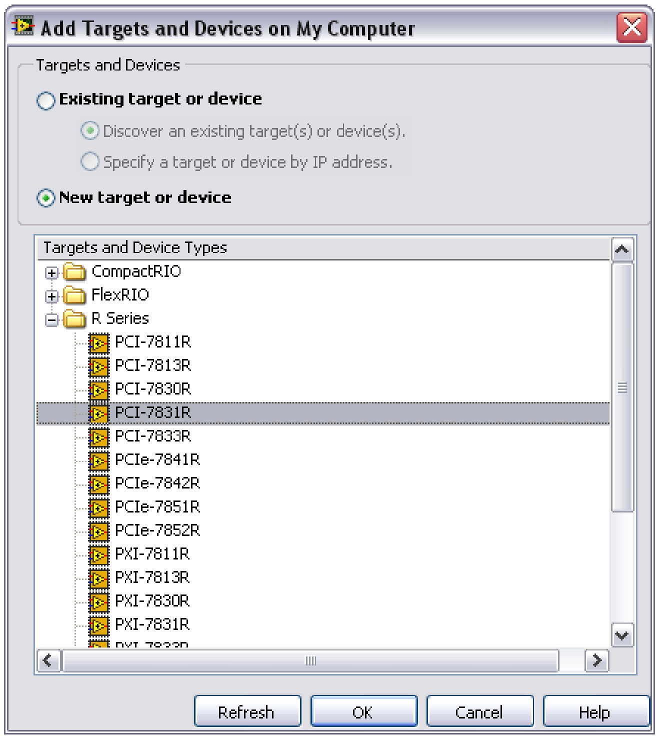

Project Explorer - Real-Time.lvproj
File Edit View Project Operate Tools Window Help
Items
Files
Project: Real-Time.lvproj
My Computer
Dependencies
Build Specifications

Figure 2-8. Real-Time LabVIEW Project

3. Referring to Figure 2-9, when would you select New target or device?
   a. You are evaluating an FPGA target to see if your application can run on the target
   b. You are developing your application while traveling and you don't have an FPGA target connected to your laptop
   c. You are curious to see which NI-RIO targets and devices are supported under My Computer
   d. All of the above


Add Targets and Devices on My Computer

Targets and Devices

( ) **Existing target or device**
  (•) Discover an existing target(s) or device(s).
  ( ) Specify a target or device by IP address.
(•) **New target or device**

Targets and Device Types
- CompactRIO
- FlexRIO
- R Series
  - PCI-7811R
  - PCI-7813R
  - PCI-7830R
  - PCI-7831R
  - PCI-7833R
  - PCIe-7841R
  - PCIe-7842R
  - PCIe-7851R
  - PCIe-7852R
  - PXI-7811R
  - PXI-7813R
  - PXI-7830R
  - PXI-7831R

Figure 2-9. Add Targets and Devices Dialog

Self-Review: Quiz Answers

1. Which of the following operations are performed in MAX?

   a. Install software on an R Series board. MAX is used to install NI-RIO and other software onto a Real-Time controller such as CompactRIO. You do not install NI-RIO onto a R Series device.

   b. Install software on a CompactRIO target

   c. Verify connection between the host and a real-time target

   d. Assign the IP address of your host computer. You set the IP address of the controller in MAX. Your Windows IP address is configured in your Local Area Network properties.

2. Referring to Figure 2-10, under which node do you add a CompactRIO target?

   a. Real-Time.lvproj. Since CompactRIO is a distributed system and doesn't reside on the host computer, the CompactRIO controller target is added directly under the project. PCI R Series devices are added under My Computer.

   b. My Computer

   c. Dependencies

   d. Build Specifications

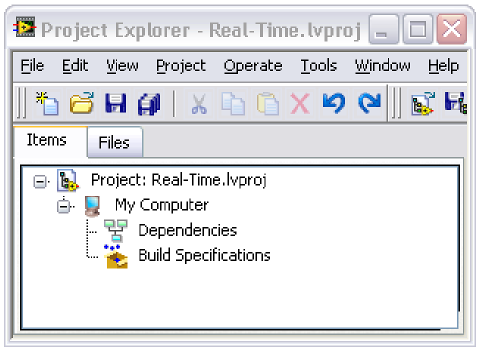
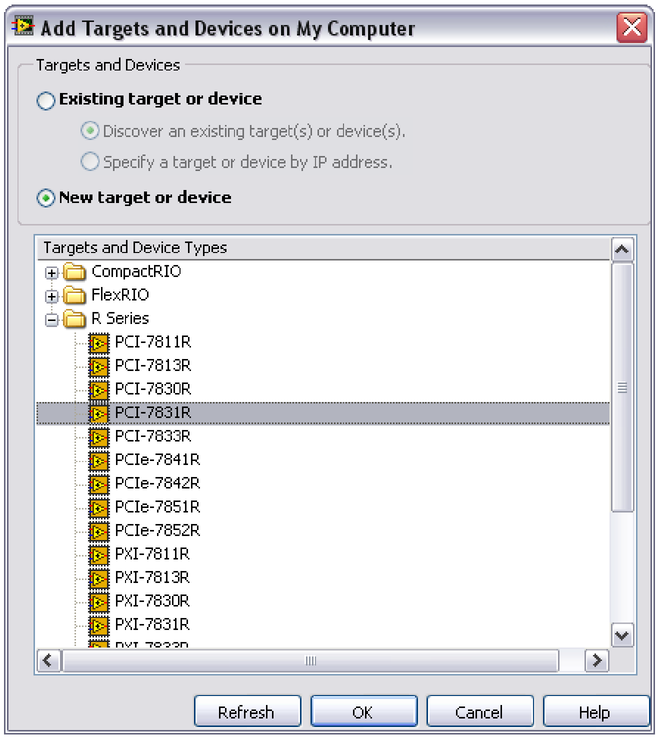


Figure 2-10. Real-Time LabVIEW Project

3. Referring to Figure 2-11, when would you select New target or device?
   a. You are evaluating an FPGA target to see if your application can run on the target
   b. You are developing your application while traveling and you don't have a FPGA target connected to your laptop
   c. You are curious to see which NI-RIO targets and devices that are supported under My Computer
   d. All of the above. There are many times where the desired hardware is not physically connected to your computer but you'd like to build an application, in which case you can still add the target or device manually.


Add Targets and Devices on My Computer

Targets and Devices

Existing target or device
- Discover an existing target(s) or device(s).
- Specify a target or device by IP address.

[✓ New target or device]

Targets and Device Types
- CompactRIO
- FlexRIO
- R Series
  - PCI-7811R
  - PCI-7813R
  - PCI-7830R
  - PCI-7831R
  - PCI-7833R
  - PCIe-7841R
  - PCIe-7842R
  - PCIe-7851R
  - PCIe-7852R
  - PXI-7811R
  - PXI-7813R
  - PXI-7830R
  - PXI-7831R

Figure 2-11. Add Targets and Devices Dialog

Notes

Notes

3

FPGA Programming Basics

In this lesson, you learn how to configure an FPGA chip using the LabVIEW FPGA Module. You will gain a high-level understanding of how logic is implemented on the FPGA and how LabVIEW code is translated and compiled into FPGA hardware. After you develop an FPGA VI, you can test and debug on the development computer and then compile and execute the VI on an FPGA target using Interactive Front Panel Communication. You examine different reports that are generated during compilation and learn several ways to optimize your code for size.

Topics

A. Introduction

An FPGA is a chip that can be reconfigured through software for different applications. The National Instruments LabVIEW FPGA Module targets NI reconfigurable I/O (RIO) devices, such as R Series data acquisition (DAQ), Compact Vision, and CompactRIO. Refer to `ni.com` for a current list of devices. R Series DAQ devices support complex data acquisition or real-time I/O applications. FPGA logic on an NI Compact Vision system adds custom triggering, pulse width modulation (PWM) signals, motion control, and custom communications protocols. CompactRIO uses FPGA for modularity, FPGA-timed I/O with built-in signal conditioning, and direct signal connectivity for maximum flexibility in embedded measurement and control applications.

A single FPGA can replace thousands of discrete components by incorporating millions of logic gates into a single integrated circuit (IC) chip. Figure 3-1 shows the FPGA as a reconfigurable digital architecture with a matrix of configurable logic blocks (CLBs) with horizontal and vertical routing channels surrounded by a periphery of I/O blocks.

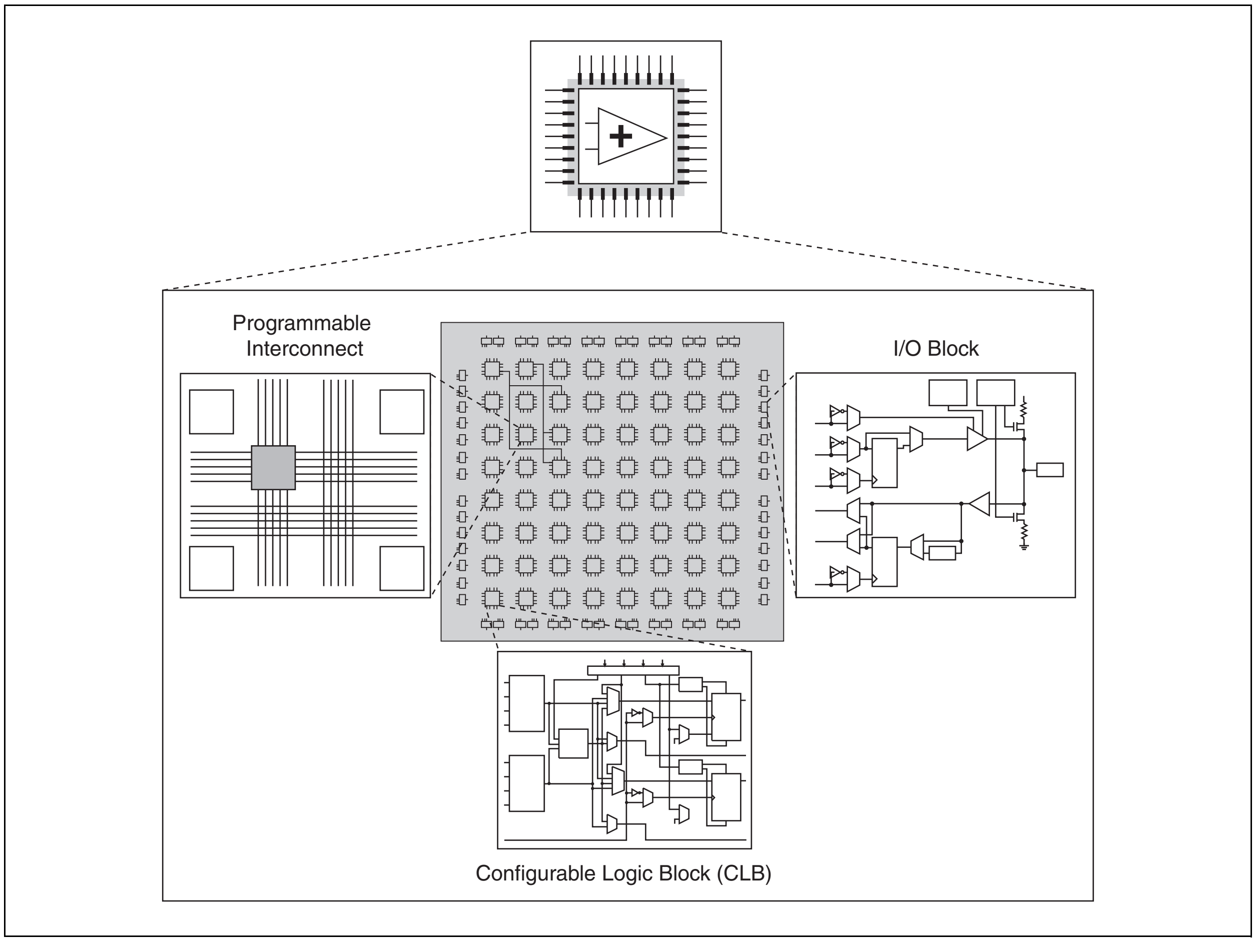


Figure 3-1. FPGA Chip

Signals can be routed within the FPGA matrix in any arbitrary manner by programmable interconnect switches and wire routes. The switches, known as register flip-flops, pass data from input to output on a rising edge of a clock. VIs you build in LabVIEW define the logical interconnections between CLBs. The VI can use look-up tables (LUT), also known as a truth tables, that define outputs from all possible input values to a logic function. Typical logic functions include Boolean operations, comparisons, and basic mathematical operations.

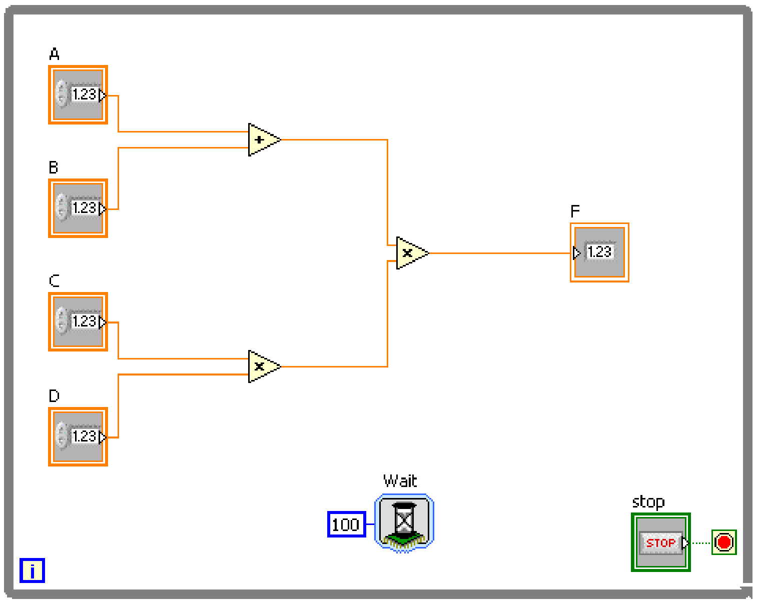
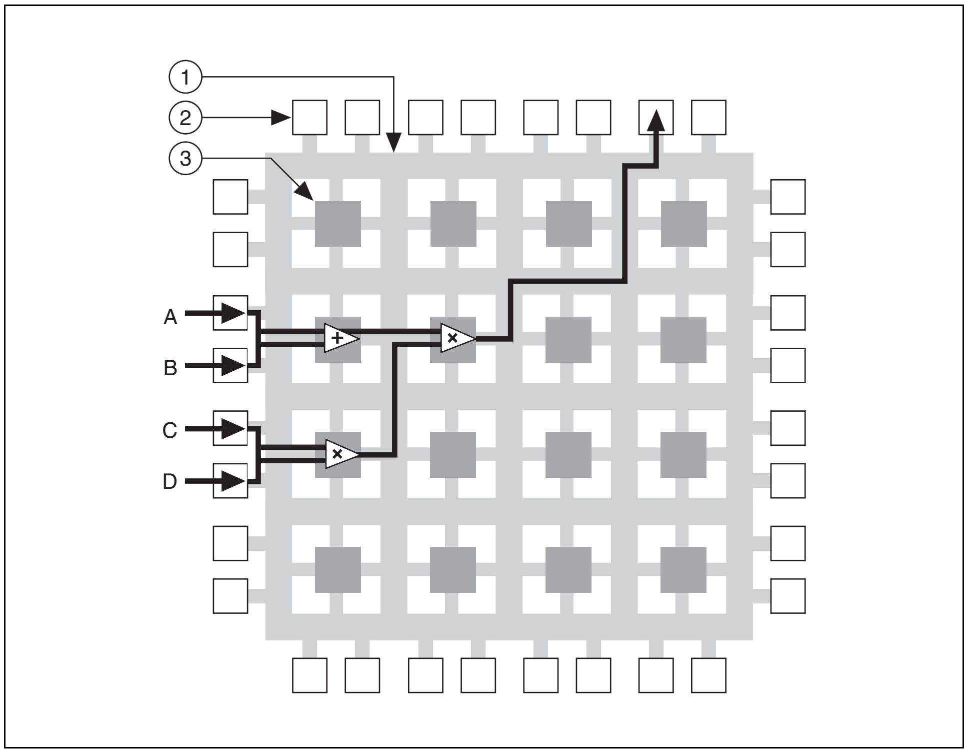


Figure 3-2. Simple LabVIEW VI

Figure 3-2 shows a simple VI that calculates a value for *F* from inputs *A*, *B*, *C*, and *D* where $F = (A + B) \times (C \times D)$. Figure 3-3 shows how the VI logic is implemented on an FPGA.



1	Interconnect Resources	2	I/O Cells	3	Logic Blocks

Figure 3-3. Implementing VI Logic on an FPGA

An FPGA is analogous to a printed circuit board that has a large number of unconnected components on it. Traditionally, the components are connected with physical wires soldered to the pins with a wire wrapping tool, or embedded in the printed circuit board. The physical wires are difficult to modify. However, the connections in an FPGA circuit are dynamically defined in software. During the FPGA compilation, the VI is optimized to reduce digital logic and create an optimal implementation of the LabVIEW application. The end result is a bitstream file. The bitstream turns semiconductor switches on or off, thereby defining the connections between gates.

B. Defining FPGA Logic in LabVIEW

The FPGA circuitry is a parallel-processing, reconfigurable computing engine that executes a LabVIEW VI in silicon circuitry on a chip. As shown in Figure 3-4, you can use the LabVIEW FPGA Module to define FPGA logic using LabVIEW VIs instead of low-level languages such as very high speed integrated circuit hardware description language (VHDL).

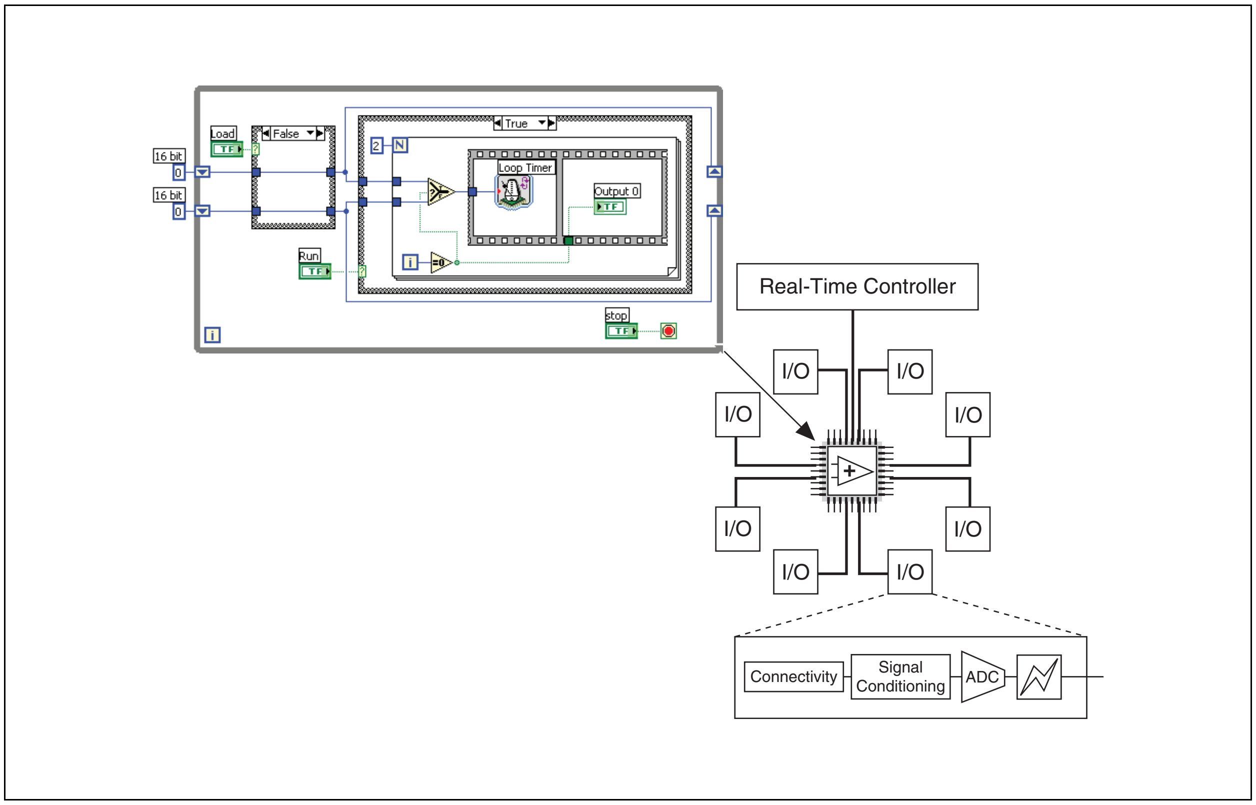
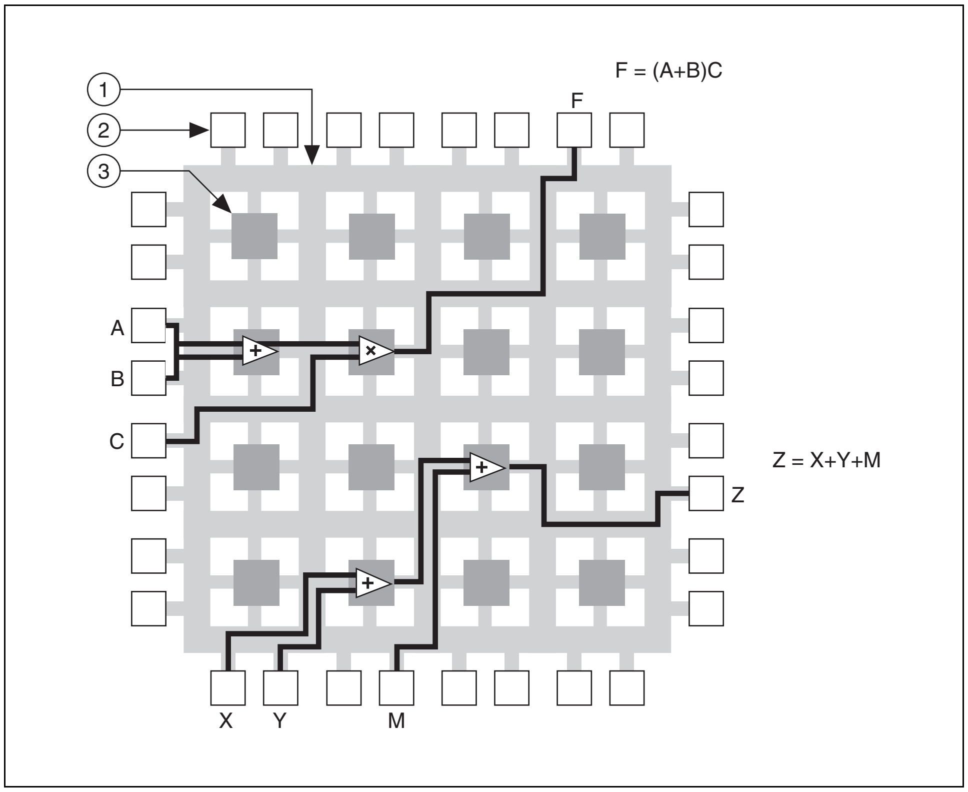


Figure 3-4. LabVIEW FPGA Module Defines FPGA Logic

With a LabVIEW FPGA system, you can implement synchronous or asynchronous parallel tasks in hardware to process and generate synchronized analog and digital signals rapidly and deterministically.

A primary benefit of running your LabVIEW code on an FPGA is that you can achieve true, simultaneous, parallel processing. There is no operating system on the module that must divide CPU time between several tasks. Figure 3-5 shows simultaneous parallel implementation of two calculations, $F = (A + B) \times C$ and $Z = (X + Y) + M$ in separate gates on an FPGA.



1	Interconnect Resources	2	I/O Cells	3	Logic Blocks

Figure 3-5. Parallel Implementation Using Separate Gates on an FPGA

The FPGA logic provides timing, triggering, processing, and custom I/O measurements. Each fixed I/O resource used by the application uses a small portion of the FPGA logic that controls the fixed I/O resource. The bus interface also uses a small portion of the FPGA logic to provide software access to the device. The remaining FPGA logic is available for higher level functions such as timing, triggering, and counting. The amount of FPGA space your application requires depends on your need for I/O and logic algorithms. Figure 3-6 shows the block diagram for the FPGA communications with I/O modules through the PCI bus.

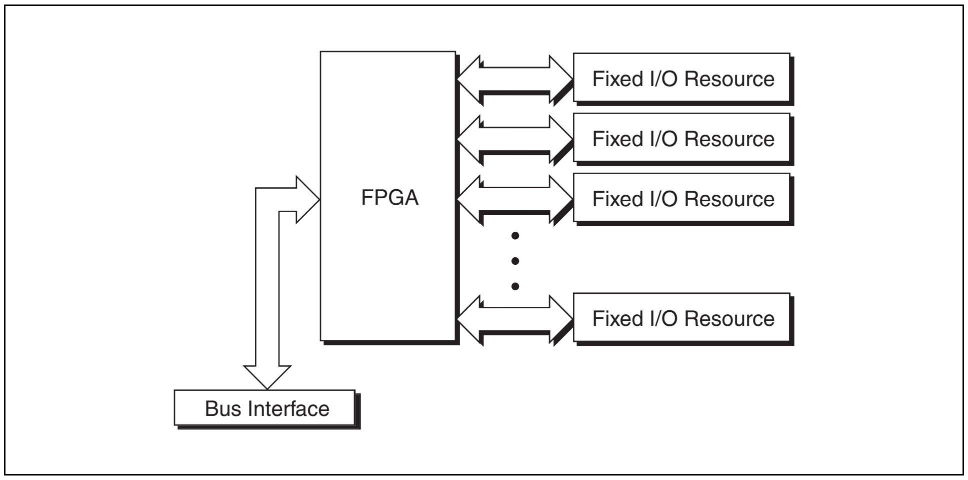


Figure 3-6. FPGA Communications Block Diagram

With the embedded RIO FPGA hardware, you can implement multiloop analog PID control systems at loop rates exceeding 100 kS/s. You can implement digital control systems at loop rates of 1 MS/s or more depending on the target. It is also possible to evaluate multiple rungs of Boolean logic using single-cycle Timed Loops at 200 MHz or more depending on the target and clock configuration. Because of the parallel nature of the RIO FPGA core, adding additional computation does not necessarily reduce the speed of the FPGA application.

C. Developing the FPGA VI

The FPGA contains the I/O portion of your application, as well as any hardware-based timing and triggering and low-level signal processing. LabVIEW FPGA Module applications range from a single FPGA VI running on an FPGA target to larger LabVIEW solutions that include multiple FPGA targets, one or more RT targets, and VIs running on Windows. In any case, you must create the FPGA VI that runs on the FPGA target.

With the LabVIEW FPGA Module you can use high-level graphical dataflow programming to create a highly optimized gate-array implementation of your analog or digital control logic. You can use normal LabVIEW programming techniques to develop your FPGA application.

You can download and run only one top-level FPGA VI at a time on a single FPGA target. If you download a second top-level VI to the FPGA target, the second VI overwrites the first VI.

When you add an FPGA VI under a hardware target, such as a CompactRIO chassis, the LabVIEW Functions palette adapts to contain only the VIs and functions that are designed to work on the target FPGA. The primary programming difference, compared to traditional LabVIEW, is that FPGA devices use integer math and fixed-point math instead of floating-point math. Also, there is no notion of multithreading or priorities because each loop executes in independent dedicated hardware and does not have to share resources—in effect, each loop executes in parallel at time-critical priority. Refer to the *LabVIEW Real-Time 1* course for more information about threads and priorities.

You start building the FPGA VI from the Project Explorer. Right-click the FPGA target and select **New»VI** as shown in Figure 3-7. Save the VI in an appropriate folder and save the project.

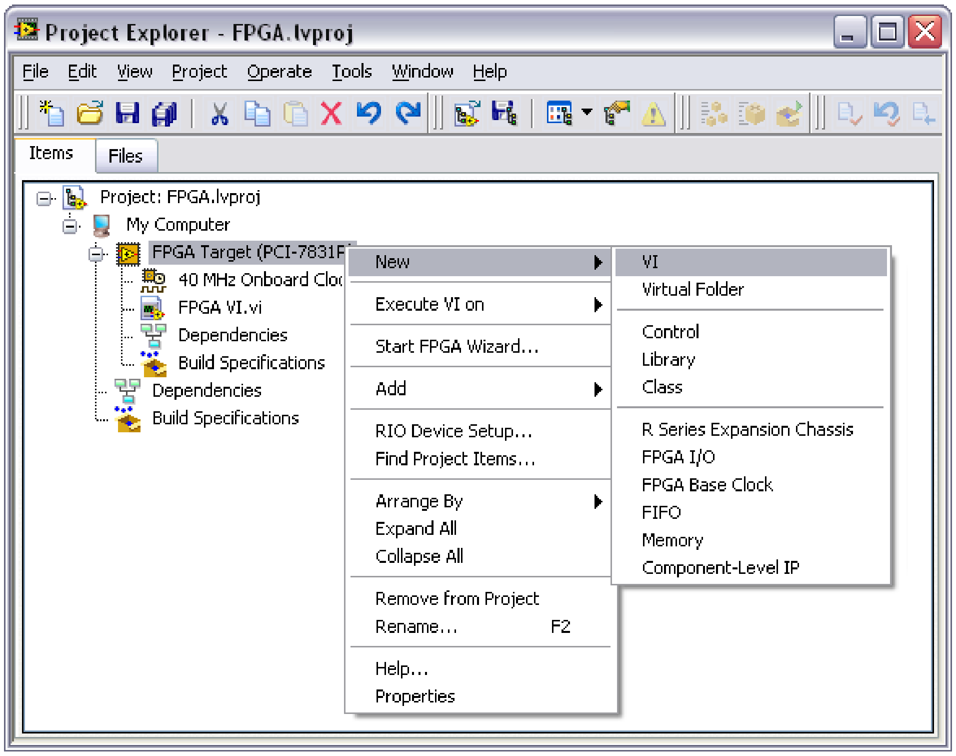


Figure 3-7. Add a VI to the FPGA Target

The FPGA VI resides under the FPGA target. The RT host VI that runs on the CompactRIO controller resides under the CompactRIO target. The host VI running on Windows resides under the My Computer target.

FPGA Functions Palettes

Each item under the FPGA target in the Project Explorer window contains FPGA target-specific information and functionality. When you select an item under an FPGA target in the Project Explorer window, LabVIEW displays only options available for the specific FPGA target. For example, if you select an FPGA VI under an FPGA target in the Project Explorer window and view the block diagram, LabVIEW displays only the subpalettes, VIs, and functions on the Functions palette that the FPGA target supports.

The FPGA Functions palette has many similar items to those on the core LabVIEW Functions palette. One difference is the FPGA I/O palette, which enables you to perform input and output operations on FPGA targets. Refer to Lesson 4, *FPGA I/O*, for more information about the use of this palette.

There are also differences in the Memory & FIFO palette. Refer to Lesson 6, *Data Sharing on FPGA*, for more information about using this palette.

Another difference is the FPGA Math & Analysis palette. Table 3-1 describes the objects in the FPGA Math & Analysis palette.

Table 3-1. FPGA Math & Analysis Palette Objects

Palette Object	Description
Control VIs	Use the Control VIs in FPGA VIs to create control applications for FPGA targets.
Generation VIs	Use the Generation VIs in FPGA VIs to generate signals.
High Throughput Math Functions	Use the High Throughput Math functions to achieve high throughput rates when performing fixed-point math and analysis on FPGA targets.
Utilities VIs	Use the Utilities VIs in FPGA VIs to perform various tasks such as detecting state changes of Boolean inputs, detecting zero crossings, delaying the input value, limiting the valid range of a signal, and performing linear interpolation.
Analog Period Measurement	Calculates the period of an evenly sampled periodic signal using threshold crossing detection.
Butterworth Filter	Filters one or more input signals using a lowpass or highpass IIR Butterworth filter.
DC and RMS Measurements	Calculates the DC (Mean) and/or RMS values of an input signal. You also can use this VI to calculate the intermediate sum, mean square, or square sum values in order to save FPGA resources.
FFT	Computes the Fast Fourier Transform (FFT) point by point.
Mean, Variance, and Standard Deviation	Calculates the mean, variance, and/or standard deviation of an input signal.
Notch Filter	Attenuates a specific frequency band in one or more input signals using a second order IIR notch filter.

Table 3-1. FPGA Math & Analysis Palette Objects (Continued)

Palette Object	Description
Rational Resampler	Provides a rational resampling filter, which updates the input sample rate by an L/M factor where L is an interpolation factor and M is a decimation factor.
Scaled Window	Minimizes spectral leakage associated with truncated waveforms. This Express VI scales the windowed time-domain signal so that when a LabVIEW object computes the power or amplitude spectrum of the windowed waveform, all windows provide the same level within the accuracy constraints of the output wavelength.

For VIs running on Windows or real-time, most math and analysis functions can be performed on a collection of data points, e.g. an array of data. In contrast, the functions of the FPGA Math & Analysis palette are all performed on a point-by-point basis. Point-by-point analysis is a method of continuous data analysis in which analysis occurs for each data point, point by point.

Creating an FPGA VI Front Panel

You create a front panel for FPGA applications, similar to the one shown in Figure 3-8.

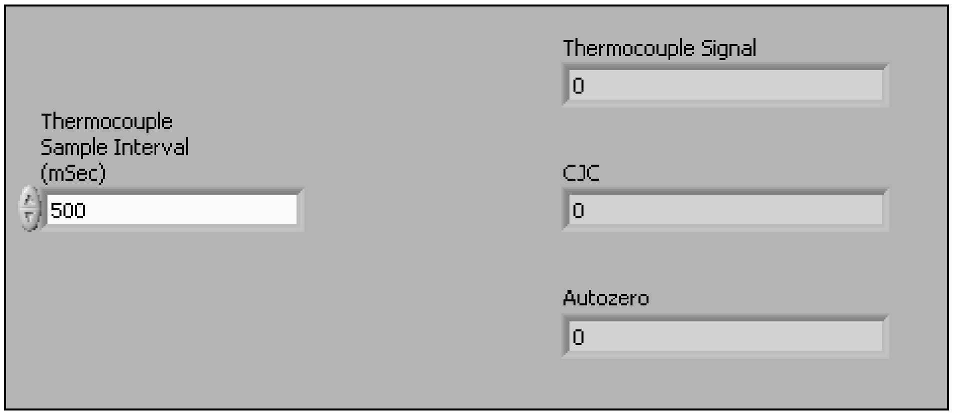


Figure 3-8. Temperature Monitor FPGA VI Front Panel

It is a good idea to set default values for controls, such as those shown Figure 3-8, because the FPGA VI is not controlled from a monitor display, but rather through the host VI. Remember that a host VI serves as the user interface, but controls and indicators take a lot of room on the FPGA and the FPGA has limited memory. So, keep the controls and indicators on the front panel of an FPGA VI simple to conserve space and enhance performance. You might add some temporary controls and indicators for testing that you remove later.

D. Interactive Front Panel Communication

You use Interactive Front Panel Communication to communicate between the host and the VI running on the FPGA, as shown in Figure 3-9. You use Programmatic FPGA Interface Communication to programmatically control and communicate with FPGA VIs from host VIs. You learn alternate communication processes later in this course.

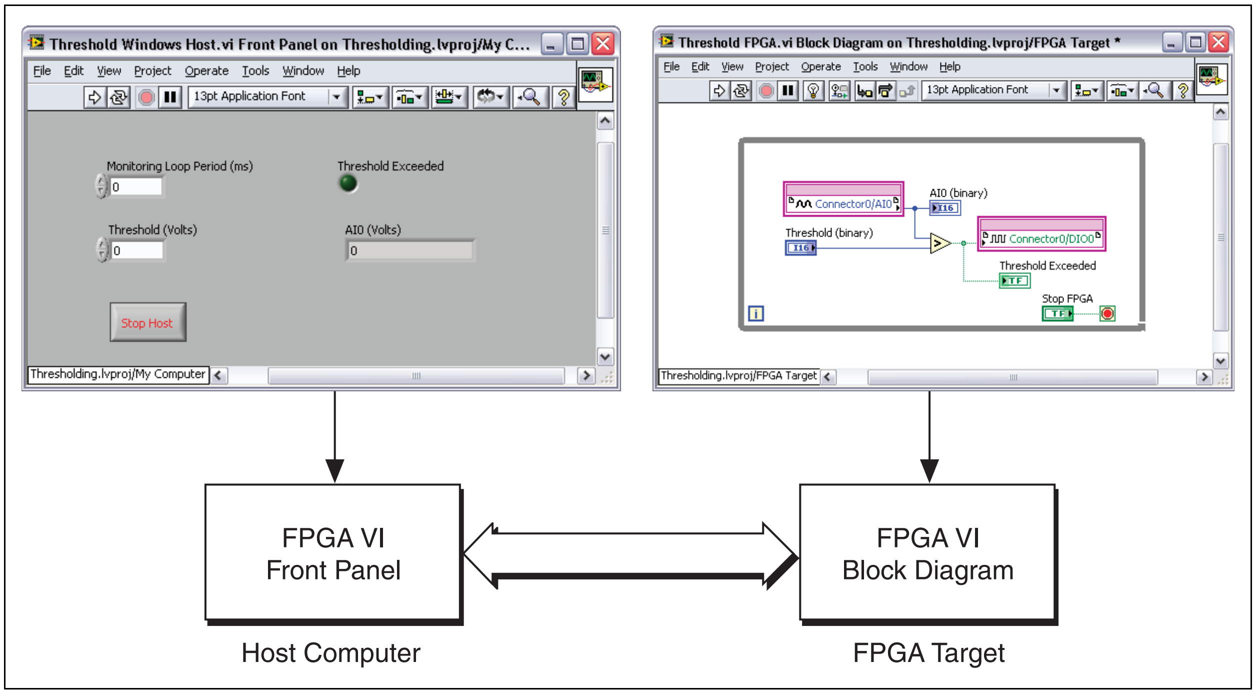

Threshold Windows Host.vi Front Panel on Thresholding.lvproj/My C...
Monitoring Loop Period (ms)
Threshold Exceeded
Threshold (Volts)
AI0 (Volts)
Stop Host
Thresholding.lvproj/My Computer
Threshold FPGA.vi Block Diagram on Thresholding.lvproj/FPGA Target *
Connector0/AI0
AI0 (binary)
Threshold (binary)
Connector0/DIO0
Threshold Exceeded
Stop FPGA
Thresholding.lvproj/FPGA Target
FPGA VI Front Panel
FPGA VI Block Diagram
Host Computer
FPGA Target

Figure 3-9. Interactive Front Panel Communication

Because there is no monitor connected to the FPGA, you must view the output on and provide input from a host computer. You can use Interactive Front Panel Communication to communicate with an FPGA VI running on an FPGA target with no additional programming. With Interactive Front Panel Communication, the host computer displays the FPGA VI front panel window and the FPGA target executes the FPGA VI block diagram.

The LabVIEW front panel window communicates with the FPGA target block diagram to exchange the state of the controls and indicators. You can communicate with an FPGA target located on the host computer or with an FPGA target located on a remote system. As the FPGA target block diagram continues to run, the host computer updates values on the FPGA VI front panel window as often as possible. The execution rate of the FPGA VI is not affected by updates to the host computer controls and indicators.

Note The front panel data you receive during Interactive Front Panel Communication is not deterministic.

Use Interactive Front Panel Communication between the FPGA target and the host computer to control and test VIs running on the FPGA target. After downloading and running the FPGA VI, keep LabVIEW open on the host computer to display and interact with the front panel window of the FPGA VI.

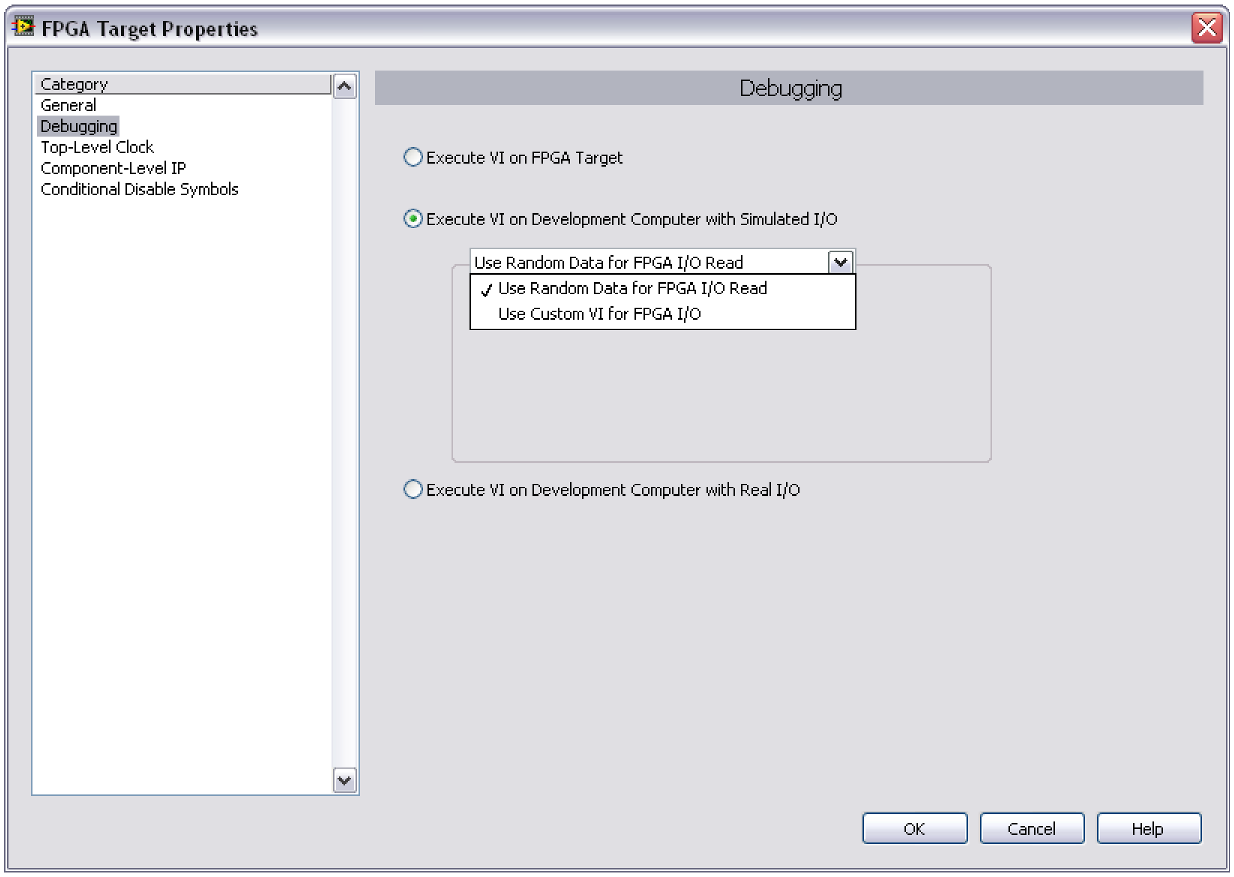
During Interactive Front Panel Communication, you cannot use LabVIEW debugging tools, so test the FPGA VI on the development computer, or add temporary controls and indicators for debugging and remove them after testing.

E. Selecting an Execution Mode

After developing your FPGA VI, you must determine where you want the FPGA VI to execute. You can select to execute the FPGA VI on the development computer in order to test it, or you can choose to execute the VI on the FPGA target, which will require compiling of code.

There are two ways to select the execution mode:

- Right-click the FPGA target in the Project Explorer window, select **Execute VI on** and select the mode that you wish to use.
- Right-click the FPGA target in the Project Explorer window and select **Properties** from the shortcut menu to display the FPGA Target Properties dialog box. Select **Debugging** from the **Category** list to display the Debugging page, as shown in Figure 3-10.


FPGA Target Properties

Category
General
Debugging
Top-Level Clock
Component-Level IP
Conditional Disable Symbols

Debugging

Execute VI on FPGA Target [✓ Execute VI on Development Computer with Simulated I/O] Execute VI on Development Computer with Real I/O

Use Random Data for FPGA I/O Read
✓ Use Random Data for FPGA I/O Read
Use Custom VI for FPGA I/O

Figure 3-10. Selecting an Execution Mode

There are three options for the execution mode of the FPGA VI.

- **Execute VI on FPGA Target**—Executes the FPGA VI on the FPGA target.
- **Execute VI on Development Computer with Simulated I/O**—Specifies whether to **Use Random Data for FPGA I/O Read** or **Use Custom VI for FPGA I/O** when executing the FPGA VI on a development computer.
- **Execute VI on Development Computer with Real I/O**—Executes the FPGA VI on the development computer using I/O from the FPGA target. Some FPGA targets do not support this option. Some FPGA targets that support this option do not support all I/O resources.

If you choose to execute the VI on the development computer using simulated I/O and you specify that you want to use a custom VI for FPGA I/O, then you can choose the path to the custom VI for FPGA I/O. You also have the option to create a new custom VI from a template. LabVIEW calls the custom VI whenever FPGA I/O Nodes, FPGA I/O Property Nodes, or FPGA I/O Method Nodes execute on the block diagram of the FPGA VI.

Testing on the Development Computer

Because compiling LabVIEW code to run on the FPGA chip may take a few minutes to several hours, LabVIEW has the ability to run the code on the development computer to verify the logic before you initiate the compile process. FPGA code running on the development computer accesses I/O from the device and executes the VI logic on the development computer, where traditional LabVIEW debugging tools, such as execution highlighting, probes, and breakpoints, are available.

When you run an FPGA VI on the development computer, LabVIEW generates random data for the inputs or downloads a pre-compiled VI to the FPGA target to provide I/O. If you use I/O from the FPGA target, LabVIEW communicates with the VI on the FPGA target while both VIs run. The availability of the development computer to simulate I/O varies by FPGA target. Refer to the specific FPGA target hardware documentation for information about development computer support.

You cannot test certain behavior, such as timing and determinism, with a development computer because the FPGA VI runs on the host computer instead of the FPGA, which has different timing constraints and abilities.

As FPGA VIs get larger and more complicated, debugging designs on the FPGA chip becomes less efficient because of the time necessary for compiling and downloading to the target. Along with other debugging techniques, cycle-accurate simulation can help you test the timing behavior of FPGA VI components. Cycle-accurate simulation means that timing is precise but the simulation might not use the exact hardware model to implement it. For more information on cycle-accurate simulation, refer to the *Debugging FPGA VIs Using Cycle-Accurate Simulation* topic of the *LabVIEW Help*.

Note Not all targets support simulation. If the FPGA target supports simulation, you will have the option to create a simulation export from the Build Specifications shortcut menu.

F. Compiling the FPGA VI

The LabVIEW FPGA Module compiles VIs to FPGA hardware using an automatic, multi-step process. Behind the scenes, the VI is translated to text-based VHDL code. Then industry-standard Xilinx ISE compiler tools optimize, reduce, and synthesize the VHDL code into a hardware circuit realization of the LabVIEW design. This process also applies timing constraints to the design and tries to achieve an efficient use of FPGA resources (sometimes called fabric).

During the FPGA compilation, the VI is optimized to reduce digital logic and create an optimal implementation of the LabVIEW application. The end result is a bitstream file that contains the gate array configuration information. When the application runs, the bitstream loads into the FPGA chip and reconfigures the gate array logic. The bitstream also can download into nonvolatile flash memory, and load instantaneously to the FPGA when power is applied to the target.

When you run an FPGA VI, the LabVIEW diagram is converted to intermediate files that are sent to the compile server where they are compiled for the FPGA, as shown in Figure 3-11. The server returns the FPGA bitstream to LabVIEW where it is stored in the VI. The compiled bitstream is unique to each type of CompactRIO FPGA and backplane. You can compile a VI to different CompactRIO targets and the different bitstreams are stored in the same VI. The bitstream downloads automatically when you a compile the VI by clicking the **Run** button in LabVIEW.

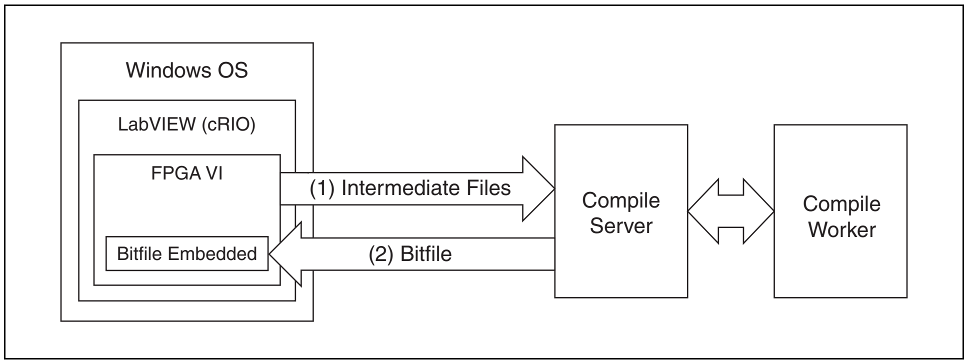


Figure 3-11. Compile Process

Caution Do *not* make changes to the FPGA VI while the FPGA VI is compiling. If you do make changes, you must recompile the FPGA VI.

Working with Build Specifications

You must create a build specification before you can compile the FPGA VI into an FPGA application. If you click the **Run** button without first making a build specification, LabVIEW automatically creates and specifies a default build specification for the corresponding VI. LabVIEW uses the build options you set to generate the HDL and bitfile from the block diagram.

Right-click a build specification under an FPGA target in the Project Explorer window and select from the following options. The options can vary depending on the hardware target.

- **Build**—Generates intermediate files only if the FPGA VI or build specification changed since the last time you compiled the VI. Then this command compiles the VI.
- **Rebuild**—Generates intermediate files regardless of whether the FPGA VI or build specification changed, and then compiles the VI.
- **Estimate Resource Usage**—Uses Xilinx tools to estimate FPGA resource usages without compiling. If the generated code signature is not current, this command first generates intermediate files and then estimates resource usage.
- **Check Signature**—Determines whether the bitfile is current. If the bitfile does not exist, the command checks the generated code signature.
- **Generate Intermediate Files**—Generates intermediate files without compiling the VI. Generating intermediate files catches certain code generations errors.
- **Display Compilation Results**—Displays the Compilation Status window. You must build the build specification once to display the Compilation Status window.

Specifying a Default Build Specification

The default build specification is the build specification that the Run button uses to compile and run an FPGA application. As previously mentioned, if you click the **Run** button without first making a build specification, LabVIEW automatically creates and specifies a default build specification for the corresponding VI.

To specify the default build specification, place a checkmark in the **Set as default build specification** checkbox on the Source Files page of the Compilation Properties dialog box.

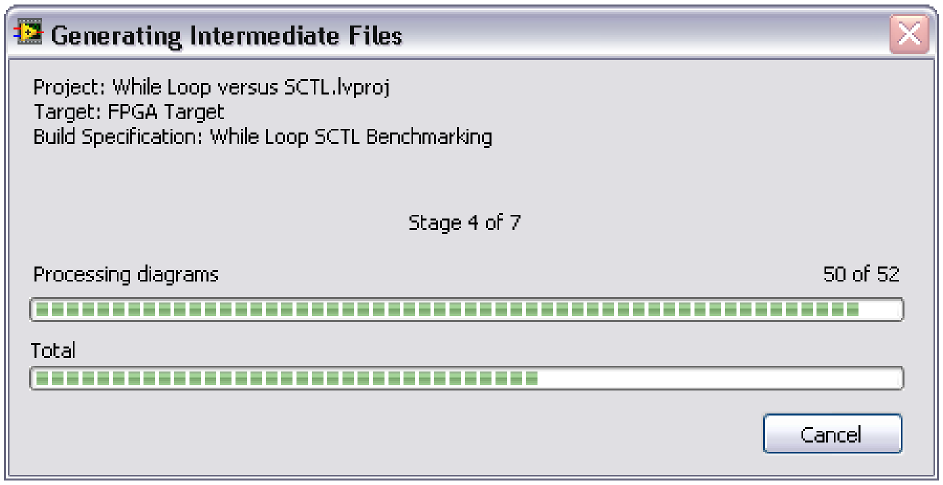
The Compilation Process

The compilation process can be broken down into the following steps:

1. Generate intermediate files—Convert the VI code into intermediate files (HDL code) to send to the FPGA compile server.
2. Resource Estimation—Estimate FPGA resource usages without compiling. If the generated code signature is not current, this command first generates intermediate files and then estimates resource usage.
3. HDL compilation, analysis and synthesis—Transform HDL code into digital logic elements.

4. Mapping—Divides the application between the physical building blocks on the FPGA.
5. Placing and routing—Assigns the logic to physical building blocks on the FPGA and routes the connections between the logic blocks to meet the space or timing constraints of the compilation.
6. Generating bitstream—Creates binary data that LabVIEW saves inside a bitfile.

When you initiate the compilation, LabVIEW displays the window shown in Figure 3-12.


Generating Intermediate Files

Project: While Loop versus SCTL.lvproj
Target: FPGA Target
Build Specification: While Loop SCTL Benchmarking

Stage 4 of 7

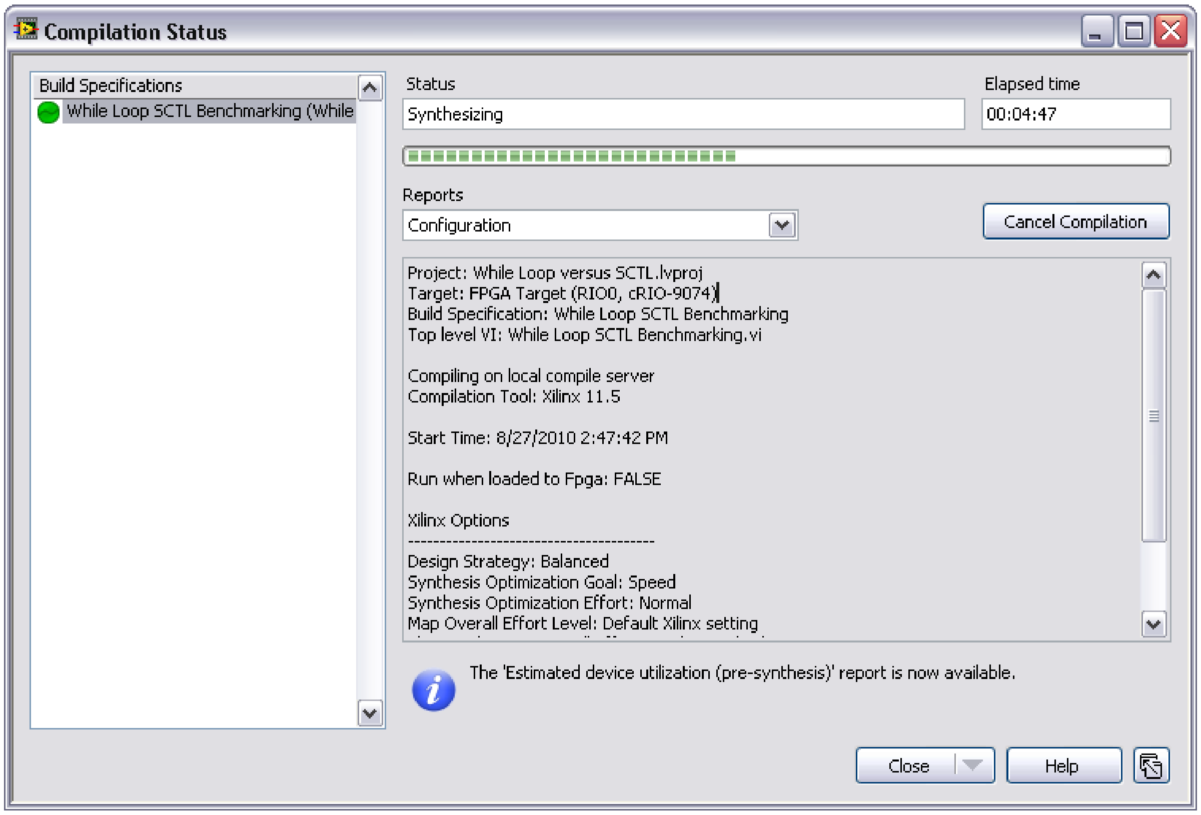
Processing diagrams 50 of 52

Total

Figure 3-12. Generating Intermediate Files Window

Once the intermediate files are generated, the compilation server uses Xilinx to convert the intermediate files into a bitstream. The Xilinx ISE compile tools optimize, reduce, and synthesize the VHDL code into a hardware circuit realization of the LabVIEW code. The end result is a bitstream file that is loaded into the FPGA chip to configure the gate array logic. During the Mapping and Synthesis process, you can view and estimate of whether you application code will fit on the FPGA.

After the intermediate files are generated, the window shown in Figure 3-13 appears. This window can be used to view status information while the FPGA VI is compiling. You also can use this window to disconnect from the compile server or cancel the compilation.



Synthesizing

00:04:47

Configuration

```
Project: While Loop versus SCTL.lvproj
Target: FPGA Target (RIO0, cRIO-9074)
Build Specification: While Loop SCTL Benchmarking
Top level VI: While Loop SCTL Benchmarking.vi

Compiling on local compile server
Compilation Tool: Xilinx 11.5

Start Time: 8/27/2010 2:47:42 PM

Run when loaded to Fpga: FALSE

Xilinx Options
---------------------------------------
Design Strategy: Balanced
Synthesis Optimization Goal: Speed
Synthesis Optimization Effort: Normal
Map Overall Effort Level: Default Xilinx setting
```

The 'Estimated device utilization (pre-synthesis)' report is now available.

Figure 3-13. Compile Status Window

The Compile Status Window contains the following components:

- **Build Specifications**—Lists the build specifications that are compiling or have compiled during this session of the Compilation Status window.
- **Status**—Displays the progress of the FPGA VI compilation.
- **Last Update Time**—Displays the last time the compile server updated the information.
- **Reports**—Specifies the type of compiler information you want to view. Not all reports are available at the beginning of the compilation. A message appears in the Compilation Status window when reports become available to view.
- **Save**—Saves the final compilation status report to a file. The Save button is available only for the Xilinx log report and only when the compilation is complete.

- **Investigate Timing Violation**—Displays the Timing Violation Analysis window. This button is available only if a timing violation occurs.
- **Cancel Compilation**—Cancels the compilation of the build specification you select in the Build Specifications list.
- **Close**—Closes the Compilation Status window. Use the pull-down menu on this button to close the window with the following options.
  - **Close Window (default)**—Closes the window and continues communicating with the compile server.
  - **Disconnect All**—Disconnects LabVIEW from the compile server. Any compilation jobs running will continue, but LabVIEW will not receive any updates. Disconnect from the compile server when you want to shutdown LabVIEW. You do not need to disconnect from the compile server to perform other operations in LabVIEW while the FPGA VI is compiling.
  - **Cancel All Compilations**—Cancels the current compilation jobs.
- **Switch to mini view**—Shrinks the Compilation Status window to show only the Build Specifications list.
- **Switch to full view**—Expands the Compilation Status window to its full size.

Compilation Reports

As the compilation progresses, different reports become available in the **Reports** selector of the Compile Status window.

- **Configuration**—This report displays project information and the Xilinx compiler configuration you specified in the Xilinx Options page of the build specification.
- **Estimated device utilization (synthesis)**—This report is available after LabVIEW estimates FPGA resource usage using the Xilinx tools. This report includes the following information:
  - **Device Utilization**—Indicates the FPGA element, such as slices, flip-flops, LUTs, and blocks of RAM.
  - **Used**—Indicates how many of the FPGA element the compiled FPGA VI uses.
  - **Total**—Indicates the total number of FPGA elements in the FPGA.
  - **Percent**—Indicates the percentage of the FPGA elements that the FPGA application uses. If **Percent** is greater than 100, a warning message alerts you that the estimated device utilization exceeds 100 percent. Depending on the FPGA VI and hardware, Xilinx still may be able to fit everything on the FPGA. However, you may want to optimize the FPGA VI.

- **Estimated Timing (Map)**—This report is available after the compile server completes the mapping step of the compilation process. This report contains a summary of the FPGA clocks, as estimated during the mapping of the FPGA VI. This report includes the following information:
  - **Clocks**—Indicates the FPGA clocks.
  - **Requested (MHz)**—Indicates the clock rate, in megahertz, at which the FPGA VI or FPGA VI component must be able to run. Some FPGA VI components, such as Timed Loops, are visible on the block diagram, while others, such as CLIP, are not. If **Requested (MHz)** is greater than **Maximum (MHz)**, a warning message alerts you that the VI does not meet timing constraints. Depending on the FPGA VI and hardware, Xilinx may still be able to compile the FPGA VI such that it meets timing constraints. However, you may want to stop the compilation and optimize the FPGA VI.
  - **Maximum (MHz)**—Indicates the theoretical maximum compilation rate, in megahertz, for the FPGA VI or FPGA VI component.
- **Final Device Utilization (Map)**—This report is available after the compile server completes the mapping step of the compilation process. This report contains a summary of the FPGA utilization, including the following information:
  - **Device Utilization**—Indicates the FPGA element, such as slices, flip-flops, LUTs, and blocks of RAM.
  - **Used**—Indicates how many of the FPGA element the compiled FPGA VI uses.
  - **Total**—Indicates the total number of FPGA elements in the FPGA.
  - **Percent**—Indicates the percentage of the FPGA elements that the FPGA application uses. If **Percent** is greater than 100, the compilation failed. You must optimize the FPGA VI for size.
- **Final Timing (Place and Route)**—This report is available after the compile server completes the routing step of the compilation process. This report contains a summary of the FPGA clocks, including the following information:
  - **Clocks**—Indicates the FPGA clocks.
  - **Requested (MHz)**—Indicates the clock rate, in megahertz, at which the FPGA VI or FPGA VI component must be able to run. Some FPGA VI components, such as Timed Loops, are visible on the block diagram, while others, such as CLIP, are not. If **Requested (MHz)** is greater than **Maximum (MHz)**, the compilation fails. Click the **Investigate Timing Violation** button to analyze the timing violations.

   - **Maximum (MHz)**—Indicates the theoretical maximum compilation rate, in megahertz, for the FPGA VI or FPGA VI component.
- **Xilinx Log**—This log file is available only after the compilation is complete. This report includes the `XilinxLog.txt` and `.twr` files and is available only after the compilation is complete. The `XilinxLog.txt` file includes Xilinx-specific details about the compilation process. If you are familiar with Xilinx tools, you might be able to use this file to troubleshoot compilation failures. Click the **Save** button to save this information to a file.

For small applications, the compiler does not optimize as completely as for larger applications. As the FPGA reaches 90 percent optimization, the compiler performs heavy optimization.

After you close the Compilation Status window, the VI runs on the FPGA and communicates with the host PC through Interactive Front Panel Communication. Data for the front panel objects are communicated over the bus to the host several times per second, typically about 10 S/s.

Compiling an FPGA VI Remotely

You can compile an FPGA VI on a computer other than the development computer. You might want to do this if the development computer is slow or does not have enough memory to compile the VI for the FPGA device. By default, LabVIEW assumes you are compiling FPGA VIs on the local computer.

Installing the Xilinx Compilation Tools Remotely

To install the necessary software on the compile server machine, first insert your LabVIEW Platform DVD and run the installer. When prompted to choose which components to install, select only **FPGA»Compilation Tools for Virtex-II FPGA Devices** or **FPGA»Compilation Tools for Devices other than Virtex II**, depending on your device.

Configure LabVIEW to Compile on a Remote Computer

Complete the following steps to configure LabVIEW to compile an FPGA VI on a remote computer:

1. Install the necessary Xilinx compilation tools on the remote computer.
2. On the remote computer, select **Start»All Programs»National Instruments»FPGA»FPGA Compile Server Configuration**.
3. Place a checkmark in the **Allow users to connect remotely to this compile server** checkbox and click the **OK** button.

4. Select **Start»All Programs»National Instruments»FPGA»FPGA Compile Worker** to launch the compile worker.

Note You must launch the compile worker from the Start menu of the computer on which it is executing. You cannot launch the compile worker remotely.

5. Leave the compile worker running on the remote computer.
6. On the local computer, open the FPGA project that contains the VI you want to compile.
7. Select **Tools»FPGA Module Options** to display the FPGA Module Options dialog box.
8. Select **Connect to a compile server** in the Compile Server section.
9. Type the name or IP address of the remote computer running the compile worker in the **Host name** text box.
10. (Optional) Type the user name and password necessary to log into the remote compile server. By default, the User name is `admin` and the Password text box is empty.
11. Click the **OK** button.
12. Compile the FPGA VI. If LabVIEW is unable to contact the compile server, LabVIEW displays an error in the Compilation Status window. Otherwise, LabVIEW compiles the FPGA VI on the remote computer every time you compile an FPGA VI.

G. Basic Optimizations

If the final timing report indicates that the requested clock rates exceeds the maximum rate, or the final device utilization report indicates that more than 100 percent of any device has been utilized, then the compilation will fail. In either case, it becomes necessary to revise your FPGA VI and optimize your code to address the cause of failure.

Basic FPGA optimization techniques are typically easy to implement, require no major changes in the code architecture, and often are FPGA programming best-practices. The basic optimizations covered in this section primarily affect FPGA code size.

Tip For more information about optimizing for FPGA, refer to the *Optimizing FPGA VIs for Speed and Size* topic of the *FPGA Help*.

Limit Front Panel Objects

Front panel objects consume a significant amount of space on the FPGA. Not only is space required to store the data itself, but a considerable amount of FPGA logic is required to implement the communication between the front panel object and the host VI.

When you transfer data across the bus between the FPGA and host, it must be broken down into 32-bit packets. This limitation is due to the number of data transfer lines on the bus. Therefore, if you have a front panel element that uses more than 32-bits, that front panel element must be broken down into several 32-bit chunks and be passed along the bus piece wise. This breaking down of the data reduces overall transfer speeds.

Use the following guidelines to optimize the FPGA code:

- Limit the use of arrays
- Use the bitpacking technique as an alternative to small arrays

These techniques are necessary only for front panel objects used in the top-level FPGA VI; front panel objects in subVIs do not consume space on the FPGA.

Avoid Front Panel Arrays

Arrays and clusters that are greater than 32-bits in size require an extra copy on the FPGA to guarantee all the data is read and they consume a significant amount of space on the FPGA. Array elements are transferred to and from the FPGA individually, so limiting the use of arrays decreases the logic cells required to implement the data transfer.

If you use an array control on the front panel, you must limit the array size. You can select the size of the array by right-clicking the control and choosing the appropriate user menu selection to add or remove dimensions.

The best way to transfer large sets of data from the FPGA to the host is with DMA (Direct Memory Access) FIFOs. Refer to Lesson 9, *DMA Data Transfers*, for more information about DMA FIFOs.

Arrays consume significant space on the FPGA because each bit in the array uses a flip-flop on the FPGA.

When you wire an array as an input to an FPGA VI or function, the FPGA compiler creates the equivalent of a For Loop to process each element of the array in sequence. If you wire a cluster as an input to an FPGA VI or function, the FPGA compiler creates parallel logic for each element of the cluster. The relationship between arrays and clusters is recursive such that if you wire a cluster of arrays as an input, the arrays are processed in parallel and the array elements are processed sequentially.

Note Performing operations on arrays can limit the maximum top-level or derived clock rate. To maximize the FPGA clock rate, process single data points instead of arrays.

One of the most common uses of array controls and constants in the FPGA VI is to store large sets of waveform information. You can replace large front panel array controls with initialized memories.

Bitpack Boolean Logic

Bitpacking is a technique that combines many small pieces of data into a larger piece of data. This technique works because the communication between the FPGA target and real-time host is in 32-bit words. For example, you can store four 8-bit integers as one 32-bit integer. Instead of four separate buses to transfer the elements of data, only one bus is required.

Another example of bitpacking is to convert Boolean controls to a Boolean array. For example, you can convert eight Boolean controls into a single 8-bit unsigned integer then use a Number to Boolean Array function to index it into individual components, as shown Figure 3-14.

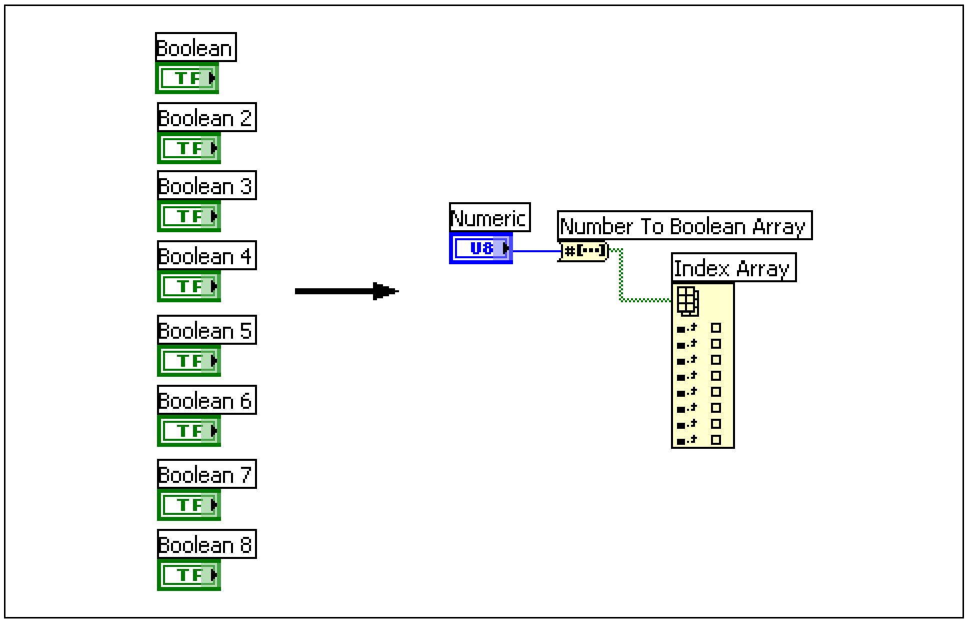


Figure 3-14. Convert Boolean Controls to a Boolean Array

Global Variables

You can use global variables scoped to the FPGA to optimize code for size. Using global variables in subVIs is especially effective if other VIs must access the data. If a piece of data is not global, you must use logic to route it to other VIs and each VI will require memory to store the data. To add a global variable to your project, select Global Variable from the Structures palette on the block diagram. Then, link the global variable to the control or indicator on your front panel. You can also replace controls with constants to further reduce FPGA usage.

Use Small Data Types

Always select the smallest data type possible when using controls and constants on the FPGA. For example, the default data type of the Index Array function is a 32-bit integer. However, if your array will have no more than 256 elements, represent the index as an unsigned 8-bit integer.

When using a Case structure, only very rarely would you need more than 256 cases. If you have fewer than 256 elements in your Case structure then use an 8-bit integer wired to the selector terminal instead of the default 32-bit integer value.

Similarly, if you are using the Loop Timer Express VI, Tick Count Express VI, or Wait Express VI, configure them to use the smallest Size of Internal Counter possible.

Eliminate Coercion Dots

When a coercion dot occurs, it means the compiler must determine the proper data type and it can lead to inefficient data conversion. Instead of leaving the coercion to chance, specify the desired data type and insert the appropriate conversion function when you see a coercion dot on your block diagram.

Avoid Large Functions

Some LabVIEW functions consume a significant amount of space on the FPGA, such as the Quotient & Remainder, Rotate 1D Array, and Scale By Power of 2 functions.

- **Quotient & Remainder**—This function consumes significant space on the FPGA. If you must divide by a power of two, consider using the Scale by Power of 2 function with a negative constant wired to the **n** input, as shown in Figure 3-15.

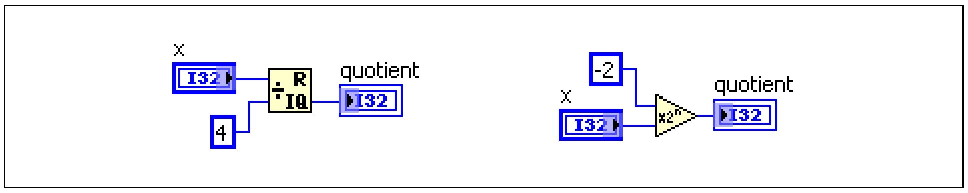


Figure 3-15. Replace Quotient & Remainder with Scale by Power of 2

Often, the numerator of the Quotient & Remainder function is tied to the iteration terminal and the remainder keeps a place within an array or look-up table. An alternative is to create code that increments on its own by using an Increment function tied to a shift register and setting the value back to zero when the count exceeds a specified value, as shown in Figure 3-16.

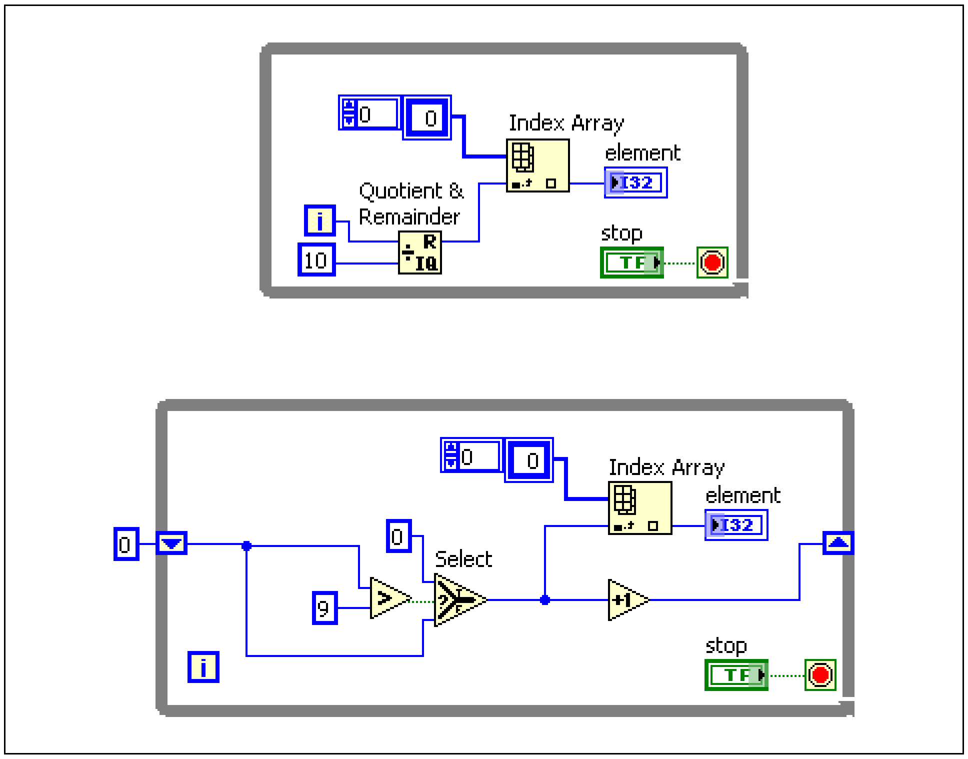


Figure 3-16. Replace Quotient & Remainder with Increment Function and Shift Register

- **Scale by Power of 2**—If you wire a control to the n input, this function can consume significant space on the FPGA. However, if you wire a constant to the n input, the function consumes no space on the FPGA. By wiring a negative constant to the input this function has the same effect as dividing by that power of 2.

Note You can optimize FPGA VIs by using constants instead of controls whenever possible. However, intermediary structures such as connector panes, loop tunnels, and shift registers can prevent LabVIEW from recognizing an input as a constant. For best results, wire the constant directly to the function where it is needed.

Self-Review: Quiz

1. You developed a VI and set the project to execute the VI on the FPGA target. You compile the code and run the VI. Which of the following statements is true?
   a. The block diagram and the front panel both execute on the FPGA.
   b. The block diagram executes on the FPGA and the front panel executes on the host computer.
   c. The block diagram executes on the host computer and the front panel executes on the FPGA.
   d. The block diagram and front panel both execute on the host computer.

2. Where should you first test of your FPGA VI's functionality?
   a. FPGA target
   b. Development computer

3. Which of the following is *not* the name of a report generated as part of the compilation process?
   a. Summary
   b. Final Device Utilization (map)
   c. Final Timing (place and route)
   d. Optimization

4. Which of the following does *not* describe a VI that is developed under the FPGA target and set to execute on the FPGA?
   a. The front panel of the FPGA VI executes on the FPGA target
   b. Non-sequential code in an FPGA VI executes in parallel
   c. FPGA VI executes independently of the host
   d. Only one top-level VI can run on the FPGA at a time

Self-Review: Quiz Answers

1. You developed a VI and set the project to execute the VI on the FPGA target. You compile the code and run the VI. Which of the following statements is true?
   a. The block diagram and the front panel both execute on the FPGA.
   b. The block diagram executes on the FPGA and the front panel executes on the host computer. This concept is known as interactive front panel communication. Values entered into the front panel controls are passed from the host computer to the compiled code that executes on the FPGA.
   c. The block diagram executes on the host computer and the front panel executes on the FPGA.
   d. The block diagram and front panel both execute on the host computer.

2. Where should you first test of your FPGA VI's functionality?
   a. FPGA target
   b. Development computer. Compilation is required before code can be executed on the FPGA target and that process can take a significant amount of time. Only after the basic code functionality is verified should it be compiled and tested on the FPGA target.

3. Which of the following is *not* the name of a report generated as part of the compilation process?
   a. Summary
   b. Final Device Utilization (map)
   c. Final Timing (place and route)
   d. Optimization. The reports that are generated can be used to determine where optimization should occur (size, timing, etc), but there is no report named Optimization.

4. Which of the following does *not* describe a VI that is developed under the FPGA target and set to execute on the FPGA?
   a. **The front panel of the FPGA VI executes on the FPGA target.** The Front Panel of the FPGA VI executes on the host computer. Only the block diagram is compiled and executed on the FPGA VI.
   b. Non-sequential code in an FPGA VI executes in parallel
   c. FPGA VI executes independently of the host
   d. Only one top-level VI can run on the FPGA at a time

Notes

Notes

4

FPGA I/O

In this lesson, you learn how to add FPGA I/O to your LabVIEW project and use it on the block diagram. You also learn about the differences between performing I/O on an R Series device and on a CompactRIO chassis and the differences between integer and fixed-point data. Using I/O Nodes, you learn how to access both analog and digital data.

Topics

A. Introduction

Inputs and outputs on FPGA targets allow you to connect the FPGA target to other devices, such as sensors and actuators. FPGA I/O resources are fixed elements of the FPGA targets that you use to transfer data among the different parts of the system. On some FPGA targets, FPGA I/O resources correspond to lines on front panel connectors, PXI backplanes, or Real-Time System Integration (RTSI) connectors. On other FPGA targets, FPGA I/O resources are nodes inside FPGAs that connect the part of the FPGA designed by National Instruments with the part of the FPGA you design. Each FPGA I/O resource has a specific type, such as digital or analog. An FPGA target might have multiple resources of the same or different types. You can create FPGA I/O items, determine the I/O resources on the FPGA target that you want to use, and then assign unique names to the I/O resources you use.

Note Refer to the specific FPGA target hardware documentation for information about supported features and I/O functionality on the FPGA target you use.

Several I/O example VIs are available in the NI Example Finder in **Toolkits and Modules»FPGA»CompactRIO»Basic IO** and **Toolkits and Modules»FPGA»CompactRIO»FPGA Fundamentals**.

Use the following terms when working with FPGA I/O Nodes:

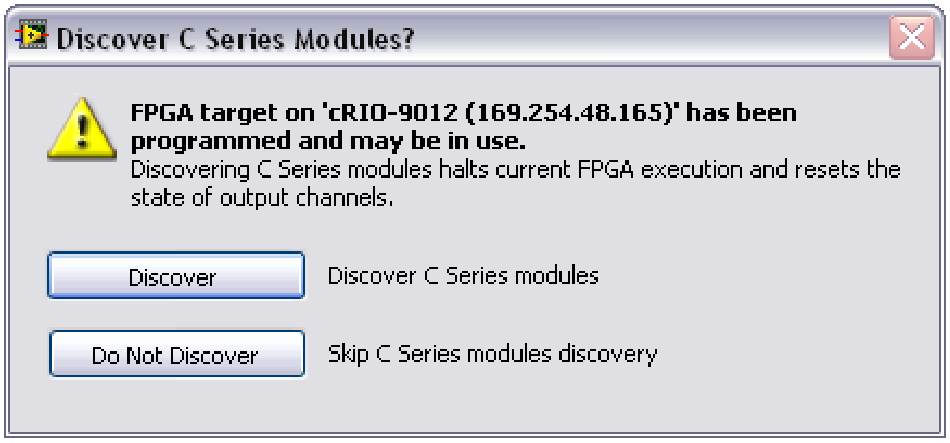
- **Terminal**—Represents a hardware connection on a CompactRIO module.
- **I/O Resource**—Specifies a logical representation in LabVIEW FPGA of a hardware terminal.
- **I/O Name**—Specifies a name assigned by the developer to a particular I/O resource that normally describes the function of the resource. Assigning a different I/O resource to an I/O name updates all instances of the I/O name within the project.

The FPGA VI configures the FPGA circuit. When the FPGA circuit is activated, it performs the I/O operations in hardware. For example, if you configure an FPGA I/O node to read a digital line, the FPGA I/O node reads the line and returns the result. Consequently the FPGA can react to the input with the speed and determinism available in the FPGA target hardware.

You can put inputs and outputs, analog and digital, all in the same node on the block diagram. You can use target-specific properties and methods on the FPGA I/O items with the FPGA I/O Property Node and the FPGA I/O Method Node, respectively.

B. Configuring FPGA I/O

In order to access I/O on the FPGA, you must first add I/O to your project. Depending on the FPGA target you are working with, there are different ways to add FPGA I/O to a LabVIEW project. If you are using a CompactRIO target, you can choose to detect modules when your add your chassis to the project. Autodetection of modules ensures that all FPGA I/O resources are added to the project. When you add the chassis to your project, the Discover C Series Modules? dialog box shown in Figure 4-1 may appear.


Discover C Series Modules?

FPGA target on 'cRIO-9012 (169.254.48.165)' has been programmed and may be in use.
Discovering C Series modules halts current FPGA execution and resets the state of output channels.

Figure 4-1. Discover C Series Modules? Dialog Box

Click **Discover** to automatically detect the modules and add I/O resources to the project. If you choose not to automatically detect modules, add them manually by right-clicking the FPGA target and selecting **New»C Series Modules**.

When using an R Series target, you can manually add I/O resources to your project to ensure that you add only the resources that you intend to use. This significantly reduces the number of items in the Project Explorer window. Right-click the FPGA target in the Project Explorer and select **New»FPGA I/O** to launch the New FPGA I/O dialog box, show in Figure 4-2. Select an I/O resource in the left pane of the New FPGA I/O dialog box and click the right arrow button to add the I/O resource to the project. Rename I/O resources by clicking the default name and entering a meaningful name for your application.

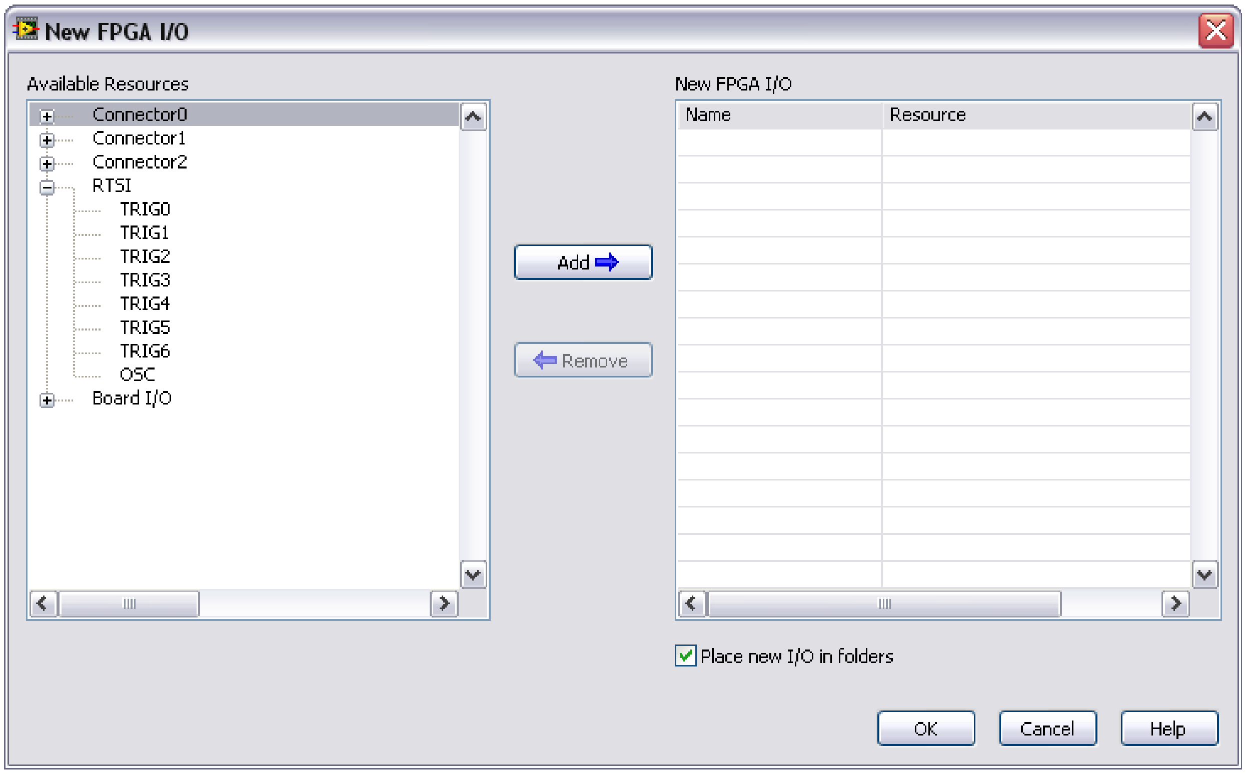


Figure 4-2. New FPGA I/O Dialog Box

After you add I/O resources to your project, you can access them in VIs developed under the FPGA target. You can access FPGA I/O resources by opening the FPGA I/O palette. Table 4-1 describes the objects available in that palette.

Table 4-1. FPGA I/O Palette Objects

Palette Object	Description
FPGA I/O Constant	Specifies an FPGA I/O item on the block diagram.
FPGA I/O Method	Invokes a method on an I/O item or hardware under and FPGA target in the Project Explorer window.
FPGA I/O Node	Performs specific FPGA I/O operations on FPGA targets.
FPGA I/O Property	Gets or sets one or more properties on an I/O item or hardware under an FPGA target in the Project Explorer window.

You place an FPGA I/O Node on the block diagram from the palette or by dragging an FPGA I/O item from the Project Explorer.

If you add the FPGA I/O Node from the Functions palette, you must configure it by right-clicking, selecting **Select FPGA I/O**, and making the appropriate choices for configuration. The Select FPGA I/O submenu displays the FPGA I/O items that appear in the Project Explorer tree. You can also click the FPGA I/O Node and use the shortcut menu to add new FPGA I/O items or select from FPGA I/O items you previously added to the project.

C. I/O Types

There are two basic FPGA I/O types in LabVIEW FPGA hardware: digital and analog. In addition to these basic two FPGA I/O types, there are also CompactRIO modules for motion control and CAN.

Digital

Digital lines are basic digital I/O. Digital lines are bi-directional on all R Series devices and some CompactRIO modules. Refer to the documentation for your specific device for more information about digital I/O functionality.

FPGA targets might organize digital I/O resources as individual lines or as groups of lines called ports. A digital line I/O uses the Boolean data type. Ports use a data type that is dependent on the target. Ports for most I/O are a group of 8 bits, but others can be 16 or 32 bits. One bit is used for each line. Some FPGA targets provide access to digital I/O resources as only lines or ports. Other FPGA targets allow you to access the same physical lines as individual lines and as ports.

You can use digital input and output resources to configure the I/O resource and control the direction of dataflow. If you use a digital resource to write an output signal, you must disable the output before you can use the same resource to read an input signal. Use the FPGA I/O Method Node with the Set Output Enable method to disable the output line.

If you use the FPGA I/O Node to write a digital output, the FPGA I/O Node writes the data and enables the terminal for output. You also can use the FPGA I/O Method Node with the Set Output Data method to write data without enabling the output. Use the FPGA I/O Method Node with the Set Output Enable method to enable the digital terminal, which allows the data to be driven out. Use the Set Output Data method before the Set Output Enable method to specify the state of the digital resource when you enable the output. For example, you might have one portion of the block diagram continuously generating an internal signal. Use the FPGA I/O Method Node with the Set Output Enable method in another portion of the block diagram to independently control when the internal signal is actually driven out to an external device.

The FPGA devices do not have built-in counter hardware. All counters must be programmed into the FPGA itself. The count register can be 32, 16, or 8 bits, depending on the type of integer selected for the counter indicator. The loop period also determines the minimum detectable pulse width. An example counter is shown in Figure 4-3.

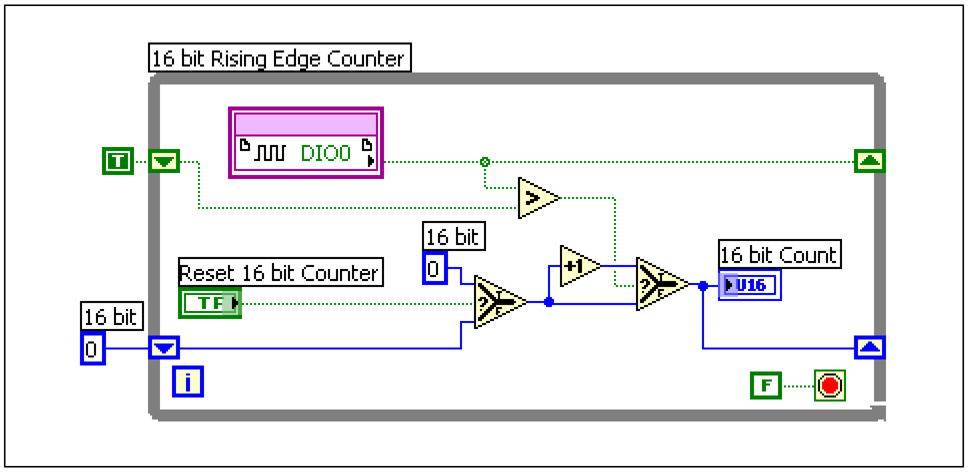


Figure 4-3. 16-bit Rising Edge Counter

It is important to benchmark your counter before using it in a final application. Refer to **Toolkits and Modules»FPGA»Compact RIO/R Series»FPGA Fundamentals»Counters** in the NI Example Finder for related examples.

Analog

There are two analog I/O data types available for use, depending on your FPGA target. If you use an R Series target, the analog I/O data type is either an I16 or an I32, depending on the device. These devices return uncalibrated data. If you use a CompactRIO target, then the default data type is fixed-point and the data has been calibrated. You can, however, choose to configure the module to return raw integer data as well.

Floating-point data cannot be used in FPGA VIs. If you must perform floating-point analysis, you should pass the data to a host VI. More information about passing data between the FPGA and the host can be found in Lesson 8, *Basic Host Integration – PC/Real-Time*.

D. Integer Math

If you configure an FPGA I/O Node to read an analog input, the FPGA I/O Node might initiate a conversion, wait for the result, and then return the binary representation of the voltage as a signed integer. The analog input process and the size of the resulting data type varies by FPGA target. For many FPGA targets, you create the FPGA VI to use the binary representation for operations within the FPGA VI. You also can pass the binary representation back to the host VI and convert the binary representation to a voltage or other physical quantity.

If you configure the FPGA I/O Node to write an analog output, the FPGA I/O Node might write the binary representation of the voltage to the digital-to-analog converter (DAC), which sets the analog output voltage. The size of the data type varies by FPGA target. You can generate voltage information in two sources-the host VI or the FPGA VI. Typically, the host VI converts the voltage to an appropriate binary representation before writing the value to the FPGA VI. If the FPGA VI determines the voltage, typically the FPGA VI performs the calculations using the appropriate binary representations. In both cases, the DAC produces a voltage that corresponds to the binary representation.

Most of the math functions on the Numeric palette support both integer and fixed-point data types. Notable exceptions include the Divide, Reciprocal and Square Root functions, which will only work on the FPGA target if you are using fixed-point data.

It is possible to achieve some of the functionality of the Divide function for integer data if you make use of the Scale by Power of 2 and Quotient & Remainder functions. You use these functions to perform simple scaling of data, as shown in Figure 4-4.

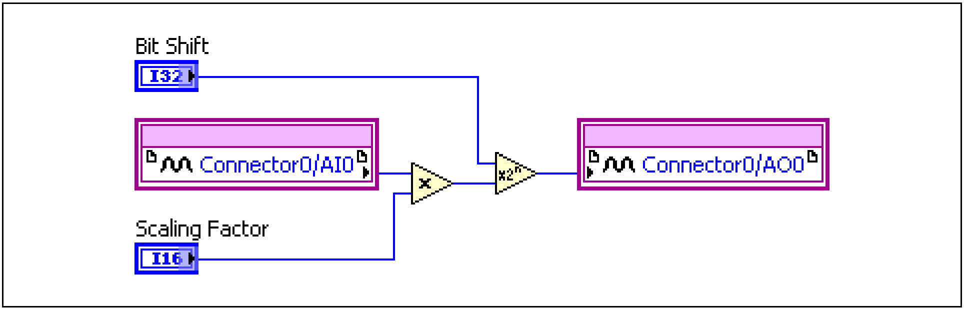


Figure 4-4. R Series Scaling on FPGA

Suppose we want to scale the data by .70. To do so, we multiply the analog input by a Scaling Factor of 11500, then divide it by 2^{14} by using the Scale by the Power of 2 function with the Bit Shift (n input of the function) set to –14. This results in the data from Connector0/AI0 being multiplied by 0.7019.

Additionally, you can divide numbers using the quotient and remainder function, however, this can use many FPGA resources when implemented.

Converting Binary Representations

When you configure the FPGA I/O Node to read an analog input, the FPGA I/O Node initiates a conversion, waits for the result, and returns the binary, uncalibrated representation of the voltage as a signed, 32-bit integer (I32). The analog input process and the size of the resultant data type varies by FPGA target.

The equation that converts the binary representation to a physical quantity depends on the FPGA target and transducer. In an R series target, avoid executing this calculation in the FPGA VI because the R Series FPGA supports only integer operations, and performing these calculations consumes a lot of space on the FPGA.

It is good practice to convert and calibrate the I/O values in a host VI.

E. Fixed-Point Math

The fixed-point data type is a numeric data type that represents a set of rational numbers using binary digits, or bits. Unlike the floating-point data type, which allows the total number of bits LabVIEW uses to represent numbers to vary, you configure fixed-point numbers to always use a specific number of bits. Hardware and targets that only can store and process data with a limited or fixed number of bits then can store and process these numbers. You can specify the range and precision of fixed-point numbers, specify any size between 1 and 64 bits, inclusive, and configure fixed-point numbers as signed or unsigned.

Note To represent a rational number using the fixed-point data type, the denominator of the rational number must be a power of 2, because the binary number system is a base-2 number system.

Use the fixed-point data type you are working with a target that does not support floating-point arithmetic, such as an FPGA target. The fixed-point data type provides some of the flexibility of the floating-point data type but also maintains the size and speed advantages of integer arithmetic. By default, each operation on the fixed-point data type generates a fixed-point result that is large enough to hold all possible output values specified by the input types.

Note FPGA Signal Generation VIs and some functions do not support the fixed-point data type.

Caution If you wire a fixed-point number to an integer, you might lose fractional bits.

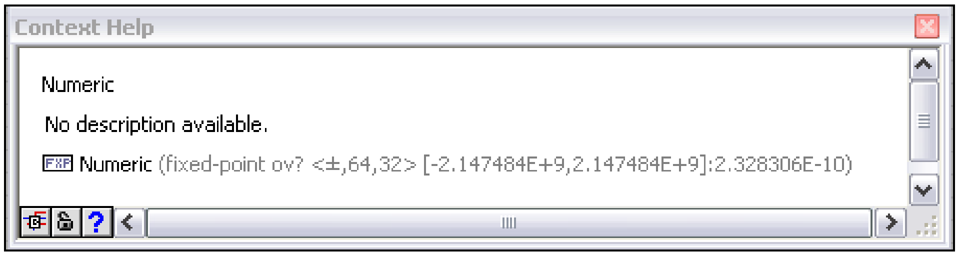
The range of a fixed-point data type describes the minimum value, maximum value and delta (the smallest change in data). These three values are determined by the sign encoding, word length, and integer word length of the fixed-point data.

Encoding, Word Length, and Integer Word Length

The sign encoding, word length, and integer word length are the three parameters that define the values that can be represented by a fixed-point number. These parameters are all user-defined and can be modified by right-clicking on a fixed-point constant, control, or function and selecting **Properties**.

- **Signed—**The setting that specifies whether the fixed-point value is signed or unsigned. If you select signed, the sign bit is always the first bit in the bit string that represents the data.
- **Word length—**The total number of bits in the bit string that LabVIEW uses to represent all possible values of the fixed-point data. LabVIEW accepts a maximum word length of 64 bits. Certain targets might limit data to smaller word lengths. If you open a VI on a target and the VI contains fixed-point data with larger word lengths than the target can accept, the VI contains broken wires. Refer to the documentation for a target to determine the maximum word length the target accepts.
- **Integer word length—**The number of integer bits in the bit string that LabVIEW uses to represent all possible values of the fixed-point data, or, given an initial position to the left or right of the most significant bit, the number of bits to shift the binary point to reach the most significant bit. The integer word length can be larger than the word length, and can be positive or negative.

When performing calculations with fixed-point data, it is important to track the configuration of the fixed-point data as the data is manipulated. Most operations result in a different configuration of fixed-point data at the output than the input. The easiest way to view the configuration of the fixed-point data, as well as the minimum, maximum and delta values, is to display the Context Help window and click on the wire, control, or indicator. Figure 4-5 displays the data type of a particular indicator.



Numeric

No description available.

Numeric (fixed-point ov? <±,64,32> [-2.147484E+9,2.147484E+9]:2.328306E-10)

Figure 4-5. Fixed Point Context Help

Numeric Representation of Fixed-Point Numbers

The maximum and minimum values of a fixed-point number are dependent on the sign encoding, word length, and integer word length of the number. The delta of the number is based on the difference between the word length and the integer word length.

When the fixed-point number is unsigned, it is represented as <+,X,Y>. For an unsigned fixed-point number, the maximum value of the fixed-point data is $2^{Y} – 1/(2^{(X–Y)})$ and the delta is $1/(2^{(X–Y)})$. The minimum value that can be represented is zero. For example:

<+,8,6> is an unsigned 8-bit number with six integer bits and two decimal bits. The maximum value that can be represented is $2^{6} – 1/(2^{(8–6)}) = 64 – ¼ = 63.75$ and the delta is $1/(2^{(8–6)}) = 0.25$.

When the fixed-point number is signed, it is represented as <±,X,Y>. In this case, the maximum value of the fixed-point data is $2^{Y–1} – 1/(2^{(X–Y)})$. The delta is still represented as $1/(2^{(X–Y)})$. The minimum value that can be represented is $–2^{Y–1}$. For example:

<±,8,6> is a signed 8-bit number with six integer bits. The maximum value that can be represented is $2^{6–1} – 1/(2^{(8–6)}) = 31.75$ and the delta is $1/(2^{(8–6)}) = 0.25$. The minimum value that can be represented is $–2^{6–1} = –32$.

Note The difference between the maximum and minimum values for a signed fixed-point number is the same as the difference between the maximum and minimum values for an unsigned fixed-point number with the same word and integer word lengths. The delta also does not change.

Table 4-2. Numeric Representation Examples

Representation	Delta	Minimum Value	Maximum Value
U8	1	0	255
I8	1	–128	127
FXP <+,8,7>	0.5	0	127.5
FXP <+,8,6>	0.25	0	63.75
FXP <±,8,7>	0.5	–64	63.5
FXP <±,8,6>	0.25	–32	31.75
FXP <+,8,0>	0.0039	0	0.9961

If the integer word length is larger than the word length, LabVIEW does not store the integer bits that exceed the word length. For <+,8,10>, the two least significant bits would not be stored. The maximum value that can be represented by this number is $2^{10} - 1/(2^{(8-10)}) = 1020$ and the delta is $1/(2^{(8-10)}) = 4$.

If the integer word length is negative, LabVIEW does not store any integer bits and also does not store the number of fractional bits equal to the negative number, starting from the binary point. Thus, for <+,8,–2>, the maximum value that can be represented is $2^{-2} - 1/(2^{(8+2)}) = 0.2490234375$ and the delta is $1/(2^{(8+2)}) = 9.765625E-4$.

Fixed-Point Configuration

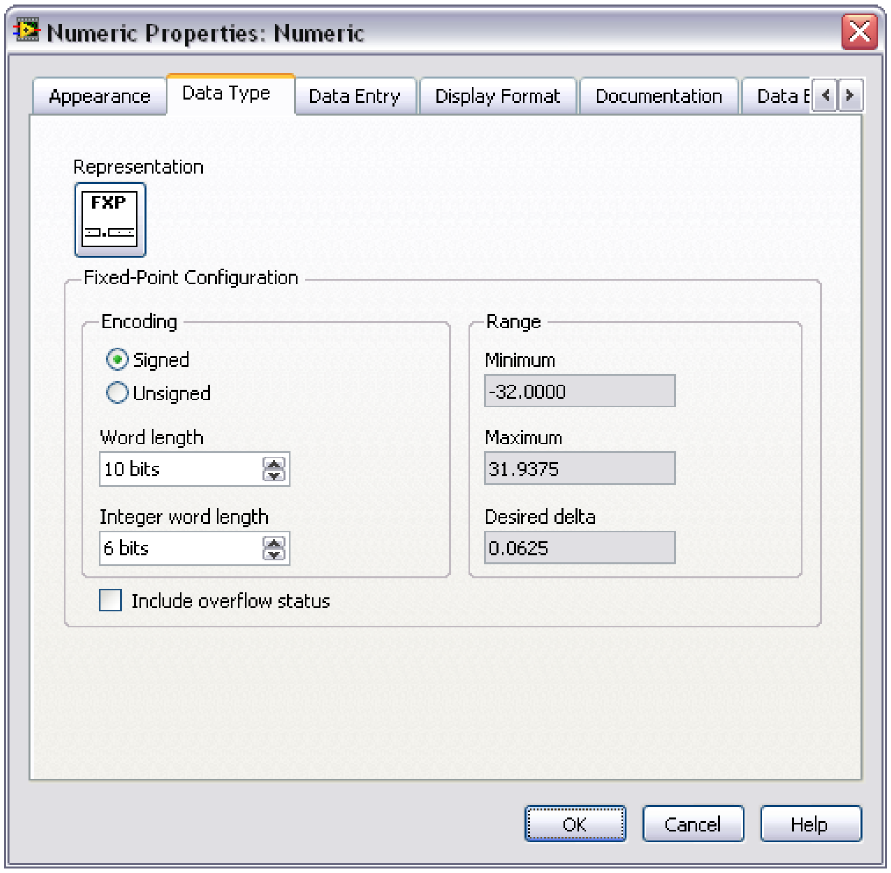
You can configure a fixed-point number in a numeric control, constant, or indicator.

After you configure a fixed-point control, constant, or indicator, that object cannot display a number that does not conform to the settings you specify. If the object you configure is an indicator, LabVIEW coerces any input value to the indicator to conform to the fixed-point configuration settings of the indicator.

Configuring Controls

Complete the following steps to configure a fixed-point number in a numeric control.

1. Right-click the control and select **Properties** from the shortcut menu to display the Numeric Properties dialog box.



Fixed-Point Configuration

Encoding: [✓ Signed] Unsigned

Word length: 10 bits

Integer word length: 6 bits

Range

Minimum: -32.0000

Maximum: 31.9375

Desired delta: 0.0625

Include overflow status

Figure 4-6. Fixed-Point Control Configuration

2. On the Data Type page, click the data type icon in the Representation section and select **FXP (Fixed-point)** from the shortcut menu. The Fixed-Point Configuration section displays default values for the Encoding and Range options.
3. Complete the following steps to configure the Range of the fixed-point number.
   a. Select **Signed** or **Unsigned** to specify whether you want to represent a signed or unsigned number.
   b. In the **Word length** field, specify the total number of bits you want to use to represent the value of the fixed-point number.
   c. In the **Integer word length** field, specify the number of integer bits you want to use to represent the value of the fixed-point number.

4. (Optional) Place a checkmark in the **Include overflow status** checkbox to include an overflow status in the fixed-point number.
5. Click **OK** to close the dialog box and apply the configuration settings.

Configuring Constants

Complete the following steps to configure a fixed-point number in a numeric constant.

1. Right-click the constant and select **Properties** from the shortcut menu to display the Numeric Constant Properties dialog box.
2. On the Data Type page, remove the checkmark from the **Adapt to entered data** checkbox.

Note When you select **Adapt to entered data**, LabVIEW displays any value you enter with the shortest possible word length and integer word length. You must remove the checkmark from this checkbox if you want to make changes to the Fixed-Point Configuration settings.

3. Click the data type icon in the Representation section and select **FXP (Fixed-point)** from the shortcut menu. The Fixed-Point Configuration section displays default values for the Range and Encoding options.
4. Complete the following steps to configure the Range of the fixed-point number.
   a. Select **Signed** or **Unsigned** to specify whether you want to represent a signed or unsigned number.
   b. In the **Word length** field, specify the total number of bits you want to use to represent the value of the fixed-point number.
   c. In the **Integer word length** field, specify the number of integer bits you want to use to represent the value of the fixed-point number.
5. (Optional) Place a checkmark in the **Include overflow status** checkbox to include an overflow status in the fixed-point number.
6. Click **OK** to close the dialog box and apply the configuration settings.

Configuring Indicators

Complete the following steps to configure a fixed-point number in a numeric indicator.

1. Right-click the indicator and select **Properties** from the shortcut menu to display the Numeric Properties dialog box.
2. On the Data Type page, click the data type icon in the Representation section and select **FXP (Fixed-point)** from the shortcut menu. The Fixed-Point Configuration section displays default values for the Range and Encoding options.

3. Place a checkmark in the **Adapt to source** checkbox if you want the value to inherit the fixed-point configuration settings of an input fixed-point value. If you select this option, skip to step 6.
4. (Optional) Complete the following steps to configure the Range of the fixed-point number.
   a. In the **Minimum** field, enter the minimum value to which you want the fixed-point number to conform.
   b. In the **Maximum** field, enter the maximum value to which you want the fixed-point number to conform.
   c. In the **Desired delta** field, enter the increment between numbers within the Range.
5. (Optional) Place a checkmark in the **Include overflow status** checkbox to include an overflow status in the fixed-point number.
6. Click **OK** to close the dialog box and apply the configuration settings.

Fixed-Point Arithmetic

When performing arithmetic on integer data, the output data type matches the input data types. However, when you add two values that are the maximum values allowed for that data type, the result will not be accurate. For example, adding together two U16 integers with a value of 65535 should result in a value of 131070. However, that value is outside of the range of values allowed by a U16, so the result is an inaccurate output.

For fixed-point arithmetic, the output of the function is configured to accommodate the largest possible result based on the inputs. The size of the output is limited to 64 bits, which is a limitation of the data type. LabVIEW propagates the range of values along each wire in the diagram, not fixed-point representation. This allows LabVIEW to reduce resources when possible. For terminals and other points when representation is necessary, the representation is then calculated to be as small as possible without losing data.

In the following sections, we examine four arithmetic functions: addition, subtraction, multiplication, and division. Because LabVIEW propagates ranges, the formulas in the following sections are helpful guidelines for calculating the representation of an output value, but are not the formulas that LabVIEW uses.

Note For information on performing high throughput math and analysis, refer to the *Using the High Throughput Math Functions* topic of the *LabVIEW Help*.

Addition

When using the Add function, the two inputs should have the same fixed-point configuration. If they have different configurations, then one or both of the inputs will be coerced to a configuration that can accommodate the full range of both inputs. For the output, the word length and integer word length are each increased by one and the sign encoding matches the setting of the inputs. Thus, if the inputs are each configured as <±,A,B>, then the output will be configured as <±,A+1,B+1>. Figure 4-7 shows an example of addition of fixed point data.

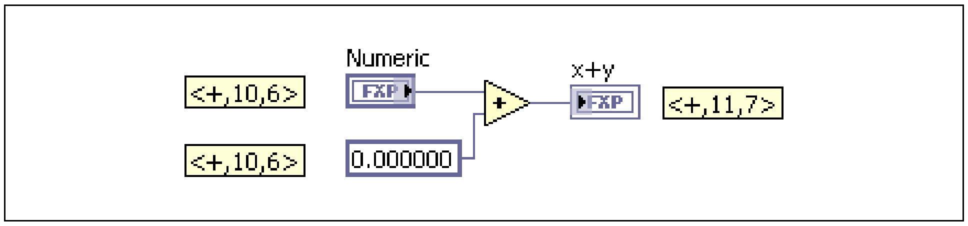


Figure 4-7. Fixed-Point Addition

For the output, integer word length increases by one to accommodate the maximum value. Since the integer word length increases, the word length must also increase so that no accuracy is lost. Thus, <±,10,6> + <±,10,6> = <±,11,7>

To add two fixed-point values with different configurations, you must use the longer integer word length from the two inputs for the output, and you must add together the longer word length and the longer decimal word length (word length - integer word length) to determine the word length of the output. This is done to ensure that no accuracy is lost.

For example, adding fixed-point values with configurations of <±,10,7> and <±,10,6> requires that the output have 7 integer bits (from <±,10,7>) and 4 decimal bits (from <±,10,6>), resulting in both inputs being coerced to <±,11,7>. For the output, <±,11,7> + <±,11,7> = <±,12,8>

If one input is signed and the other is unsigned, then the word length and integer word length will increase by one more bit.

Subtraction

When using the Subtract function, the two inputs should have the same fixed-point configuration, as described for addition. Also, as with addition, the input word length and integer word length each be increase by one for the output. Unlike addition, the output is always signed, regardless of whether or not the inputs were signed. Thus, if the inputs are each configured as <+,A,B>, then the output is configured as <±,A+1,B+1>. Figure 4-8 shows an example of subtraction of fixed point data.

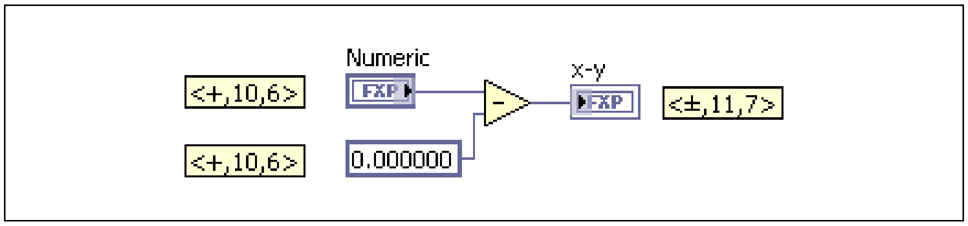


Figure 4-8. Fixed-Point Subtraction

As with addition, if the inputs have different configurations, they are coerced to a common configuration that accommodates both data types.

Multiplication

When using the Multiply function, the two inputs are not required to have the same fixed-point configuration. For the integer value of the output to accommodate the largest possible value of each operand, the integer word length must increase to match the sum of the integer word lengths of the inputs. For the resolution of the output to accommodate the highest resolution of both inputs, the decimal word length must also increase. Thus, the word length of the output must increase to match the sum of the word lengths of the inputs. If either input is signed, then the output is signed. Otherwise, the output is unsigned. Thus, if the inputs are each configured as <±,A,B> and <+,C,D>, then the output is configured as <±,A+C,B+D>. Figure 4-9 shows an example of multiplication of fixed point data.

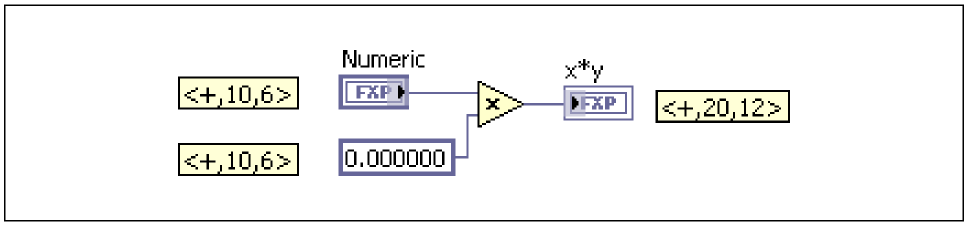


Figure 4-9. Fixed-Point Multiplication

Division

When using the Divide function, the two inputs are not required to have the same fixed-point configuration. For the output, the word length and integer word length are calculated differently depending on whether or not the inputs are signed.

- Signed: <±,A,B> + <±,C,D> = <±,A+C+1,B+C–D+1>
- Unsigned: <+,A,B> + <+,C,D> = <+,A+C, B+C–D>

An example of both signed and unsigned fixed-point division is shown in Figure 4-10.

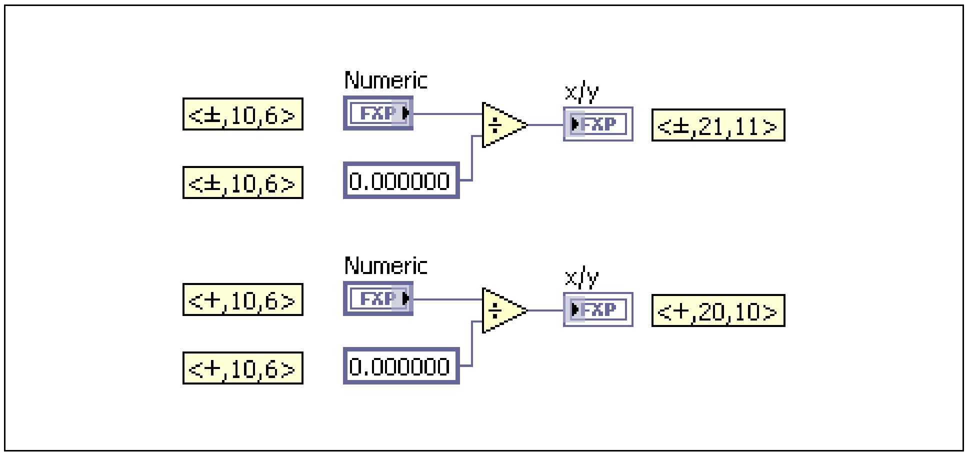
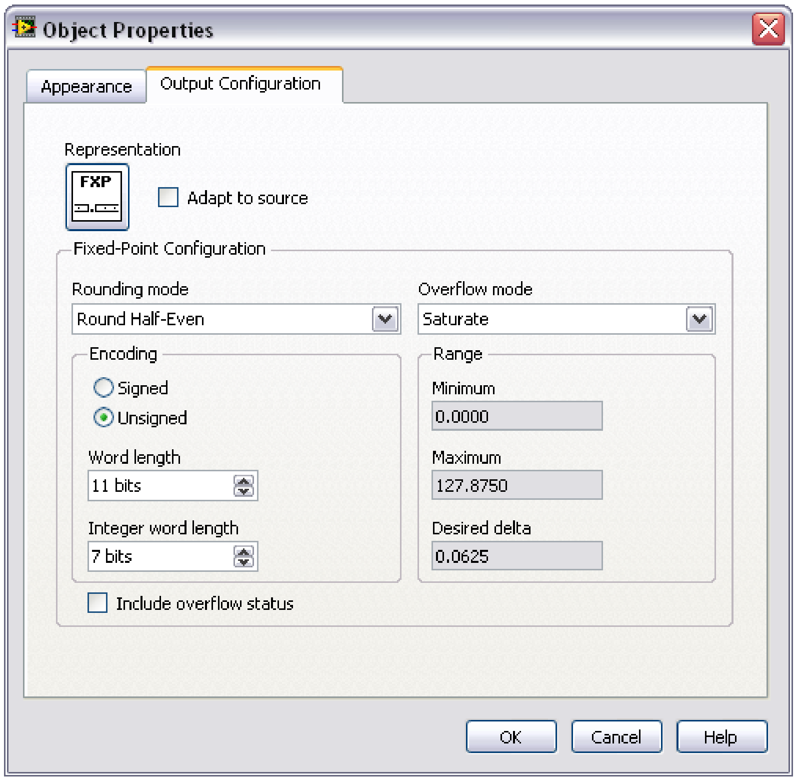


Figure 4-10. Fixed-Point Division

Configuring Fixed-Point Functions

Functions that support the fixed-point data type (add, subtract, multiply, and so on) include modes to handle the overflow and rounding. In the Properties dialog box for the function, you select the overflow and rounding modes. Right-click a function and select **Properties** from the shortcut menu to display the Properties dialog box, shown in Figure 4-11.



Representation
Adapt to source: ☐

Fixed-Point Configuration

Rounding mode: Round Half-Even
Overflow mode: Saturate

Encoding: Signed [✓ Unsigned]
Word length: 11 bits
Integer word length: 7 bits

Range
Minimum: 0.0000
Maximum: 127.8750
Desired delta: 0.0625

Include overflow status: ☐

Figure 4-11. Fixed-Point Function Configuration

The Properties dialog box contains the following three options that are specific to configuring a fixed-point function:

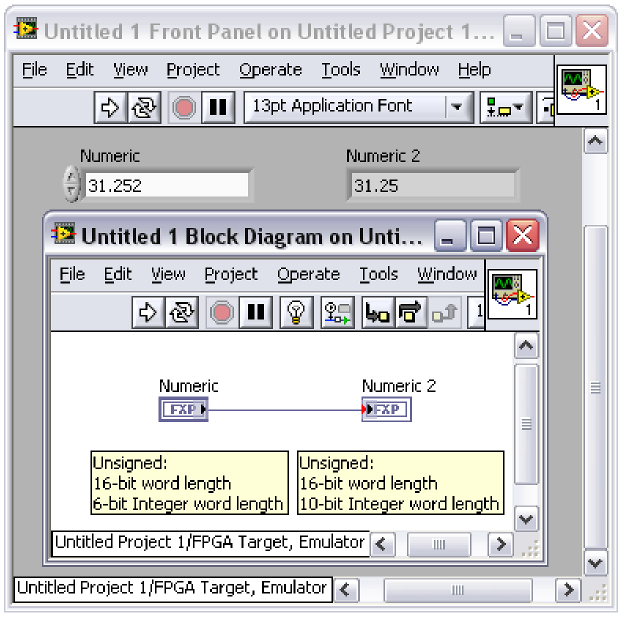
- **Adapt to Source**—Sets the configuration settings for the output value to adapt to the input values you wire to the function. For fixed-point data, LabVIEW automatically sets the Fixed-Point Configuration settings to avoid data loss, if possible.
- **Rounding Mode**—Determines how the function handles quantization conditions.
- **Overflow Mode**—Determines how the function handles overflow conditions.

Rounding occurs when the precision of the input value or the result of an operation is greater than the precision of the output type. The rounding modes affect the logic generated within the FPGA as follows:

- **Truncate**—Removes fractional bits and therefore does not require any additional hardware resources. However, this mode produces the largest mean error for most data streams. This mode is the default for integer operations.
- **Round-Half-Up (Asymmetric)**—Adds to the least significant bit of the output data type. This option requires an adder that is the width of the output data type.
- **Round-Half-Even**—Requires the most FPGA resources and results in the longest combinatorial path of the three rounding modes. However, this mode returns the most statistically correct results for most data streams and is therefore the default rounding mode for the fixed-point data type.

Overflow occurs when the result of an operation is outside the range of values that the output type can represent. The overflow modes affect the logic generated within the FPGA as follows:

- **Saturate**—Requires FPGA resources to determine whether the input value is within the range of the output type.
- **Wrap**—Requires fewer hardware resources than the Saturate mode.


Untitled 1 Front Panel on Untitled Project 1...
File Edit View Project Operate Tools Window Help
13pt Application Font
Untitled 1 Block Diagram on Unti...
File Edit View Project Operate Tools Window
FXP
FXP
Untitled Project 1/FPGA Target, Emulator
Untitled Project 1/FPGA Target, Emulator

Numeric: 31.252

Numeric 2: 31.25

Numeric — Unsigned: 16-bit word length, 6-bit Integer word length

Numeric 2 — Unsigned: 16-bit word length, 10-bit Integer word length

Figure 4-12. Mismatched Fixed-Point Data

If you select the Saturate, Round-Half-Up (Asymmetric), or Round-Half-Even modes and the output can handle the overflow or rounding, the operation does not require additional hardware resources. If a coercion dot does not appear on the block diagram, the output can handle the overflow or rounding without additional hardware resources. If a coercion dot does appear, then there is a risk of data loss.

For more information about fixed-point data and fixed-point arithmetic, refer to the following *LabVIEW Help* topics:

- *Using the Fixed-Point Data Type* (FPGA-specific topic)
- *Numeric Data*
- *Configuring Fixed-Point Numbers*
- *Numeric Data Types Table*

F. CompactRIO

One of the differentiating characteristics of CompactRIO is the high degree of customization that can be achieved using modules that fit your specific needs. Each CompactRIO module has different functional features and specifications. Similarly, the input/output for each module has its own features.

Different modules return data in different fixed-point data types. Differences in configuration are due to differences in the hardware for each module. Hardware differences result in variations in accuracy and range for the data that is generated or obtained. Some modules enable you to configure their I/O nodes to return data that is both calibrated and scaled.

All modules share certain common properties, like Module ID, Serial Number, and Vendor ID. Some modules, however, have their own unique properties. For example, the NI 9233 has a Data Rate property that allows the user to programmatically configure the sample rate of the device. Some modules even have their own methods that can be called. For example, the NI 9263 analog output module has three methods: Update, Wait For Update, and Write Data. The NI 9233, however, does not have any module-specific methods associated with it.

Different modules have different timing and sampling capabilities. For example, the NI 9211 acquires a single sample per cycle at its set data rate. The NI 9233 is able to acquire a single sample from each of its channels at a user-defined sample rate.

It is important to understand the features and specifications of the CompactRIO modules you use. Refer to the *CompactRIO References and Procedures* book in the *LabVIEW Help* for more information.

G. Error Handling

If an error occurs, you might receive incorrect data. Add error parameters to be sure the data you receive is valid. To add standard LabVIEW error in and error out parameters to a function, you can right-click the FPGA I/O Node on the block diagram and select **Show Error Terminals**.

Note FPGA targets might report errors differently. Refer to the specific FPGA target hardware documentation for information about how specific FPGA targets report errors.

Adding error in and error out parameters increases the amount of space the function uses on the FPGA target. The error in and error out parameters also slows execution on the FPGA target. So, use them as a general rule, but if you have difficulty with either the size or the speed of the FPGA application, remove them after carefully testing your code.

If FPGA resources are limited, there are number of optimizations related to the use of error clusters that you can perform.

- If you are using the error wire for dataflow, remove the error wires and use a Sequence Structure instead. The Sequence Structure has a minimal cost when used in FPGA.
- Show error terminals only for modules whose functions are critical to system operation.
- Show terminals only once per module if multiple calls are made to the module. The exception to this rule is modules that have specific errors. Checking once allows you to check for general errors.
- Do not pass error clusters through the FPGA VI or display the error cluster on the front panel.
- Unbundle source and/or code items and handle the errors immediately.

Some types of FPGA I/O errors must be handled in all cases.

- All safety-critical I/O operations should be monitored for errors.
- Some modules have specific error codes associated with them. For example, the NI CAN modules generate an error if a timeout occurs. The NI 9802 generates an error if it attempts to open a file that cannot be found.
- If the module being used has been removed or is not secure, an error is generated.

Self-Review: Quiz

1. Match each FPGA I/O term to the definition that best fits.

Terminal	A name assigned by the developer to a particular I/O Resource.
I/O Resource	A hardware connection, such as on a CompactRIO module.
I/O Name	A logical representation in LabVIEW FPGA of a terminal.

2. What is the default data type of the result of adding two signed fixed-point numbers with a word length of 20 and an integer word length of 10?
   a. <±,40,20>
   b. <+,20,10>
   c. <±,21,11>
   d. <+,30,30>

3. Which of the following are parameters used to define the range of values that are represented by a fixed-point number?
   a. Terminal
   b. Sign Encoding
   c. Word Length
   d. Integer Word Length

4. Match each analog I/O device to its default data type.

NI PCI-7831R R Series board	Integer
NI 9233 CompactRIO module	Fixed-point

Self-Review: Quiz Answers

1. Match each FPGA I/O term to the definition that best fits.

Terminal	**A hardware connection, such as on a CompactRIO module.**
I/O Resource	**A logical representation in LabVIEW FPGA of a terminal.**
I/O Name	**A name assigned by the developer to a particular I/O Resource.**

2. What is the default data type of the result of adding two signed fixed-point numbers with a word length of 20 and an integer word length of 10?
   a. <±,40,20>—This answer would have been correct if the values were being multiplied.
   b. <+,20,10>
   c. <±,21,11>—Since the inputs have the same data type, the integer word length and word length are each increased by one.
   d. <+,30,30>

3. Which of the following are parameters used to define the range of values that are represented by a fixed-point number?
   a. Terminal
   b. Sign Encoding—Determines whether the value is signed or unsigned.
   c. Word Length—Determines the number of bits used to represent the number.
   d. Integer Word Length—Determines the number of bits used to represent the integer portion of the number.

4. Match each analog I/O device to its default data type.

NI PCI-7831R R Series board	**Integer**
NI 9233 CompactRIO module	**Fixed-point**

Notes

5

Timing an FPGA VI

In this lesson, you learn to use the Loop Timer to set your FPGA loop rates, the Wait to add delays between events, and the Tick Count to benchmark your FPGA code.

Topics

A. Timing Express VIs

Every VI or function you place in an FPGA VI takes a certain amount of time to execute. You can allow operations to occur at the rate determined by the dataflow without additional programming. If you want to control or measure the execution timing, use Timing VIs. You also can use the Timing VIs to create custom I/O applications, such as counters and triggers.

Timing Express VIs

Loop Timer

The Loop Timer Express VI waits between loop iterations based on a value specified in Count. You can call this function in a loop to control the loop execution rate. If an execution instance is missed, such as when the logic in the loop takes longer to execute than the specified interval, the Loop Timer returns immediately and establishes a new reference time stamp for subsequent calls.

Use the Loop Timer Express VI to control a For Loop or While Loop and set the iteration rate of the loop. A common use of the Loop Timer Express VI is to control the acquisition or update rate of an analog or digital I/O function.

Wait

The Wait Express VI waits a specified time and then returns the value of a free-running counter. The Wait Express VI adds an explicit delay between two operations on the FPGA. Use the Wait Express VI to control the pulse length of a digital output or to add a trigger delay between the trigger signal and the resulting operation.

Tick Count

The Tick Count Express VI returns the value of a free running counter at the time the VI wakes up. A free running counter rolls over when the counter reaches the maximum of Size of Internal Counter specified in the configuration dialog box. Use the Tick Count Express VI to benchmark loop rates or create your own custom timers.

Timing Express VI Configuration

Each timing Express VI has a configuration dialog box that appears when you add the VI to the block diagram or when you right-click the VI and select **Properties**. The configuration dialog box for the Timing Express VIs includes the following options:

- **Counter Units**—Unit of time the VI uses for the counter.
  - **Ticks**—Sets the counter units to a single clock cycle, the length of which is determined by the clock rate for which the VI is compiled.
  - **µSec**—Sets the counter units to microseconds.
  - **mSec**—Sets the counter units to milliseconds.
- **Size of Internal Counter**—Specifies the maximum size of the internal counter (8, 16, or 32 bits). The free-running counter rolls over when the counter reaches the maximum count that can represented using the selected number of bits. To save space on the FPGA, use the smallest Size of Internal Counter possible for the FPGA VI.

B. Implementing Loop Execution Rates

Applications often require loops to execute at a specific frequency. For example, the algorithms used in control loops typically require that inputs are sampled at a known rate. Use the Loop Timer VI in a While Loop to control the loop execution rate, as shown in Figure 5-1.

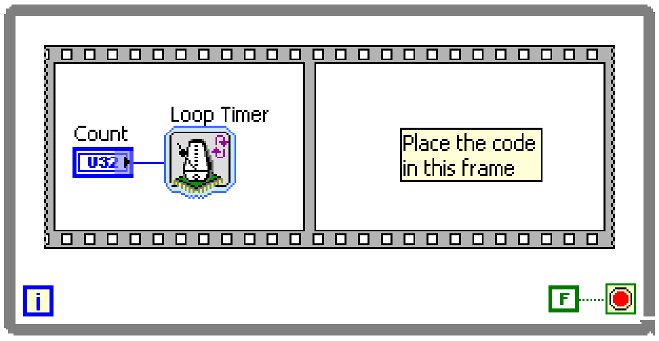


Figure 5-1. Implementing Loop Execution Rate with Loop Timer Express VI

To control the loop execution rate using the Loop Timer VI, place a sequence structure inside a While Loop. Place the Loop Timer VI in the first frame of the sequence structure. In the Configure Loop Timer dialog box that appears, specify Counter Units and Size of Internal Counter. Place the LabVIEW code for the I/O in subsequent frames of the sequence structure.

The first time the Loop Timer executes in a loop, it records the current time. The next time the Loop Timer executes, it adds Count to the initial time and waits until Count has elapsed from the initial recorded time. The Loop

Timer does not wait the first time you call it in an FPGA VI. If you place the Loop Timer in a loop so that it executes when the loop starts, all the code parallel with the Loop Timer in the loop executes twice, after the initial time, before Count elapses. To prevent code from executing twice before Count elapses, use a Flat Sequence structure or a Stacked Sequence structure with the Loop Timer in the first frame and place the rest of the code in subsequent frames to ensure that the code for the first and subsequent iterations is properly timed.

When called repeatedly through nested structures or continuous run mode, the Loop Timer's timing does not reset each time. The Loop Timer continues to increment the time record it initiated upon the first call.

While Loop Considerations

Wiring a False constant to the Loop Condition terminal of a While Loop as shown in Figure 5-1 is acceptable in many FPGA applications. FPGA logic is often meant to run indefinitely on the FPGA. Wiring controls and application logic to the Loop Condition terminal instead is also acceptable as shown in Figure 5-2.

The iteration terminal count of the While Loop does not roll over. Instead, it will keep outputting 2,147,483,647 once it reaches that value. With FPGA speeds, it is possible for the While Loop iteration terminal to max out rather quickly. If you need a counter to rollover when it reaches its maximum value, you should programmatically implement your own counter. An example of this is shown in Figure 5-2.

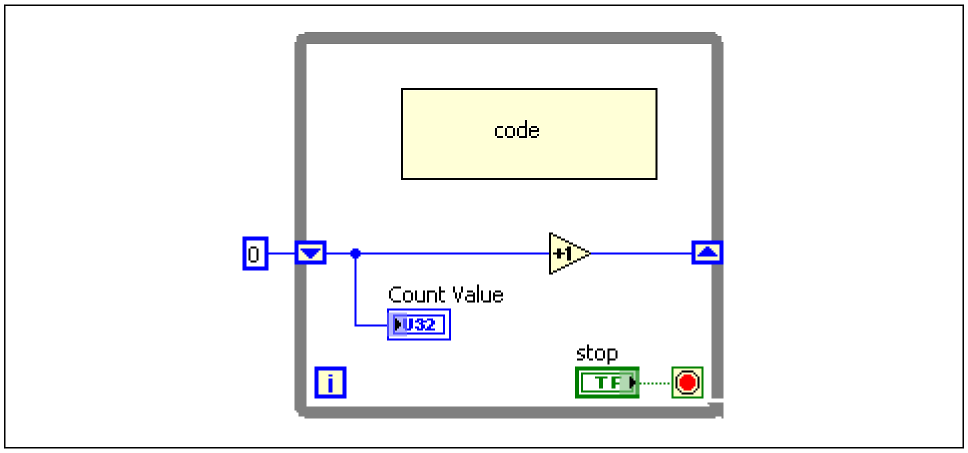


Figure 5-2. Programmatic Implementation of a Counter

CompactRIO Timed Modules

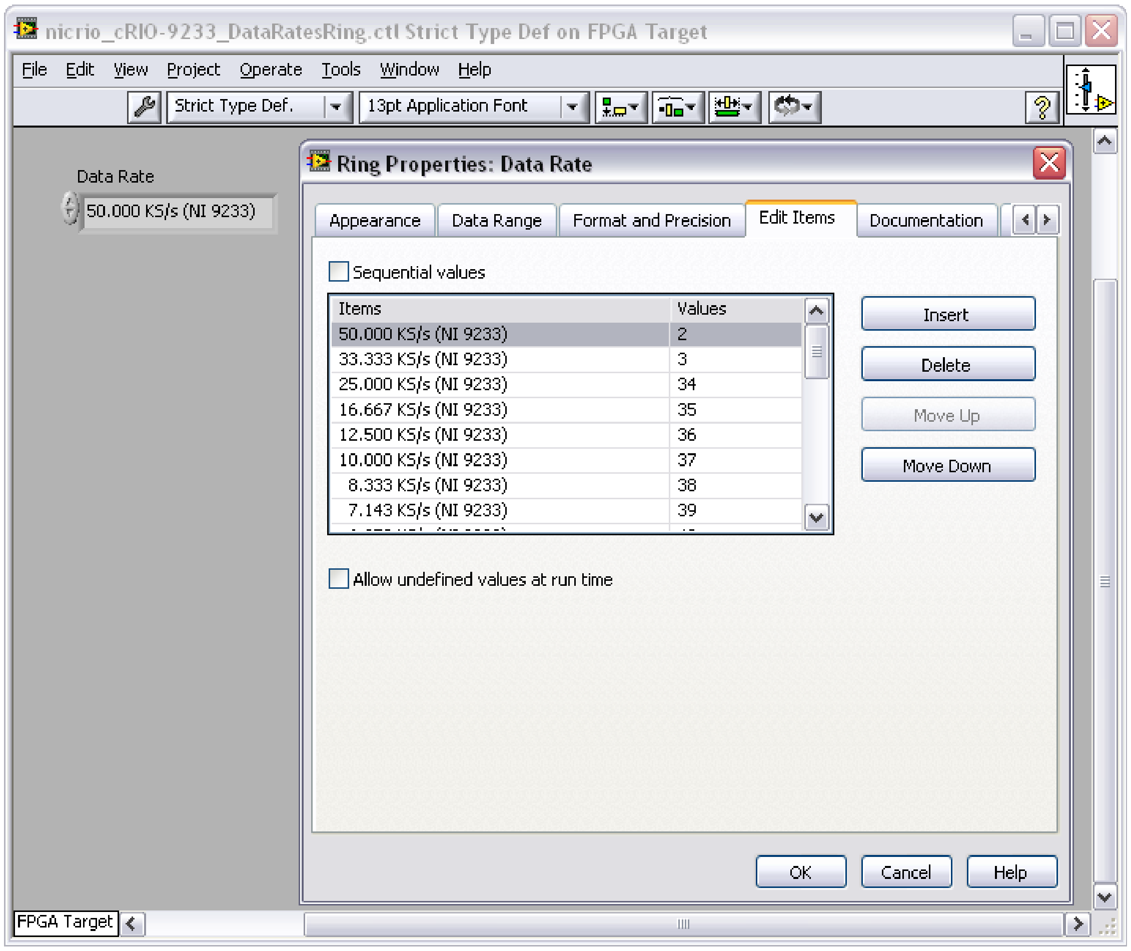
Some CompactRIO modules can be configured to transfer data at a user-specified data rate. For specific information, refer to the product documentation for CompactRIO modules with this feature.

For example, the NI 9233 has four analog input channels that are sampled simultaneously at a user specified data rate. The NI 9233 has two additional digital channels, Start and Stop, that control the acquisition mode. To start acquisition, drag the Start channel from the Project Explorer to the block diagram. Wire a Boolean constant set to TRUE to the **Start** input to send a synchronization pulse to the module and start acquiring data at the user-specified data rate.

Use the FPGA I/O Property Node to configure the data rate programmatically. Execute the Property Node before the Start node or after the Stop node. You cannot change the properties while the NI 9233 is in acquisition mode.

To configure the Property Node, right-click the **Property** section and select **Data Rate** and then right-click the **Data Rate** input and select **Create» Control**.

The Data Rate control, shown in Figure 5-3, is a Strict Type Definition custom ring control that limits user selections to rates allowed for the NI 9233. You write to the control from the host VI so that you can change the data rate at run time.



nicrio_cRIO-9233_DataRatesRing.ctl Strict Type Def on FPGA Target

Data Rate: 50.000 KS/s (NI 9233)

Ring Properties: Data Rate

☐ Sequential values

Items	Values
50.000 KS/s (NI 9233)	2
33.333 KS/s (NI 9233)	3
25.000 KS/s (NI 9233)	34
16.667 KS/s (NI 9233)	35
12.500 KS/s (NI 9233)	36
10.000 KS/s (NI 9233)	37
8.333 KS/s (NI 9233)	38
7.143 KS/s (NI 9233)	39

☐ Allow undefined values at run time

Figure 5-3. NI 9233 Data Rate Control

Use the C Series Module Properties dialog box to set the rate statically. Right-click the NI 9233 in the Project Explorer and select **Properties** to display the C Series Module Properties dialog box. Select the rate from the **Data Rate** pull-down menu, click **OK**, and save the project.

The execution of an I/O Property Node configured with a Data Rate property overwrites the value you configure in the C Series Module Properties dialog box.

Use an FPGA I/O Node to read data after acquisition starts. Connect the AI output of the FPGA I/O Node to an FPGA Memory function or an FPGA FIFO function. If you read from multiple channels on the module, place the channels in the same FPGA I/O Node to ensure that the VI reads the data synchronously.

Because the NI 9233 internally acquires data at a specified rate, the FPGA I/O Node does not return data until the module acquires new data. If the NI 9233 did not start acquiring data or stops acquiring data while an FPGA

I/O Node is waiting for data from the module, the FPGA I/O Node returns a timeout error.

Configure an FPGA I/O Node with the Stop channel of the NI 9233 to exit acquisition mode. Write a True constant to the Stop input. The FPGA must exit acquisition mode before changing properties and methods.

You can use the FPGA I/O Method Node to read the data from the module. You cannot perform other operations, such as accessing properties or Transducer Electronic Data Sheet (TEDS) information, when the NI 9233 module is in acquisition mode.

The NI 9233 is internally timed. Do not use the Loop Timer or Wait functions in a loop with an FPGA I/O Node that acquires data from an NI 9233. When you create a loop that reads data from an NI 9233, make sure the loop does not execute slower than the data rate of the NI 9233.

C. Creating Delays Between Events

Use the Wait Express VI to create a delay between events in an FPGA VI. For example, you might want to create a delay between a trigger and a subsequent output. You can place the LabVIEW code for the trigger in the first frame of a sequence structure. Then, place the Wait Express VI in the following frame. Finally, place the LabVIEW code for the output in the last frame of the sequence structure. You also can create a series of delays using multiple Wait VIs in a sequence structure, as shown in Figure 5-4.

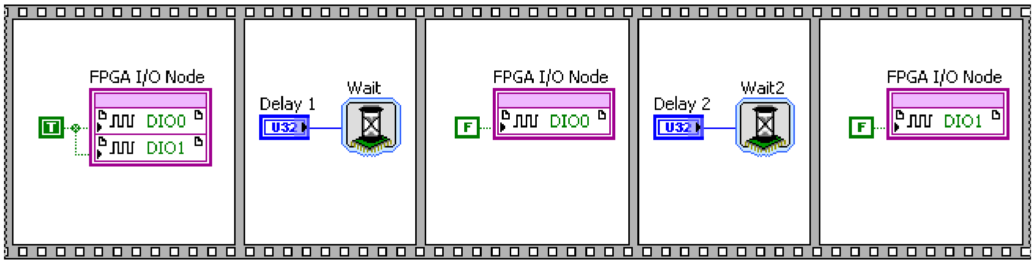


Figure 5-4. Creating Delays Between Events with Wait Express VI

D. Measuring Time Between Events

Use the Tick Count Express VI to measure the time between events such as edges on a digital signal. You can use the Tick Count Express VI when you need to determine the period, pulse-width, or frequency of an input signal or if you want to determine the execution time of a section of LabVIEW code. For example, to determine the amount of time it takes to execute a function or a section of LabVIEW code, use a sequence structure with two Tick Count Express VIs, as shown in Figure 5-5.

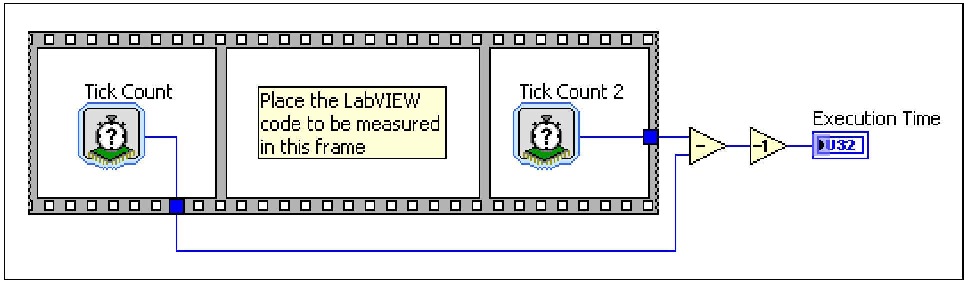


Figure 5-5. Measuring Time Between Events with Tick Count Express VI

Place one Tick Count Express VI in the first frame of the sequence structure. Then, place the LabVIEW code you want to measure in the second frame of the sequence structure. Finally, place the other Tick Count Express VI in the last frame of the sequence structure. You then can calculate the difference between the results of the two Tick Count Express VIs to determine the execution time. Subtract one from the result of the calculation to compensate for the execution time of the Tick Count Express VI.

The Tick Count Express VI tracks time using an internal counter. The internal counter for each Tick Count Express VI on the same block diagram shares the same start time. Therefore, every Tick Count Express VI using the same values for the Counter Units and Size of Internal Counter options tracks the same time. For example, if you call two Tick Count Express VIs that use the same Configure Tick Count options at the same time, they return the same Tick Count value.

The Tick Count Express VI returns an integer value in Counter Units. The Tick Count value cannot represent any fractional time periods that occur when Counter Units is configured for uSec or mSec. Configuring Counter Units for uSec or mSec may result in timing measurements with an accuracy of ±1 Counter Unit value. For example, you can configure Tick Count Express VIs to measure time in milliseconds. If the first Tick Count Express VI executes at 47.9 milliseconds, Tick Count returns a value of 47. If the second Tick Count Express VI executes at 53.2 milliseconds, Tick Count returns a value of 53. Although this example has a 5.3 millisecond delay, the difference between the returned values is 6 milliseconds.

E. Benchmarking Loop Periods

Using tick counts is the best way to calculate loop periods where one tick equals one clock cycle of the timebase (default 40 MHz on most FPGA targets). To measure the number of ticks between one iteration and the next, simply use the Tick Count VI with a shift register.

Place the Tick Count VI in the desired loop and wire the output to a shift register. Then, compare the prior iterations tick count to the current iterations tick count by subtracting the difference between the two. On the output of the subtract function, wire an indicator to see how much time elapsed between iterations.

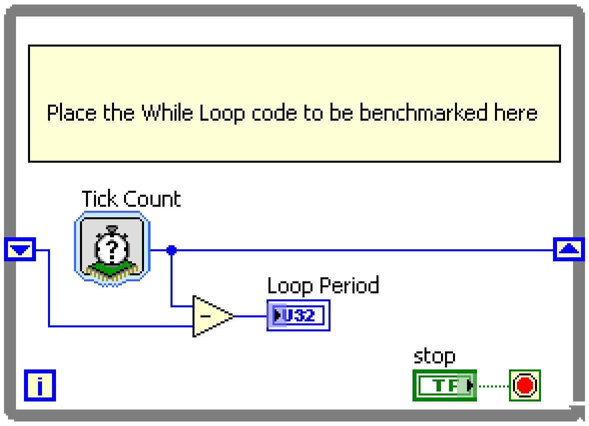


Figure 5-6. Benchmarking Loop Period with Tick Count Express VI

Although this technique uses some FPGA space, it is typically used during the development phase and can be deleted later. Because an FPGA runs the code in true parallel, there are no significant timing effects from adding the additional code.

Self-Review: Quiz

1. Match each VI to the definition that best fits.

Loop Timer Express VI	Use to create delays between events
Wait Express VI	Use to benchmark execution speeds
Tick Count Express VI	Use to control loop execution rate

2. What is a potential drawback of using an 8-bit counter instead of a 32-bit counter?
   a. Decreased FPGA resources used
   b. 32-bit clocks are slower
   c. Maximum time the timer can track is not large enough for the application

3. If you place a Loop Timer Express VI inside a While Loop, the Loop Timer Express VI will wait during every iteration of the While Loop.
   a. True
   b. False

4. Adding benchmarking code to a While Loop will not affect the execution speed of the loop.
   a. True
   b. False

Self-Review: Quiz Answers

1. Match each VI to the definition that best fits.

Loop Timer Express VI	**Use to control loop execution rate**
Wait Express VI	**Use to create delays between events**
Tick Count Express VI	**Use to benchmark execution speeds**

2. What is a potential drawback of using an 8-bit counter instead of a 32-bit counter?
   a. Decreased FPGA resources used
   b. 32-bit clocks are slower
   c. Maximum time the timer can track is not large enough for the application

3. If you place a Loop Timer Express VI inside a While Loop, the Loop Timer Express VI will wait during every iteration of the While Loop.
   a. True
   b. False

4. Adding benchmarking code to a While Loop will not affect the execution speed of the loop.
   a. True
   b. False

Notes

6

Data Sharing on FPGA

In this lesson, you learn how to transfer data between multiple loops on your FPGA VI. You examine three data sharing methods: variables, FPGA memory, and FPGA FIFOs. You learn the benefits of each technique and when each should be used.

Topics

A. Parallel Loops

Parallel operations are a very powerful concept in current computer architecture. In a standard processor-based architecture, parallel operations are not truly parallel. In processor-based architectures, programs running on the processor are sliced into many fragments and are interleaved with code fragments of other processes. The operating system then decides which processes are the most important and schedules the fragments of code accordingly.

In an FPGA VI, parallel functions and loops execute in true parallel. The two loops shown in Figure 6-1 execute in parallel at different rates.

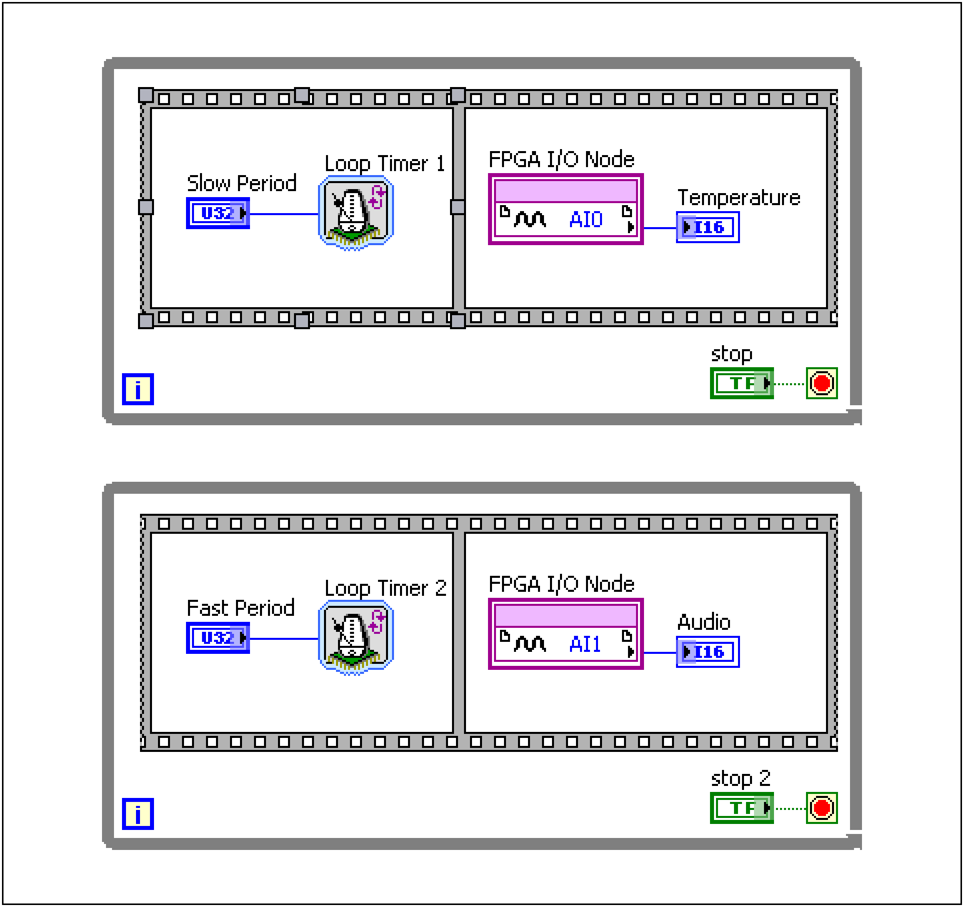


Figure 6-1. Parallel Loops with Different Sampling Rates

Another advantage of running code in parallel is that some sections of code can run faster than other sections. As shown in Figure 6-2, one piece of code in a loop can severely limit the speed of another piece of code.

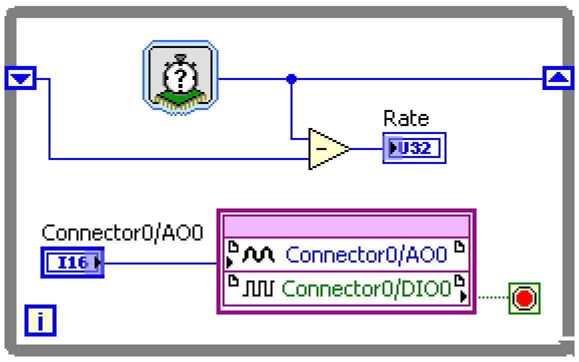


Figure 6-2. Loop Rate Limitation – Same I/O

In this application, the analog output runs 35 times slower than the digital input. This becomes particularly critical if the code were arranged such that the digital line corresponded to an emergency stop switch and the recognition of the response had to happen immediately.

In Figure 6-3, the code from the previous figure is divided into two parallel operations.

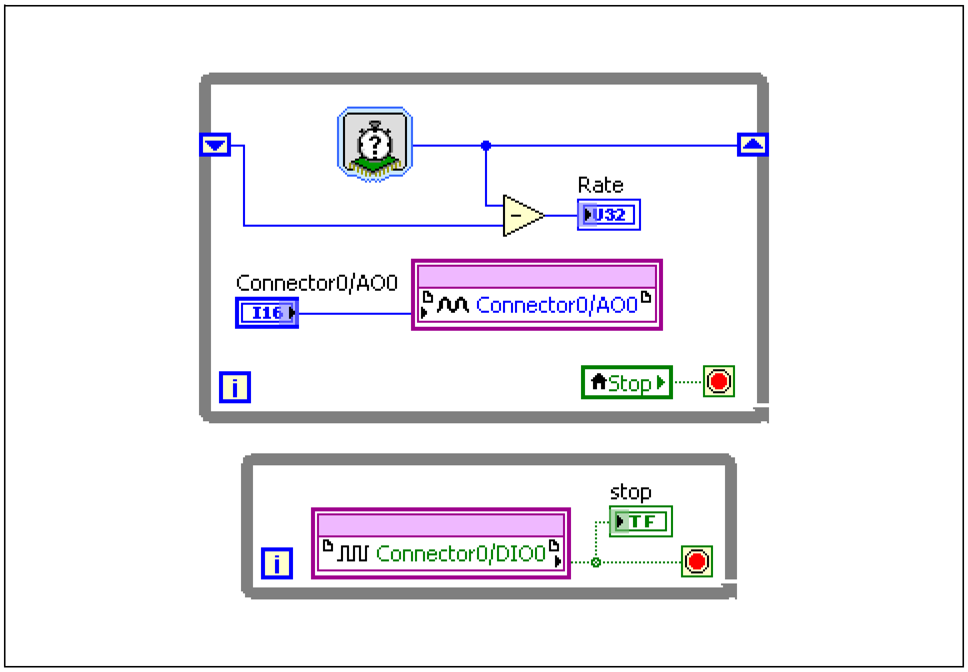


Figure 6-3. Loop Rate Limitation – Separate I/O

The top loop runs at the same analog output-limited rate of about 1 MHz. However, the code in the bottom loop can run at a rate of 10 MHz, or 10 times faster. Often, when code runs too slowly you can separate the code into parallel operations to prevent unrelated processes from interfering with each other.

By default, parallel LabVIEW loops execute independently and in parallel when implemented in FPGA hardware. In order to time loops off each other, you can affect the execution order or sequence of events so that each task is performed when you want. A structure commonly used for controlling the program flow and synchronizing parallel loops are FPGA FIFOs.

You can use FIFOs, memory, or local variables to transfer data between parallel loops.

B. Shared Resources

Some FPGA applications contain shared resources that are accessed by multiple objects in an FPGA VI, such as functions or subVIs. The following components are considered to be shared resources:

- Analog inputs on most targets
- Analog outputs on most targets
- Memory and FIFOs
- Non-reentrant VIs
- Local variables
- Digital outputs on most targets
- Digital inputs on some targets (such as C Series modules with more than 8 inputs)

Sharing resources between two different tasks or loops may affect the deterministic execution of the tasks, even when they are in parallel. Resources that may only be accessed by one task at a time can prevent code from executing independently and are referred to as shared resources.

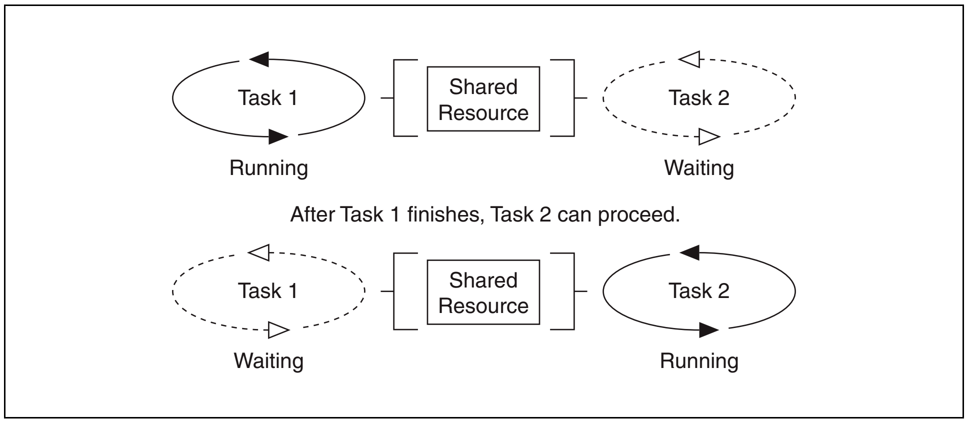


Figure 6-4. Shared Resources

As shown in Figure 6-4, Task 1 runs and accesses a shared resource that can be accessed by only one task at a time. Task 2 must wait until Task 1 releases the resource before accessing it. After Task 1 finishes, Task 2 can access the shared resource and execute while Task 1 waits for the shared resource to be free.

Using shared resources causes jitter. To maximize the determinism of our VI, we must avoid shared resources

Figure 6-5 contains a shared resource. Can you locate it?

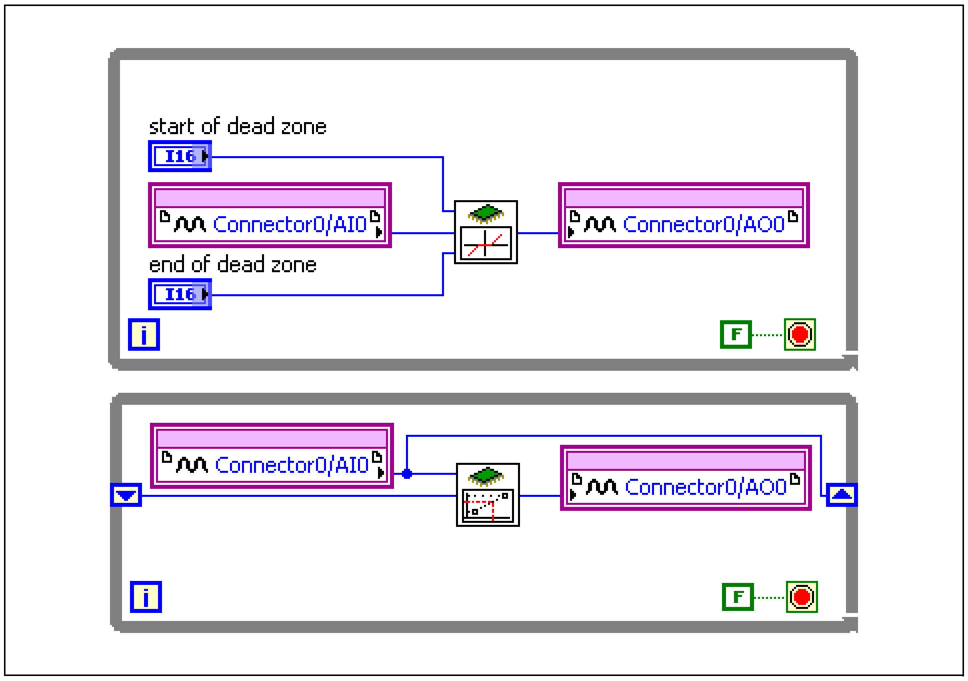


Figure 6-5. Shared Resources Example

If you guessed the analog input function, you are correct. Analog inputs from the same channel are shared resources. While one loop reads from the analog input channel, the other loop must wait until the top loop finishes reading from that channel before it can execute.

C. Variables

In LabVIEW, the flow of data, rather than the sequential order of commands, determines the execution order of block diagram elements. Therefore, you can create block diagrams that have simultaneous operations.

However, if you use wires to pass data between parallel block diagrams, they no longer operate in parallel. Parallel block diagrams can be two parallel loops on the same block diagram without any dataflow dependency, or two separate VIs that are called at the same time.

The block diagram in Figure 6-6 does not run the two loops in parallel because of the wire between the two subVIs.

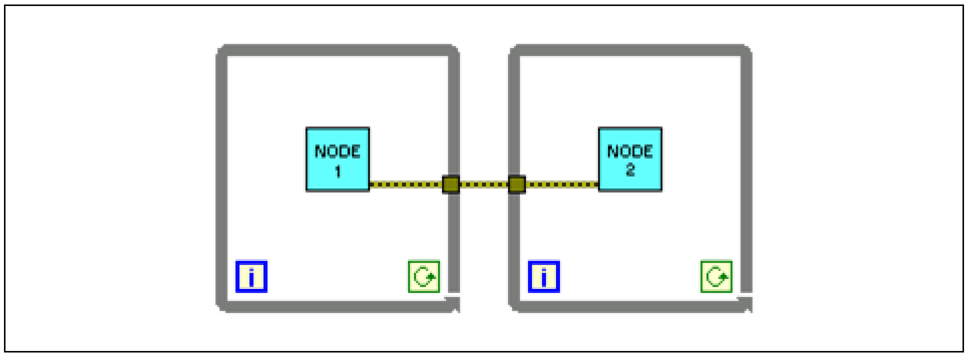


Figure 6-6. Data Dependency Imposed by Wire

The wire creates a data dependency because the second loop does not start until the first loop finishes and passes the data through its tunnel. To make the two loops run concurrently, remove the wire. To pass data between the subVIs, use another technique, such as a variable.

In LabVIEW, variables are block diagram elements that allow you to access or store data in another location. The actual location of the data varies depending on the type of the variable. Local variables store data in front panel controls and indicators. Global variables and single-process shared variables store data in special repositories that you can access from multiple VIs. Functional global variables store data in While Loop shift registers. Regardless of where the variable stores data, all variables allow you to circumvent normal dataflow by passing data from one place to another without connecting the two places with a wire. For this reason, variables are useful in parallel architectures, but also have certain drawbacks, such as race conditions.

Using Variables in a Single VI

In LabVIEW, you read data from or write data to a front panel object using its block diagram terminal. However, a front panel object has only one block diagram terminal, and your application may need to access the data in that terminal from more than one location.

Local and global variables pass information between locations that you cannot connect with a wire in the application. Local variables transfer data within a single VI. Use local variables to access front panel objects from more than one location in a single VI. Use global variables to access and pass data among several VIs.

Creating Local Variables

Right-click an existing front panel object or block diagram terminal and select **Create»Local Variable** from the shortcut menu to create a local variable. A local variable icon for the object appears on the block diagram.

You also can select a local variable from the Functions palette and place it on the block diagram. The local variable node is not yet associated with a control or indicator.

To associate a local variable with a control or indicator, right-click the local variable node and select **Select Item** from the shortcut menu. The expanded shortcut menu lists all the front panel objects that have owned labels.

LabVIEW uses owned labels to associate local variables with front panel objects, so label the front panel controls and indicators with descriptive owned labels.

Reading and Writing to Variables

After you create a local or global variable, you can read data from a variable or write data to it. By default, a new variable receives data. This kind of variable works as an indicator and is a write local or global. When you write new data to the local or global variable, the associated front panel control or indicator updates to the new data.

You also can configure a variable to behave as a data source, or a read local or global. Right-click the variable and select **Change To Read** from the shortcut menu to configure the variable to behave as a control. When this node executes, the VI reads the data in the associated front panel control or indicator.

To change the variable to receive data from the block diagram rather than provide data to it, right-click the variable and select **Change To Write** from the shortcut menu.

On the block diagram, you can distinguish read local variables or global variables from write local variables or global variables the same way you distinguish controls from indicators. A read local variable or global variable has a thick border similar to a control. A write local variable or global variable has a thin border similar to an indicator.

Local Variable Example

The front panel for this example contains a single switch that stops the data generation displayed on two graphs. On the block diagram, the data for each chart is generated within an individual While Loop to allow for separate timing of each loop. In this example, the two loops must share the switch to stop both loops at the same time.

For both charts to update as expected, the While Loops must operate in parallel. Connecting a wire between While Loops to pass the switch data makes the While Loops execute serially, rather than in parallel. Figure 6-7 shows a block diagram of this VI using a local variable to pass the switch data.

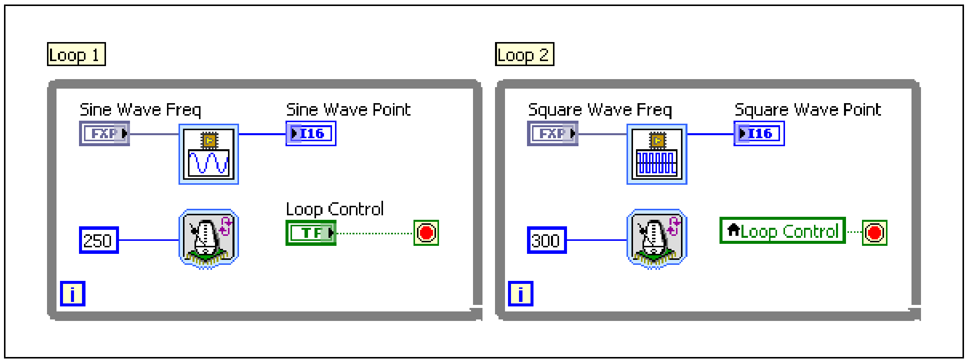


Figure 6-7. Local Variable Used to Stop Parallel Loops

Loop 2 reads the local variable associated with the switch. When you set the switch to False on the front panel, the switch terminal in Loop 1 writes a False value to the conditional terminal in Loop 1. Loop 2 reads the **Loop Control** local variable and writes a False to the Loop 2 conditional terminal. Thus, the loops run in parallel and terminate simultaneously when you turn off the single front panel switch.

With a local variable, you can write to or read from a control or indicator on the front panel. Writing to a local variable is similar to passing data to any other terminal. However, with a local variable you can write to it even if it is a control or read from it even if it is an indicator. In effect, with a local variable, you can access a front panel object as both an input and an output.

For example, if the user interface requires users to log in, you can clear the **`Login`** and **`Password`** prompts each time a new user logs in. Use a local variable to read from the login and password string controls when a user logs in and to write empty strings to these controls when the user logs out.

D. Memory Nodes

You can use memory items to store data in the FPGA block memory. Memory items reference the block memory on the FPGA target in multiples of 2 kilobytes. Each memory item references a separate address or block of addresses, and you can use memory items to access all available memory on the FPGA. Use memory items if you need random access to the data you store.

Memory items do not consume logic resources on the FPGA because they do not include the extra logic necessary to ensure data integrity across clock domains. If you must transfer data across clock domains, you can use local or global variables or FIFOs configured with Block Memory as the Implementation. For more information about FIFOs, refer to the *FPGA FIFOs* section in this lesson.

Each memory address in a memory item stores only the latest value. If you write to a memory address *N* times before reading from that address, the *N–1* values preceding the latest value are lost. If you do not use every data value you acquire, memory items are a good choice because you do not need to write extra code to discard unnecessary values.

Target-Scoped Memory Items

Target-scoped memory items are available within any FPGA VI under the same target in the Project Explorer window. If you send an FPGA VI with a target-scoped memory item to another user, you must also send the project. Otherwise, the FPGA VI is broken.

To create a target-scoped memory item, which you can access in the entire FPGA VI hierarchy, right-click the FPGA target in the Project Explorer window and select **New»Memory** from the shortcut menu, as shown in Figure 6-8. The Memory Properties dialog box appears.

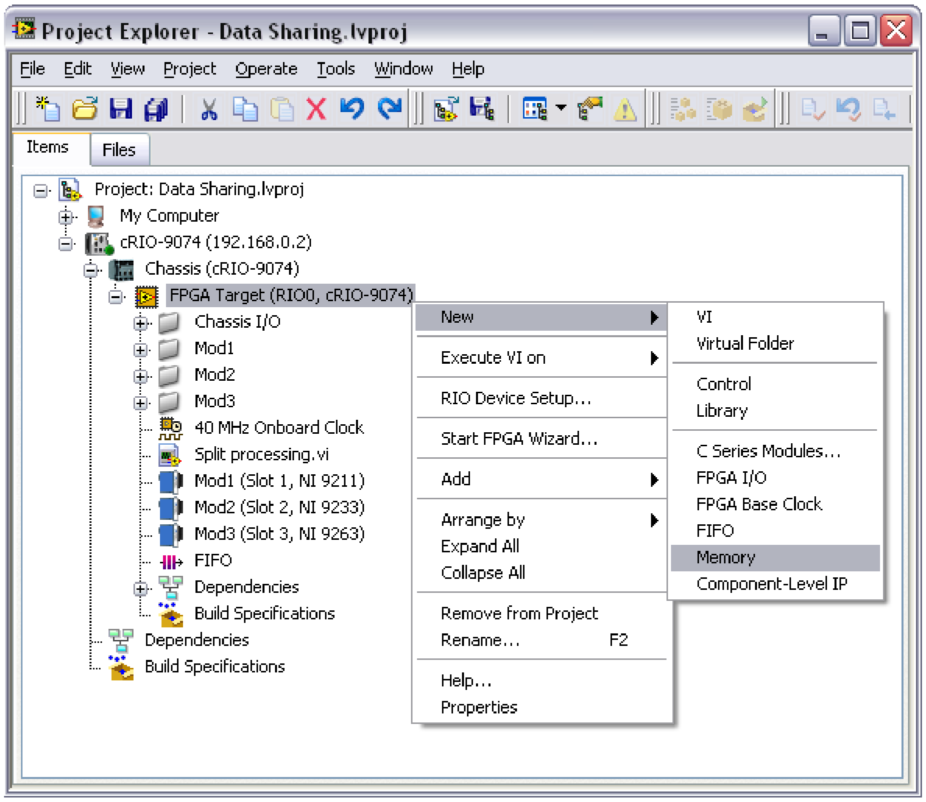


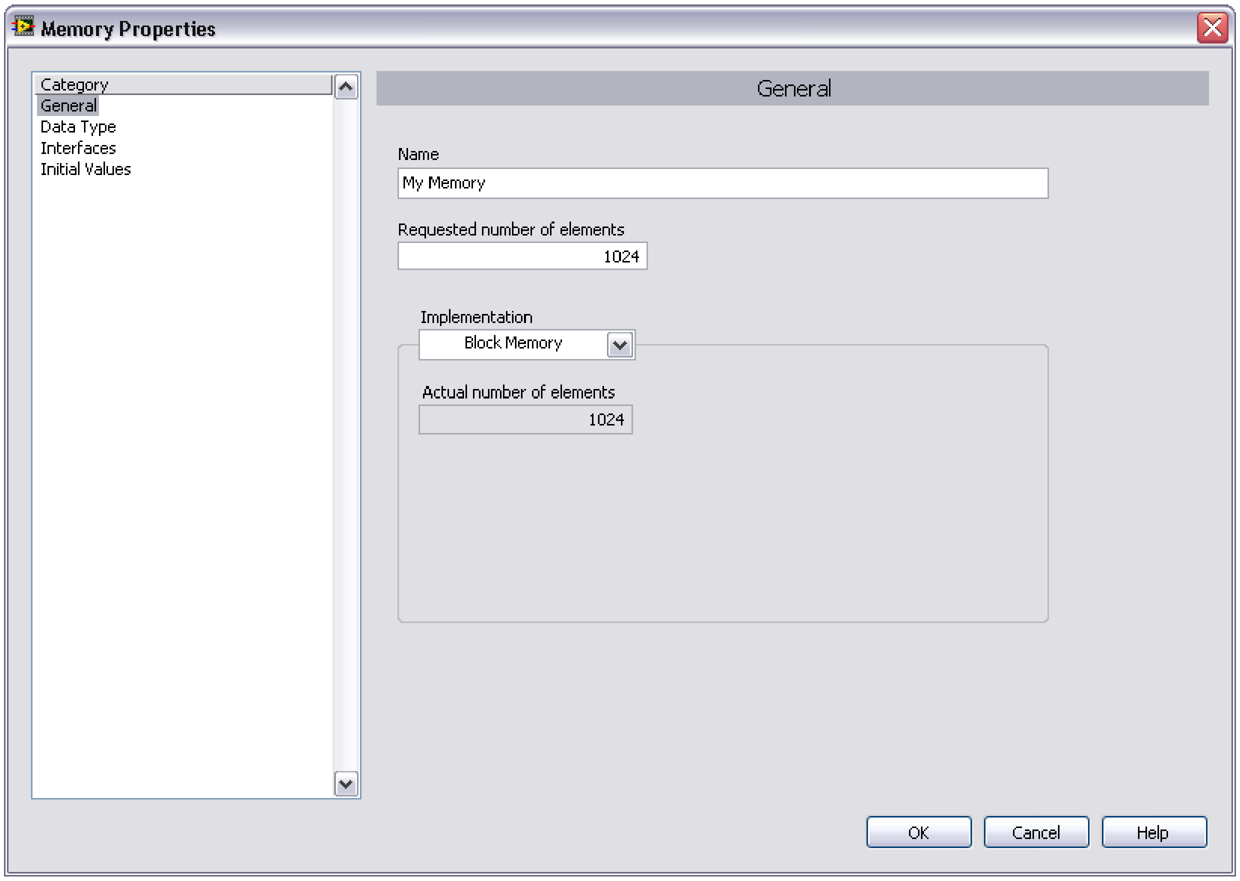
Figure 6-8. Creating a Target-Scoped Memory Item

Memory Properties Dialog Box

Configure the memory item in the Memory Properties dialog box.

General Page

In the Memory Properties dialog box, select **General** from the Category list to display the page shown in Figure 6-9, where you can edit properties for memory items.



Name: My Memory
Requested number of elements: 1024
Implementation: Block Memory
Actual number of elements: 1024

Figure 6-9. Memory Properties Dialog Box – General Page

This dialog includes the following components:

- **Name**—Specifies the name of the memory item that appears in the Project Explorer window or in the VI-Defined Memory Configuration node. The name also appears in the Memory Method Node on the block diagram.
- **Requested number of elements**—Specifies the number of elements you want to hold in the memory item. The actual memory usage, in bytes, depends on the number of elements and data type you specify.

- **Implementation—**Specifies how the FPGA stores this memory item. Contains the following options:
  - **Block Memory—**Stores the data using embedded blocks of memory. Xilinx literature describes this implementation as block RAM or BRAM. Memory items using embedded block memory take one clock cycle to execute. Use block memory in the following situations:
    - In a single-cycle Timed Loop, when you do not need to access this memory during the same cycle as the one in which you give the address.
    - When the amount of memory you need is large.
    - When you do not have enough free logic resources available on the FPGA.

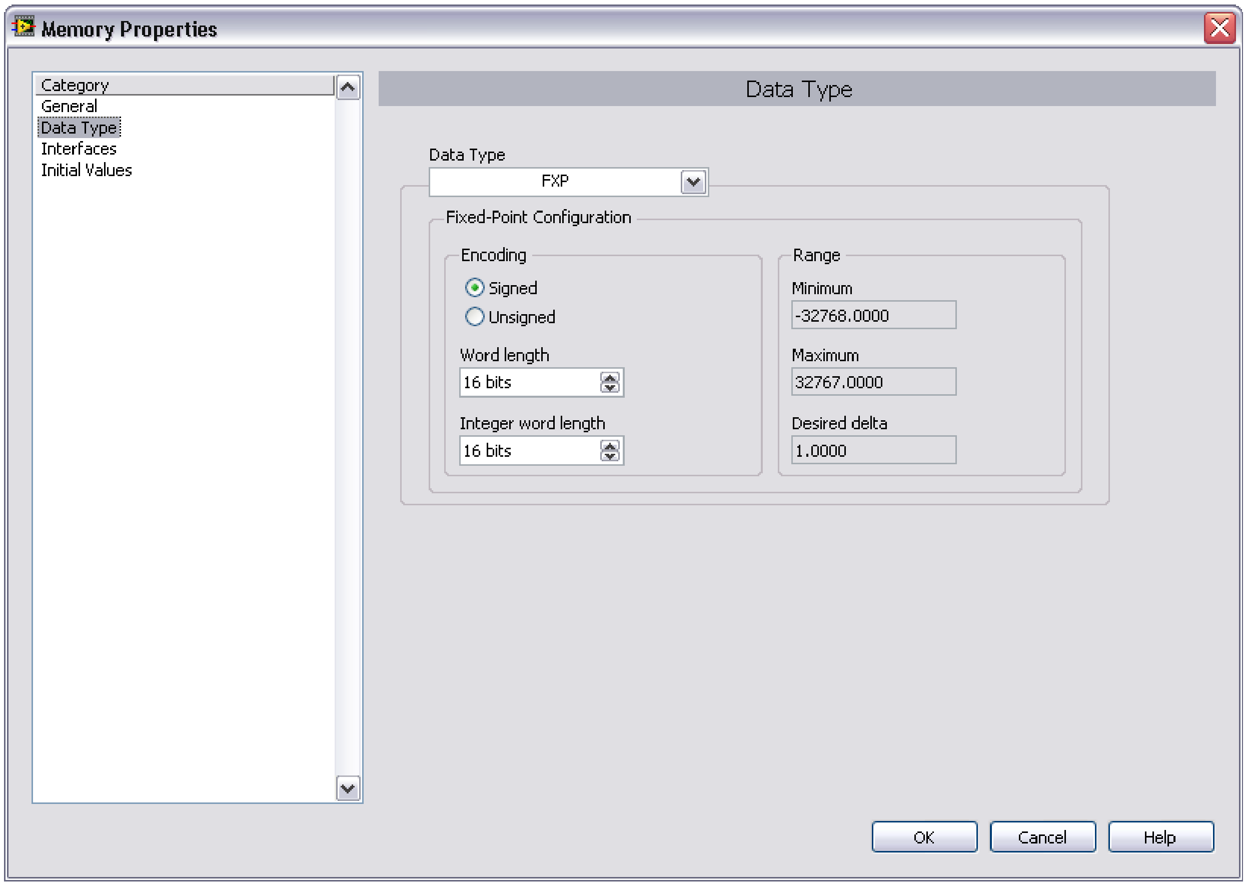
      This option contains the following component:
      - **Actual number of elements—**Returns the configured number of elements. Sometimes the requested number of elements is not compatible with the memory configuration. In these case, the **Actual Number of Elements** is coerced to a compatible number.
  - **Look-Up Table—**Stores the memory item in look-up tables available on the FPGA. This storage consumes logic resources that the FPGA uses for other logical operations, such as addition and subtraction. Xilinx literature describes this implementation as distributed RAM or LUT RAM. Use look-up tables in the following situations:
    - You are accessing this memory in a single-cycle Timed Loop and need to read data from the memory item during the same cycle as the one in which you give the address.
    - The amount of memory you need is smaller than the minimum amount of embedded block memory on the FPGA.
    - You do not have enough free embedded block memory on the FPGA.

      This option contains the following component:
      - **Actual number of elements—**Returns the configured number of elements. Sometimes the requested number of elements is not compatible with the memory configuration. In these case, the Actual Number of Elements is coerced to a compatible number.

- **DRAM**—Stores the memory item in DRAM available on the FPGA. Not all hardware supports using DRAM for memory. This option contains the following components:
  - **Actual number of elements**—Returns the configured number of elements. Sometimes the requested number of elements is not compatible with the memory configuration. In these case, the Actual Number of Elements is coerced to a compatible number.
  - **Maximum outstanding requests for data**—Specifies the number of maximum requests for data the application allows to be outstanding.
  - **DRAM bank**—Specifies which DRAM bank to use.
    - **Allocated here:**—Indicates the amount of memory allocated in this memory item.
    - **Allocated elsewhere:**—Indicates how much memory is allocated in other items.
    - **Available:**—Indicates how much memory is still available in the bank.
    - **Total physical size:**—Indicates the total size of the bank.

Data Type Page

In the Memory Properties dialog box, select **Data Type** from the Category list to display the page shown in Figure 6-10.



Data Type: FXP

Fixed-Point Configuration

Encoding: (● Signed) ○ Unsigned

Word length: 16 bits

Integer word length: 16 bits

Range

Minimum: -32768.0000

Maximum: 32767.0000

Desired delta: 1.0000

Figure 6-10. Memory Properties Dialog Box – Data Type Page

This page includes the following components:

- **Data Type**—Specifies the data type of the data in the FIFO or memory. You can select a fixed-point data type, a Boolean data type, or an 8-, 16-, 32-, or 64-bit signed or unsigned integer data type. You also can select a custom control as the data type. If you select FXP, you must configure the data type in the Fixed-Point Configuration section.
  - **Fixed-Point Configuration**—Sets the configuration settings for fixed-point data. Set Data Type to FXP to enable the fixed-point settings. You configure the Encoding settings, and LabVIEW automatically determines the Range settings.

Note Fixed-point data type FIFOs do not include an overflow bit when transferring data. If you need to transfer the overflow bit, you may use a separate FIFO. You also may specify a wider data type for the FIFO. Then you can manipulate the data to add the overflow bit when writing to the FIFO and subtract the overflow bit when reading from the FIFO.

- **Encoding**—Sets the binary encoding settings for a fixed-point value.
  - **Signed**—Sets whether the fixed-point value is signed.
  - **Unsigned**—Sets whether the fixed-point value is unsigned.
  - **Word length**—Sets the number of bits that LabVIEW uses to represent all the possible fixed-point values.
  - **Integer word length**—Sets the number of integer bits, or the number of bits to shift the binary point to reach the most significant bit, for all the possible fixed-point values. Integer word length can be positive or negative.
- **Range**—Indicates the range for a fixed-point value.

Note Range values display the values in double-precision floating-point representation, so the precision of the display at Maximum, Minimum, and Desired delta might not be exact in terms of fixed-point representation.

  - **Minimum**—Indicates the minimum value for the fixed-point data range.
  - **Maximum**—Indicates the maximum value for the fixed-point data range.
  - **Desired delta**—Indicates the maximum distance between any two sequential numbers in the fixed-point data range.

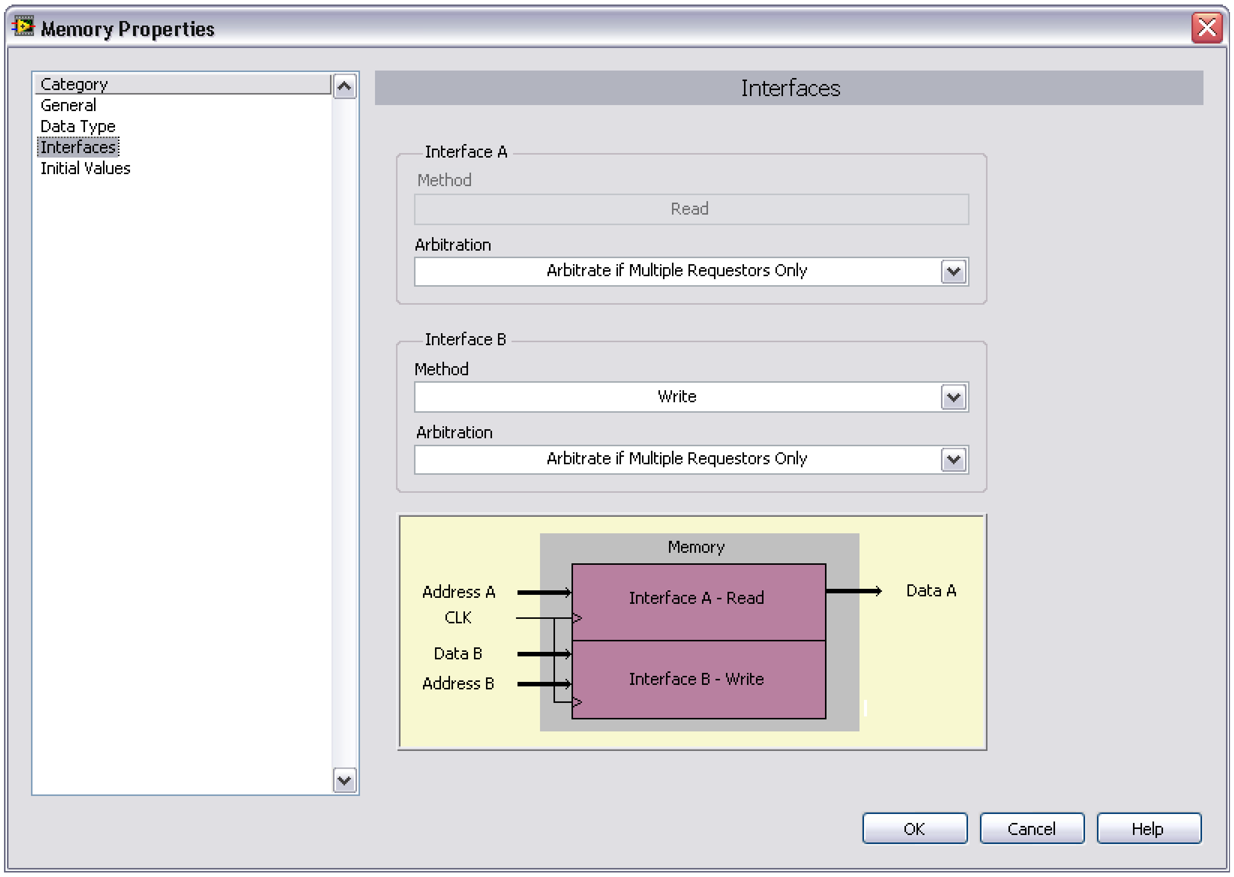
– **Select Control**—Opens a dialog box in which you can navigate to the custom control you want to use. This button appears only when you select **Custom control** in the Data Type pull-down menu.

- **Maximum Data Width for DRAM**—Indicates the physical port width of the DRAM. The data type you select for the DRAM memory cannot be wider than the data width.

  Make sure the sum of the data widths of all items in a custom control is not greater than the Maximum Data Width for DRAM.

Interfaces Page

In the Memory Properties dialog box, select **Interfaces** from the Category list to display the page shown in Figure 6-11.



Interface A
Method: Read
Arbitration: Arbitrate if Multiple Requestors Only

Interface B
Method: Write
Arbitration: Arbitrate if Multiple Requestors Only

Figure 6-11. Memory Properties Dialog Box – Interfaces Page

Use this page to configure the arbitration options and to specify read and/or write ports for the memory item.

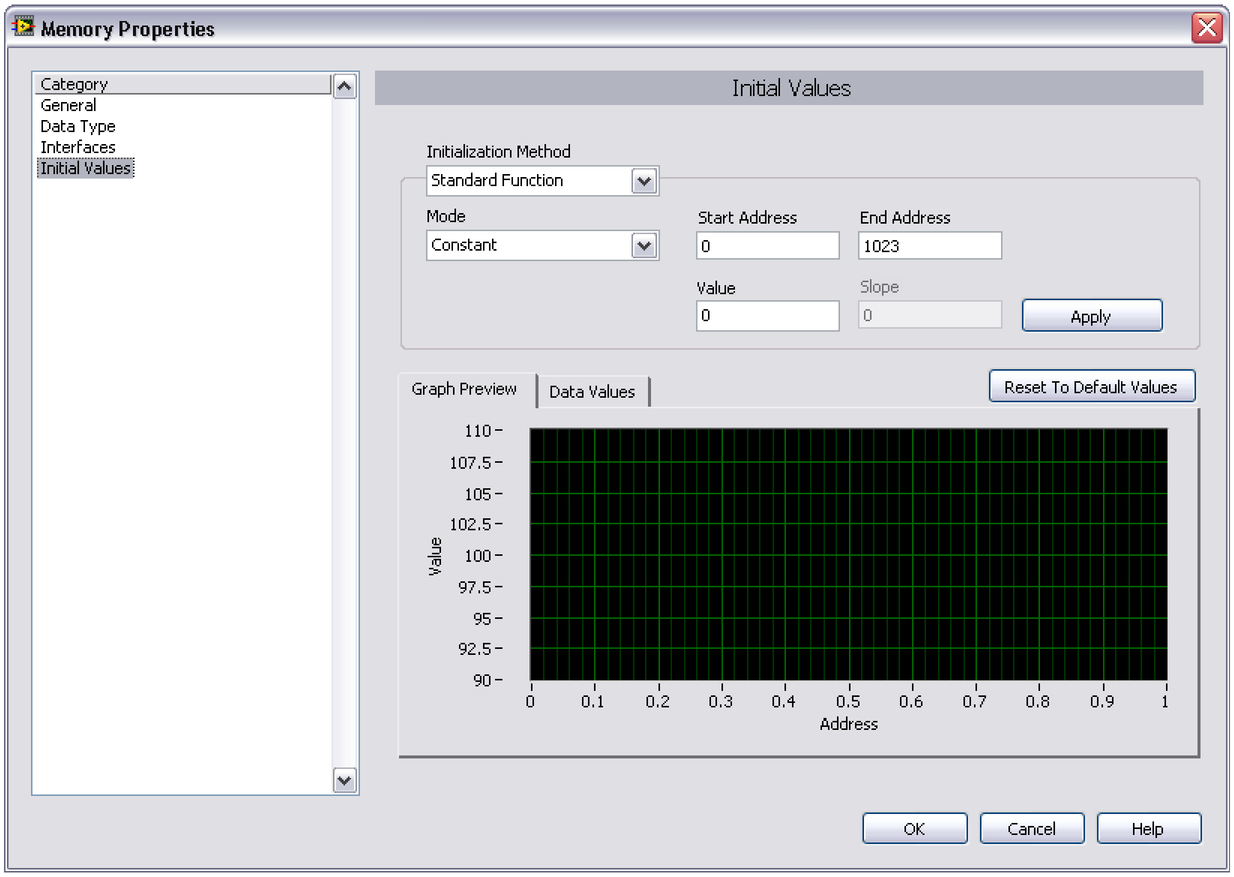
This page includes the following components:

- **Interface A**—Specifies the type of arbitration for reading from Interface A.
  - **Method**—Indicates the type of memory access for the interface. Interface A is always read-only.
  - **Arbitration**—Sets the type of arbitration for the memory interface. Select **Arbitrate if Multiple Requestors Only** or **Never Arbitrate** if you use the Memory Method Node configured for Interface A in a single-cycle Timed Loop.

- **Interface B**—Specifies the method and type of arbitration for Interface B.
  - **Method**—Specifies the type of memory access for the interface. Interface B has write access by default. The read method for Interface B is not compatible with custom data types. If you select a custom data type in the Data Type page, the Method for Interface B is dim.
  - **Arbitration**—Sets the type of arbitration for the memory interface. Select **Arbitrate if Multiple Requestors Only** or **Never Arbitrate** if you use the Memory Method Node configured for Interface B in a single-cycle Timed Loop.
- **Memory Schematic Diagram**—Reflects the configuration of the memory block.

Initial Values Page

In the Memory Properties dialog box, select **Initial Values** from the Category list to display the page shown in Figure 6-12.


Memory Properties

Category: General, Data Type, Interfaces, Initial Values

Initial Values

Initialization Method: Standard Function

Mode: Constant

Start Address: 0

End Address: 1023

Value: 0

Slope: 0

Figure 6-12. Memory Properties Dialog Box – Initial Values Page

Use this page to initialize the memory item you create in the Memory Properties dialog box. You can edit data elements individually, define linear segments, use predefined functions, or call an initialization VI you create.

This page includes the following components:

- **Initialization Method**—Specifies the method you want to use to initialize the memory item. Select **Standard Function** to populate the memory item with constants, straight lines, sine waves, or cosine waves. Select **Initialization VI** to populate the memory item with an array that an initialization VI creates.
  - **Standard Function**—Contains the following options:
    - **Mode**—Specifies the contents with which LabVIEW populates the address interval you select. You can select from the following values:
      - **Constant**—Populates using the constant you specify in **Value**.
      - **Linear**—Populates using a linear segment computed from the values you specify in **Starting Value** and **Slope**.
      - **Sine**—Populates using a full-scale sine wave that you specify in **Number of Cycles**. This option is available only for signed integers and signed fixed-point numbers.
      - **Cosine**—Populates using a full-scale cosine wave that you specify in **Number of Cycles**. This option is available only for signed integers and signed fixed-point numbers.
    - **Start Address**—Specifies the low end of the memory item interval you want to populate.
    - **End Address**—Specifies the high end of the memory item interval you want to populate.
    - **Value**—Specifies the constant LabVIEW enters in the memory item if you select **Constant** from the Mode pull-down menu.
    - **Starting Value**—Specifies the first value LabVIEW enters in the memory item if you select **Linear** from the Mode pull-down menu.
    - **Slope**—Specifies the slope of the line LabVIEW enters in the memory item if you select **Linear** from the Mode pull-down menu.
    - **Number of Cycles**—Specifies the number of cycles LabVIEW enters in the memory item if you select **Sine** or **Cosine** from the Mode pull-down menu.

        - **Apply**—Displays the initial values in the Graph Preview and Data Values tabs. Clicking the **Apply** button does not save the initialization data. You must click the **OK** button to save the initialization data.
    - **Initialization VI**—Contains the following options:
        - **VI Path**—Specifies the path to the initialization VI.
        - **New VI from Template**—Creates an instance of a template VI, saves the VI to the location you specify in the Name the New Initialization VI dialog box, and opens the VI. You must close the Memory Properties dialog box to edit the VI.
        - **Open VI**—Opens the VI you specify in the VI Path field. You must close the Memory Properties dialog box to edit the VI.
        - **Run VI**—Runs the VI you specify in the VI Path field. LabVIEW then imports the output array into the memory item and displays the corresponding values in the Graph Preview and Data Values tabs.
- **Graph Preview**—Displays the current contents of the memory item in a waveform graph.
- **Data Values**—Displays the current contents of the memory item.
- **Reset to Default Values**—Resets the contents of the memory item to the default value. For the Boolean data type, the default is 0 (FALSE). For integer data types, the default is 0. For fixed-point data types, the default is 0 if Maximum is greater than or equal to 0 and Minimum is less than or equal to 0 on the Data Type page of the Memory Properties dialog box. If Maximum is less than 0, the default is equal to Maximum. If Minimum is greater than 0, the default is equal to Minimum.

VI-Defined Memory Item

If you use a VI-defined memory item in a reentrant FPGA VI, LabVIEW creates a separate copy of the memory item for each instance of the VI, which allows you to create reusable subVIs while avoiding resource conflicts. If you send an FPGA VI with a VI-defined memory item to another user, you do not need to send the LabVIEW project because the VI-defined memory item does not include a corresponding item in the Project Explorer window.

To create a VI-defined memory item, place a VI-defined Memory Configuration node on the block diagram, right-click the node, and select **Configure** from the shortcut menu. The Memory Properties dialog box appears, shown in Figure 6-13.

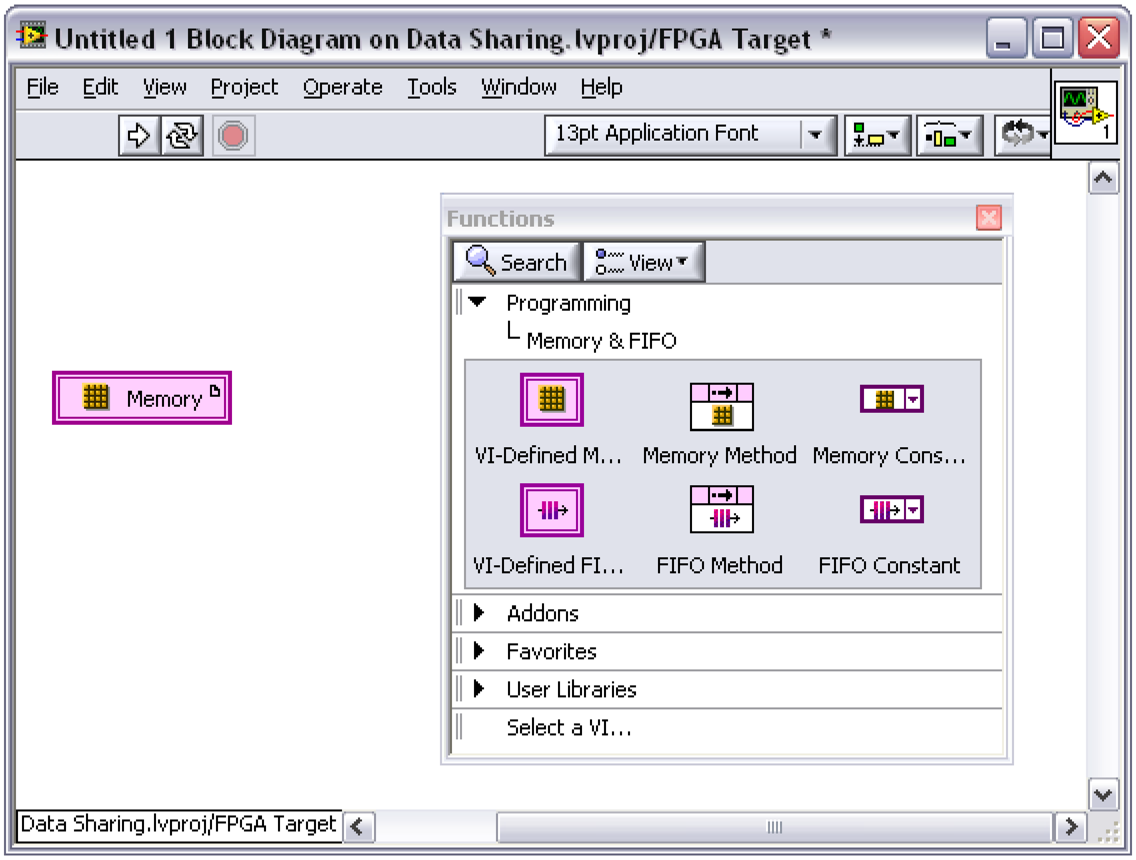


Figure 6-13. Creating a VI-Defined Memory Item

Memory & FIFO Palette

Use the Memory functions listed in Table 6-1 to access memory on an FPGA.

Table 6-1. Memory Objects in the Memory & FIFO Palette

Memory Constant	Specifies a memory item on the block diagram.
Memory Method Node	Invokes a method on a memory block in the FPGA. Use methods to write or read memory blocks.
VI-Defined Memory Configuration	Represents a VI-defined memory item.

The VIs and functions on this palette can return general LabVIEW error codes, specific FPGA Module error codes, or error codes specific to the FPGA target.

Memory Method Node

To use a Memory Method Node on your block diagram, add a Memory Method Node to the block diagram of the FPGA VI from the Memory & FIFO palette. Right-click the Memory Method Node and select **Select Memory»x** from the shortcut menu, where **x** is the name of the memory item in the Project Explorer window or the VI-defined memory item on the block diagram. LabVIEW prepends `VI::` to the name of VI-defined memory items.

You also can click the memory item in the Project Explorer window and drag it onto the block diagram. LabVIEW adds a Memory Method Node, configured for the memory item, to the FPGA VI block diagram. Click the node and select **Write** or **Read** from the shortcut menu to change the configuration of the node.

Write

This method writes data to the memory that has been allocated on the FPGA target. This method has the following terminals:

- **Memory In**—Specifies the FPGA memory. If you leave **Memory In** unwired, you can specify the FPGA memory by right-clicking the Memory Method Node and selecting a memory item from the shortcut menu. Otherwise, you can wire a Memory control, Memory constant, VI-Defined Memory Configuration node, or another Memory Method node to **Memory In**.
- **Address**—Specifies the location of the data in memory on the FPGA target. The valid address range depends on the **Requested Number of Elements** you specify in the Memory Properties dialog box. For example, if you specify a **Requested Number of Elements** of 1000, the valid address range is 0–999. If **Address** exceeds the address range, the Memory Method Node returns an error and does not write the data. Add error terminals so LabVIEW can notify you if **Address** exceeds the address range.
- **Data**—Specifies the data to be written to the memory on the FPGA target. You can directly write to **Data** only from the FPGA VI. If you do not initialize the memory item, the data is undefined.
- **Memory Out**—Returns **Memory In** if **Memory In** is wired. Otherwise, **Memory Out** returns the memory that you specify in the Memory Method Node.

Read

This method reads from memory available on the FPGA target. This method has the following terminals:

- **Memory In**—Specifies the FPGA memory. If you leave **Memory In** unwired, you can specify the FPGA memory by right-clicking the Memory Method Node and selecting a memory item from the shortcut menu. Otherwise, you can wire a Memory control, Memory constant, VI-Defined Memory Configuration node, or another Memory Method node to **Memory In**.
- **Address**—Specifies the location of the data in memory on the FPGA target. The valid address range depends on the **Depth** you specify in the Memory Properties dialog box. For example, if you specify a **Depth** of 1000, the valid address range is 0–999. If **Address** exceeds the address range, the Memory Method Node returns an error and does not write the data. Add error terminals so LabVIEW can notify you if **Address** exceeds the address range.
- **Memory Out**—Returns **Memory In** if **Memory In** is wired. Otherwise, **Memory Out** returns the memory that you specify in the Memory Method Node.
- **Data**—Specifies the data read from memory on the FPGA target. **Data** is directly accessible only from within the FPGA VI. The Data data type is the data type you configure in the Memory Properties dialog box when you create the memory item. If you do not initialize the memory item, the data is undefined.

Note If the memory is configured for both read and write access, you can use this method in conjunction with the Write method. If you configure the memory for dual port read access, the label includes an [A] or [B] to indicate which interface this node will access.

E. Race Conditions

A race condition is a situation in which the timing of events or the scheduling of tasks may unintentionally affect an output or data value. Race conditions are a common problem for programs that execute multiple tasks in parallel and share data between the tasks. Race conditions are difficult to identify and debug. Sharing data between parallel loops is common in FPGA hardware.

Example

Race conditions can occur in a simple program like the one shown in Figure 6-14. What value do you think the Race Value chart will show after the first execution of the Race VI? What value do you think it will show after the second execution?

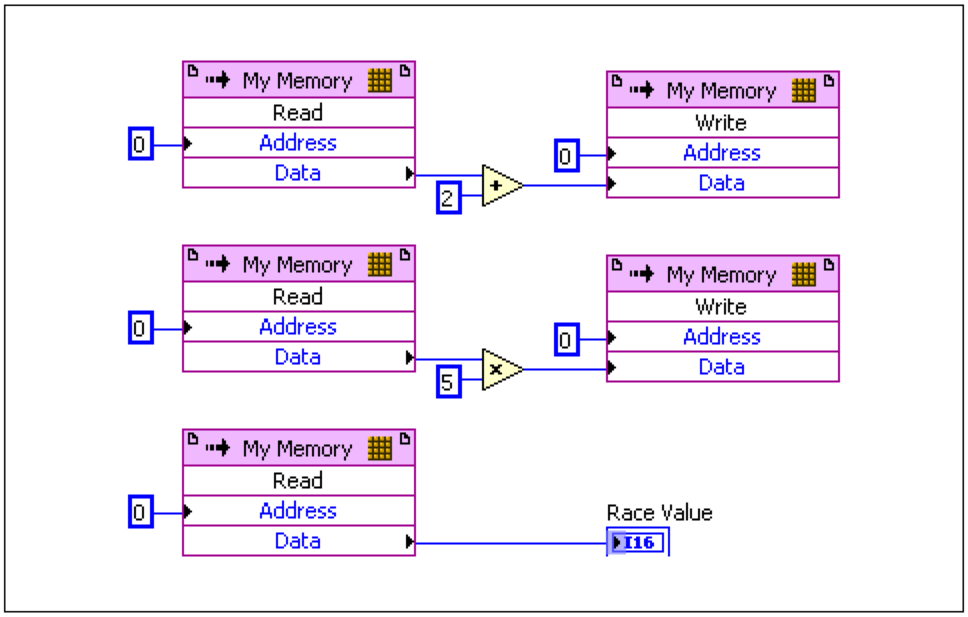


Figure 6-14. Race Condition VI Block Diagram

Compare your prediction with the results of 15 executions, shown in Figure 6-15, where the Race Value consistently increased by 2 in each execution. Is this what you predicted?

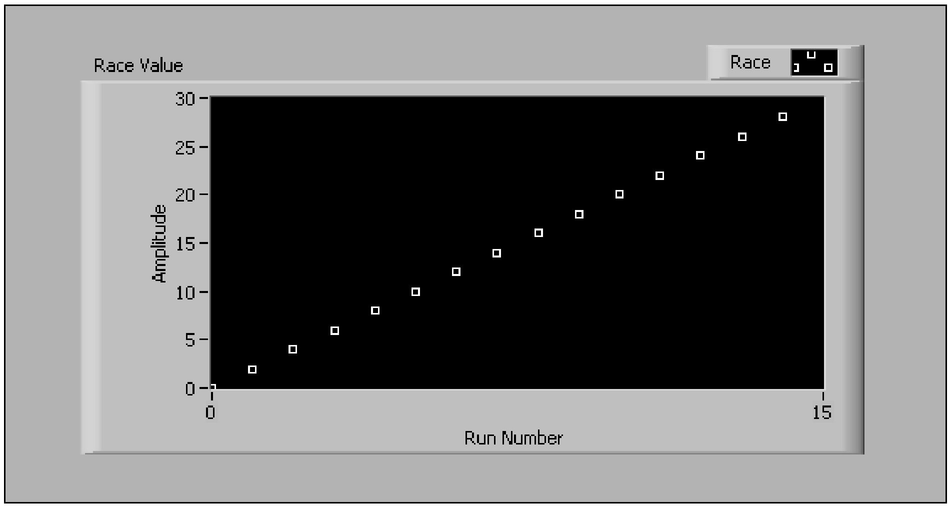


Figure 6-15. Results from 15 Executions of the Race Sequence VI

Single stepping through the program may reveal the following execution sequence:

1. LabVIEW reads the three instances of address 0 of the memory allocated by My Memory, which all return `0` in the first execution.
2. The Multiply function executes and writes `0 (5 × 0)` to address 0 of My Memory.
3. The Add function executes and writes `2` (which replaces `0`) to address 0 of My Memory.
4. The value of address 0 of My Memory, `0`, was read before the Add or Multiply functions executed, so the `0` is written to the chart.
5. In the second execution, LabVIEW again reads address 0 of My Memory, but this time, the value is `2`.
6. The Multiply function executes and writes `10 (5 × 2)` to address 0 of My Memory.
7. The Add function executes and writes `4 (2 + 2)` to address 0 of My Memory, replacing the previous value of 2.
8. The value stored at address 0 of My Memory, 2, was read before the Add or Multiply functions executed, so 2 is written to the chart.

The code consistently repeated the above execution sequence 15 times, but that execution sequence is not guaranteed with each compile.

The Race VI has the following problems:

- It does not control the order of operations. The placement of code in LabVIEW does not control execution order the way text programming does. Data flow controls execution order in LabVIEW.
- There are multiple writers to address 0 of My Memory.
- We cannot tell from the block diagram whether or not address 0 of My Memory was initialized.

Parallel Loops – Shared Data

FPGA VIs commonly use multiple loops as shown in Figure 6-16. This program has two While Loops—one acquires data from an FPGA I/O Node and writes the data to a memory item, the other reads the data from the memory item and displays the data. Inspection of the front panel charts in Figure 6-17 shows that the data transfers without error because the loops run at approximately the same rate. But what happens if the read-and-display loop runs faster than the generate-and-write loop? What happens if it runs more slowly?

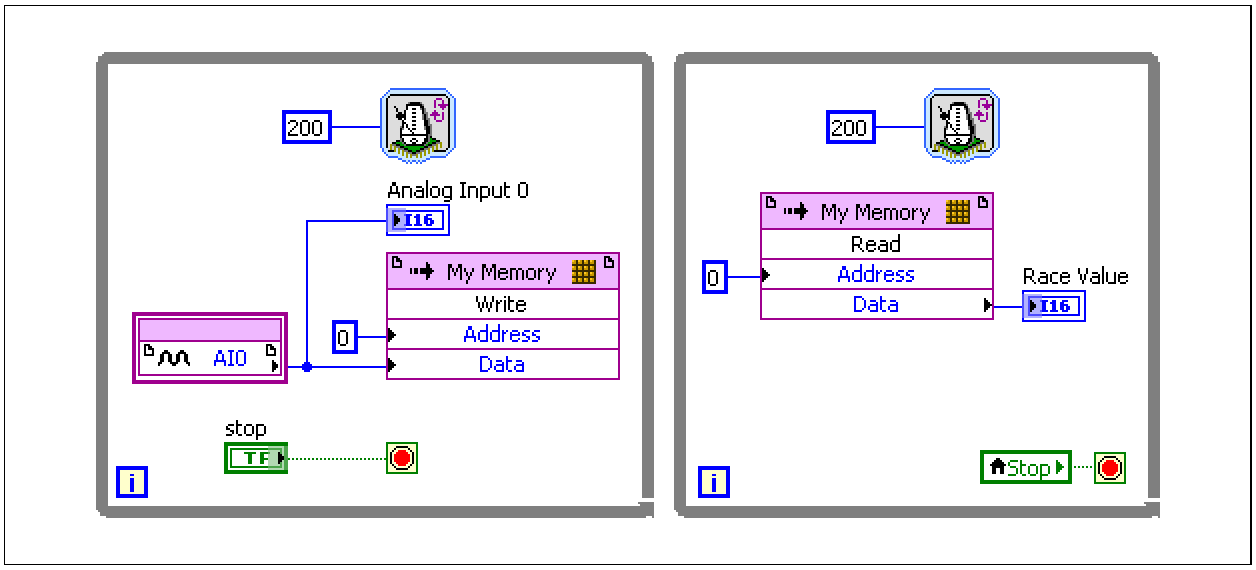


Figure 6-16. Multiple Loops VI with Equal Wait Constants Block Diagram

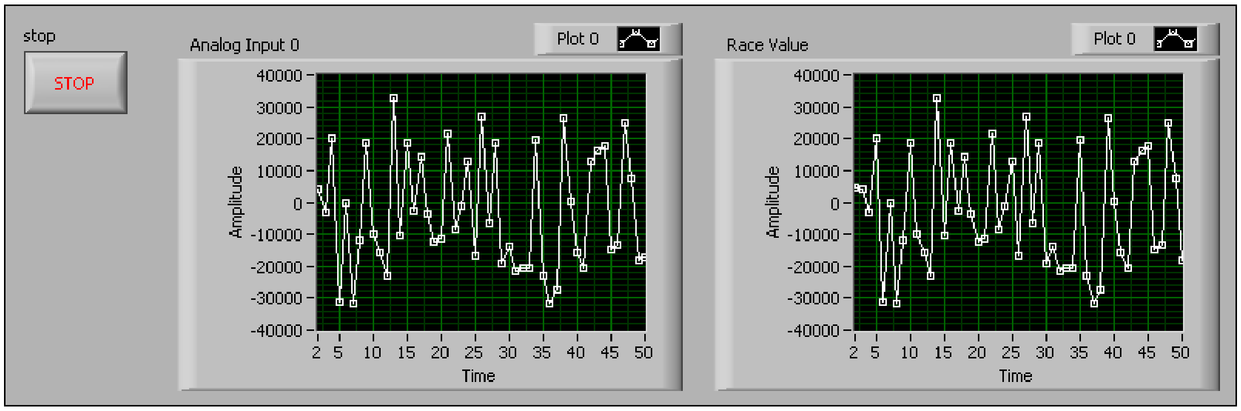


Figure 6-17. Multiple Loops VI with Equal Wait Constants Front Panel

The following two sections describe the results.

Parallel Loops – Oversampling

Figure 6-18 and Figure 6-19 show that if the read-and-display loop runs faster, it reads and displays more data than is actually acquired. The loop reads and displays some data points multiple times. This condition is known as oversampling. Oversampling occurs when acquiring data from slower hardware (analog input signals, thermocouples) and then processing on faster hardware (FPGA). Oversampling is often necessary when trying to capture fast edges, transients and one-time events.

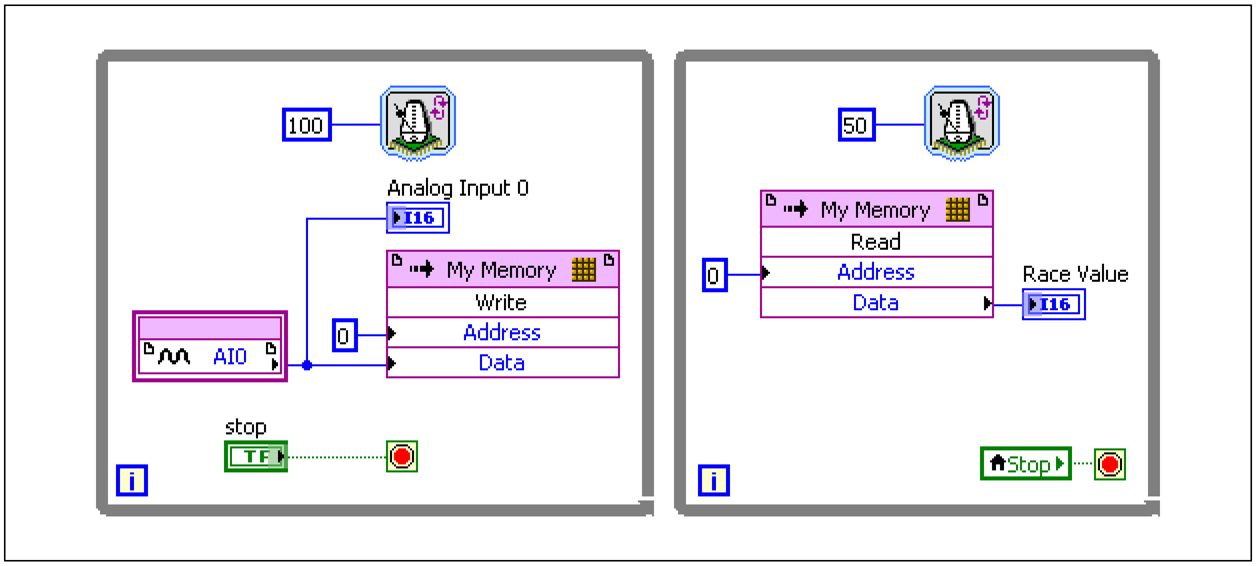


Figure 6-18. Multiple Loops VI with Faster Read-and-Display Loop Block Diagram

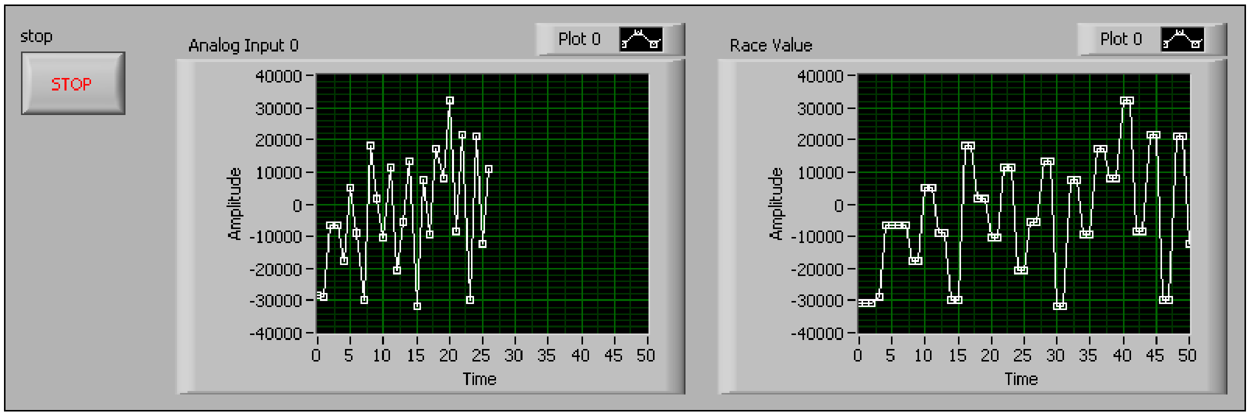


Figure 6-19. Multiple Loops VI with Faster Read-and-Display Loop Front Panel

Parallel Loops – Undersampling

Figure 6-20 and Figure 6-21 show that if the read-and-display loop runs slower, it reads and displays fewer data points than are actually acquired. Some data points are written over and lost before they are read and displayed. This condition is known as undersampling. Undersampling is sampling too slowly for a particular signal of interest.

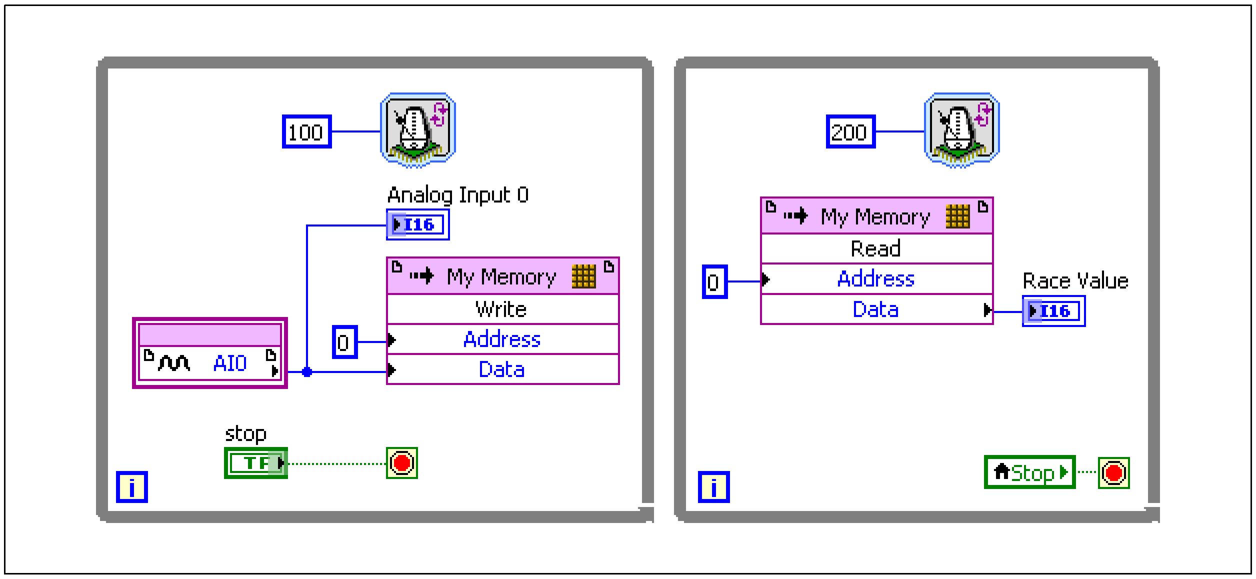


Figure 6-20. Multiple Loops VI with Slower Read-and-Display Loop Block Diagram

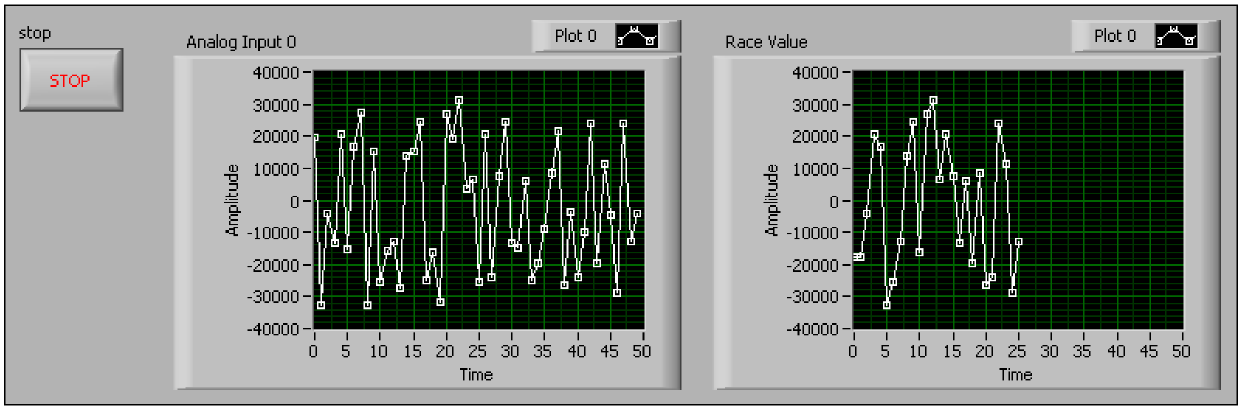


Figure 6-21. Multiple Loops VI with Slower Read-and-Display Loop Front Panel

Avoiding Race Conditions

By default, parallel LabVIEW loops execute independently and in parallel when implemented in FPGA hardware. In order to time loops off of each other, you must affect the execution order or sequence of events so that each task is performed when you want. A structure commonly used for controlling the program flow and synchronizing parallel loops are FPGA FIFOs.

F. FPGA FIFOs

A common buffering technique used with CompactRIO is a FIFO buffer, where the first data item written to memory is the first item read and removed from memory. The *LabVIEW Core 2* course describes a similar structure called a queue.

Usage

An FPGA FIFO functions like a fixed-length queue where multiple data items can be written to and read from memory. Unlike a queue, an FPGA FIFO ensures deterministic behavior by imposing a size restriction on the data, and both the reader and the writer can access the data simultaneously.

The LabVIEW FPGA Module includes the following types of FIFOs:

- **Target-Scoped**—Transfers data to and from sections of code in multiple VIs on the same target with a single FIFO.
- **VI-Defined**—Transfers data to and from multiple loops within a single VI.
- **DMA**—Directly accesses memory to transfer data to RT host VIs and from FPGA target VIs and vice versa.

DMA FIFOs are discussed further in Lesson 9, *DMA Data Transfers*. For now, we will focus on the use of target-scoped and VI-defined FIFOs.

Target-Scoped FIFO

Target-scoped FIFOs appear under an FPGA target in the Project Explorer window and you can access them from any VI running on the FPGA target. Use target-scoped FIFOs to store data you must access from multiple VIs.

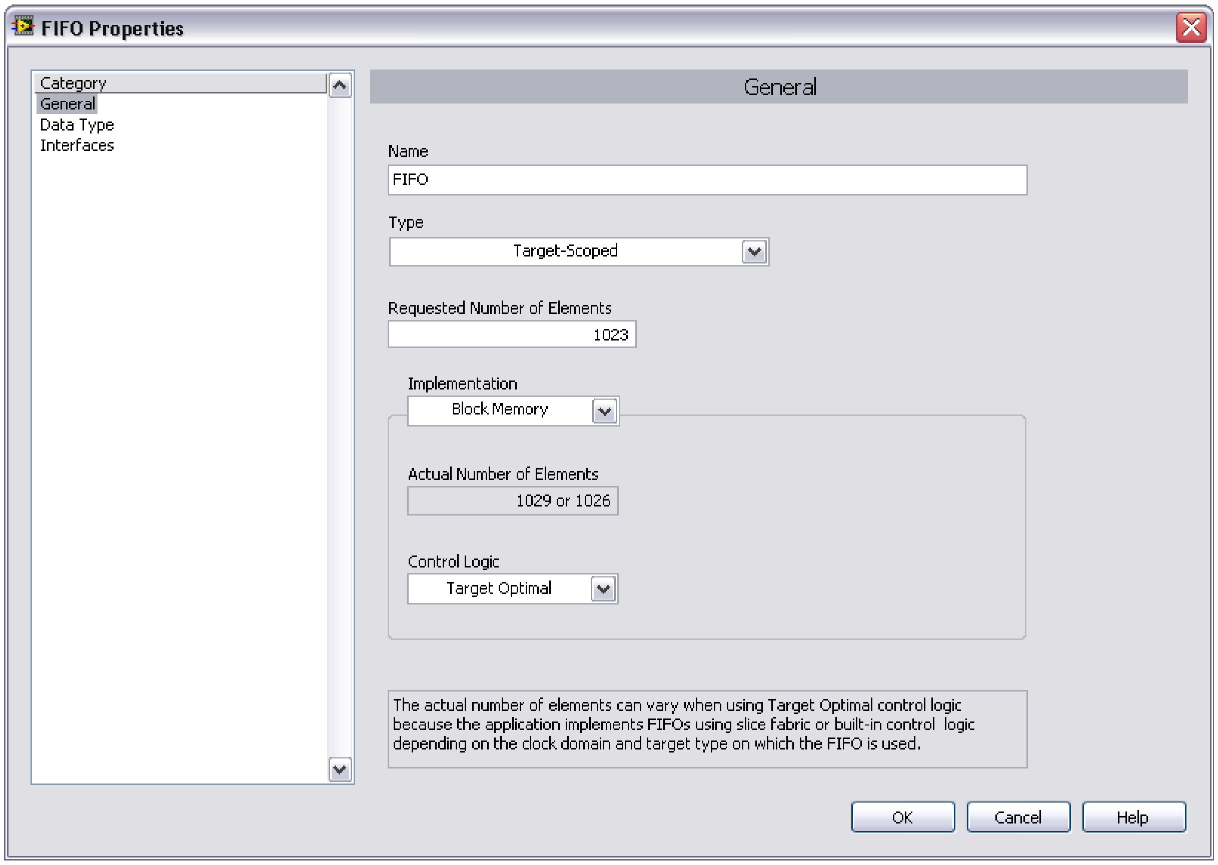
To create a target-scoped FIFO, right-click the FPGA target in the Project Explorer window and select **New»FIFO** to display the FPGA FIFO Properties dialog box.

FIFO Properties Dialog Box

You configure the properties of the FIFO in the FPGA FIFO Properties dialog box.

General Page

In the FIFO Properties dialog box, select **General** from the Category list to display the page shown in Figure 6-22.



Name: FIFO

Type: Target-Scoped

Requested Number of Elements: 1023

Implementation: Block Memory

Actual Number of Elements: 1029 or 1026

Control Logic: Target Optimal

The actual number of elements can vary when using Target Optimal control logic because the application implements FIFOs using slice fabric or built-in control logic depending on the clock domain and target type on which the FIFO is used.

Figure 6-22. FPGA FIFO Properties Dialog Box – General Page

Use this page to edit properties for FIFOs.

This page includes the following components:

- **Name**—Specifies the name of the FIFO that appears in the Project Explorer window or in the VI-Defined FIFO Configuration node. The name also appears in the FIFO Method Node on the block diagram, and you can use it in FIFO name controls and constants to access target-scoped FIFOs.

- **Type—**Specifies the type of FIFO to use.

Note The Type option is not available for VI-defined FIFOs.

  - **Target-Scoped**—FIFOs can transfer data within the FPGA VI as well as between FPGA VIs under the same target in the Project Explorer window.
  - **Host to Target - DMA or Target to Host - DMA**—DMA FIFOs can transfer data between the host VI and target.
  - **Peer to Peer Writer or Peer to Peer Reader**—FIFOs can transfer data using a peer-to-peer stream.
- **Disable on Overflow—**Specifies to disable the peer-to-peer stream when the writer FIFO attempts to write to the stream and fails. This option is only available for peer-to-peer writer FIFOs.
- **Disable on Underflow**—Specifies to disable the peer-to-peer stream when the reader FIFO does not receive data from the stream. This option is only available for peer-to-peer reader FIFOs.
- **Requested Number of Elements**—Specifies the desired number of elements the FIFO can hold. The maximum number of elements the FIFO can hold depends on the Implementation you select and the amount of space available on the FPGA for the Implementation. If the FIFO uses built-in control logic, the maximum number of elements also depends on the data type. The width of the built-in FIFO must be less than or equal to 1024.

  If the FPGA does not have enough space for the Requested Number of Elements you enter, the FPGA VI fails to compile and an error message appears. If you select **DMA - Host to Target** or **DMA - Target to Host** in the Type pull-down menu, Requested Number of Elements specifies the size of the FPGA part of the DMA FIFO.

  If you select **Block Memory** in the Implementation control, restrictions apply to the number of elements the FIFO can hold. Actual Number of Elements indicates the number of elements in the FIFO, which may not be the same as Requested Number of Elements.
- **Implementation**—Specifies the type of storage the FIFO uses on the FPGA. You can specify the implementation only for target-scoped and VI-defined FIFOs. Contains the following options:
  - **Flip-Flops**—Stores the data in flip-flops available on the FPGA and provides the fastest performance. National Instruments recommends using this option for small FIFOs, up to 100 bytes. You cannot use FIFOs with an Implementation of Flip-Flops or Look-Up Table across multiple clock domains.

- **Look-Up Table**—Stores the data in look-up tables available on the FPGA. Xilinx literature describes this implementation as distributed RAM or LUT RAM. National Instruments recommends using this option for FIFOs that are 100–300 bytes. You cannot use FIFOs with an Implementation of Flip-Flops or Look-Up Table across multiple clock domains.

Note You cannot use FIFOs with an Implementation of Flip-Flops or Look-Up Table across multiple clock domains.

- **Block Memory**—Stores the data using embedded blocks of memory. Xilinx literature describes this implementation as block RAM or BRAM. National Instruments recommends using this option for FIFOs larger than 300 bytes. If you select the **Block Memory** option, you might not be able to read data in a target-scoped FIFO or VI-defined FIFO until up to six clock cycles after you write the data to the FIFO. Use the Timed Out? output of the FIFO Method Node configured with the Read or Write method to determine when the data is ready.

Note If you select the Block Memory option, you might not be able to read data in a target-scoped FIFO or VI-defined FIFO until up to six clock cycles after you write the data to the FIFO. Use the Timed Out? output of the FIFO Method Node configured with the Read or Write method to determine when the data is ready.

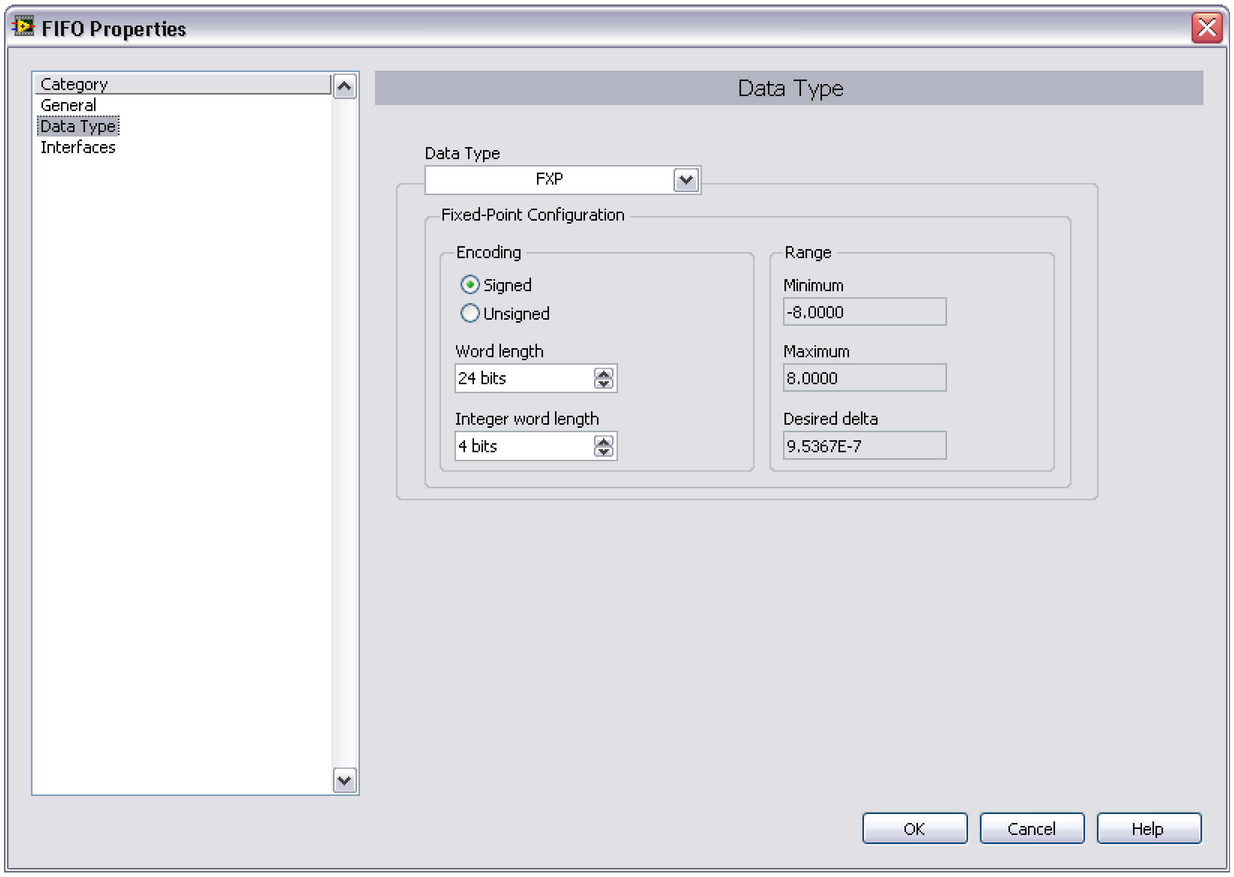
All of the above implementation options contain the following components:

- **Actual Number of Elements**—Returns the configured number of elements. Sometimes the requested number of elements is not compatible with the FIFO configuration. In these case, the Requested Number of Elements is coerced to a compatible number. The actual number of elements can vary, such as 1026 or 1029, when using target optimal control logic because the application implements FIFOs using slice fabric or built-in control logic depending on the clock domain and target type on which the FIFO is used.
- **Control Logic**—Specifies how the FIFO is implemented in the FPGA.
  - **Slice Fabric**—Specifies to use flip-flops, LUTs, and block memory to implement the control logic for the FIFO.
  - **Target Optimal**—Specifies to use built-in FIFO control logic or slice fabric control logic depending on the target and application.

– **Built-In**—Specifies to use built-in FIFO control logic. Not all FPGAs support built-in FIFO control logic and restrictions apply.

Data Type Page

In the FIFO Properties dialog box, select **Data Type** from the Category list to display the page shown in Figure 6-23.



Data Type

Data Type: FXP

Fixed-Point Configuration

Encoding: [✓ Signed] Unsigned

Word length: 24 bits

Integer word length: 4 bits

Range

Minimum: -8.0000

Maximum: 8.0000

Desired delta: 9.5367E-7

Figure 6-23. FPGA FIFO Properties Dialog Box – Data Type Page

Use this page to specify the data type for a FIFO or memory.

This page includes the following components:

- **Data Type**—Specifies the data type of the data in the FIFO or memory. You can select a fixed-point data type, a Boolean data type, or an 8-, 16-, 32-, or 64-bit signed or unsigned integer data type. You also can select a custom control as the data type. If you select **FXP**, you must configure the data type in the Fixed-Point Configuration section.
  – **Fixed-Point Configuration**—Sets the configuration settings for fixed-point data. Set Data Type to **FXP** to enable the fixed-point

settings. You configure the Encoding settings, and LabVIEW automatically determines the Range settings.

Note Fixed-point data type FIFOs do not include an overflow bit when transferring data. If you need to transfer the overflow bit, you may use a separate FIFO. You also may specify a wider data type for the FIFO. Then you can manipulate the data to add the overflow bit when writing to the FIFO and subtract the overflow bit when reading from the FIFO.

- **Encoding**—Sets the binary encoding settings for a fixed-point value.
  - **Signed**—Sets whether the fixed-point value is signed.
  - **Unsigned**—Sets whether the fixed-point value is unsigned.
  - **Word length**—Sets the number of bits that LabVIEW uses to represent all the possible fixed-point values.
  - **Integer word length**—Sets the number of integer bits, or the number of bits to shift the binary point to reach the most significant bit, for all the possible fixed-point values. Integer word length can be positive or negative.
- **Range**—Indicates the range for a fixed-point value.

Note These values display the values in double-precision floating-point representation, so the precision of the display at Maximum, Minimum, and Desired delta might not be exact in terms of fixed-point representation.

  - **Minimum**—Indicates the minimum value for the fixed-point data range.
  - **Maximum**—Indicates the maximum value for the fixed-point data range.
  - **Desired delta**—Indicates the maximum distance between any two sequential numbers in the fixed-point data range.

- **Select Control**—Opens a dialog box in which you can navigate to the custom control you want to use. This button appears only when you select **Custom control** in the Data Type pull-down menu.

- **Maximum Data Width for DRAM**—Indicates the physical port width of the DRAM. The data type you select for the DRAM memory cannot be wider than the data width.

  Make sure the sum of the data widths of all items in a custom control is not greater than the Maximum Data Width for DRAM.

Memory & FIFO Palette

Use the FIFO functions, shown in Table 6-2, on the Memory & FIFO palette to transfer data between structures in your FPGA VI.

Table 6-2. FIFO Objects in the Memory & FIFO Palette

FIFO Constant	Specifies a FIFO item on the block diagram.
FIFO Method Node	Invokes a FIFO method or action on an FPGA VI. You can use methods such as write, read, clear, enable and disable on FPGA FIFOs.
VI-Defined FIFO Configuration	Represents a VI-defined FIFO.

The VIs and functions on this palette can return general LabVIEW error codes, specific FPGA Module error codes, or error codes specific to the FPGA target.

VI-Defined FIFO

Use VI-defined FIFOs for data that you need to access only from a single VI, or data that you must maintain separately for separate instances of a reentrant subVI, because each instance of the subVI contains a new instance of the VI-defined FIFO.

To create a VI-defined FIFO, place a VI-Defined FIFO Configuration node onto the block diagram, as shown in Figure 6-24.

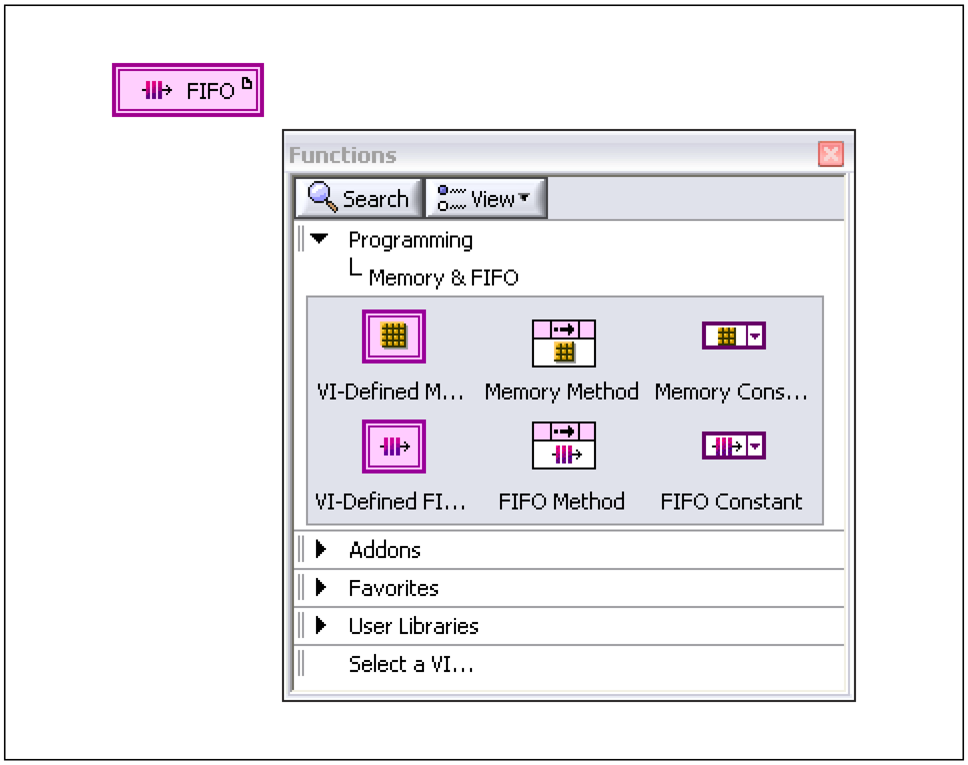


Figure 6-24. Creating a VI-Defined FIFO

Right-click on the VI-Defined FIFO Configuration node and select **Configure**. This launches the FIFO Properties dialog box where you can configure the FIFO.

Note Since this FIFO is VI-Defined, the Type selector will not be available in the FIFO Properties dialog box.

FPGA FIFO Write and Read

Place the FIFO Method Node from the Memory & FIFO palette onto the block diagram. Right-click and choose **Select FIFO»My FIFO** to associate the node with the previously defined FIFO, as shown in Figure 6-25.

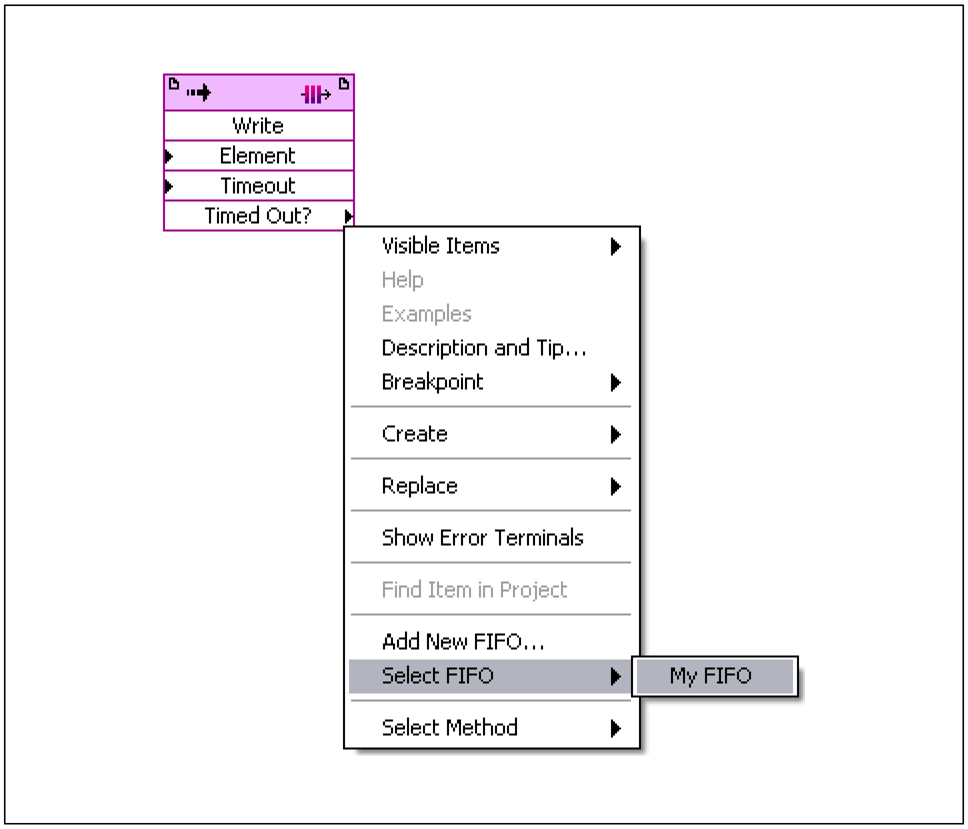


Figure 6-25. Selecting a FIFO

Alternatively, you can click a FIFO item in the Project Explorer window and drag it onto the block diagram to add a FIFO Method Node to the block diagram.

Once you have placed a FIFO Method Node and selected the FIFO that you want to access, you can write to or read from the FIFO.

Write

You can use the FIFO Write method to put data into the target FIFO. Configure the FIFO Method Node by right-clicking on the node and choosing **Select Method»Write**. The Write method has the following terminals:

- **FIFO In**—Specifies the FIFO. If you leave FIFO In unwired, you can specify the FIFO by right-clicking the FIFO Method Node and selecting a FIFO from the shortcut menu. Otherwise, you can wire a FIFO control, FIFO constant, VI-Defined FIFO Configuration node, or another FIFO Method node to FIFO In.
- **Element**—Specifies the data element you want to store in the FIFO. The Element data type is the data type you configure in the FIFO Properties dialog box when you create the FIFO.
- **Timeout**—Specifies the number of clock ticks the function waits for available space in the FIFO if the FIFO is full. A value of –1 prevents the function from timing out. A value of 0 indicates no wait. Wire a constant of 0 to Timeout if you use the FIFO Method Node in a single-cycle Timed Loop.
- **FIFO Out**—Returns FIFO In if FIFO In is wired. Otherwise, FIFO Out returns the FIFO that you specify in the FIFO Method Node.
- **Timed Out?**—Returns TRUE if space in the FIFO is not available before the function completes execution. If Timed Out? is TRUE, the function does not add Element to the FIFO.

Read

You can use the FIFO Read method to retrieve the data contained in the FIFO from another structure or subVI in the FPGA VI. The FIFO Read method reads the oldest element in an FPGA FIFO and removes the element from the FIFO. Configure the FIFO Method Node by right-clicking on the node and choosing **Select Method»Read**. The Read method has the following terminals:

- **FIFO In**—Specifies the FIFO. If you leave FIFO In unwired, you can specify the FIFO by right-clicking the FIFO Method Node and selecting a FIFO from the shortcut menu. Otherwise, you can wire a FIFO control, FIFO constant, VI-Defined FIFO Configuration node, or another FIFO Method node to FIFO In.
- **Timeout**—Specifies the number of clock ticks the function waits for available data in the FIFO if the FIFO is empty. A value of –1 prevents the function from timing out, so the function completes execution only when data is available for reading. A value of 0 indicates that the function does not wait. Wire a constant of 0 to Timeout if you use the FIFO Method Node in a single-cycle Timed Loop.

- **FIFO Out**—Returns FIFO In if FIFO In is wired. Otherwise, FIFO Out returns the FIFO that you specify in the FIFO Method Node.
- **Element**—Returns the oldest data element in the FIFO. The Element data type is the data type you configure in the FIFO Properties dialog box when you create the FIFO. If the FIFO is empty, Element is undefined.
- **Timed Out?**—Returns TRUE if an element is not available in the FIFO before the function completes execution. If Timed Out? is TRUE, Element is undefined.

FIFO Overflow and Underflow

If there is a difference in the rate at which data is written to the FIFO and the rate at which it is read, then the FIFO may not behave as expected.

If the Write method occurs more often than the data is read, then the FIFO will eventually fill and an overflow condition occurs. When the FIFO is full, additional Write methods time out and data is not written to the FIFO until space becomes available. Space can be created by reading data from the FIFO or by resetting the FIFO. Any data that was not written as a result of an overflow is lost.

If the Read method occurs more often than the data is written, then the FIFO will eventually empty and an underflow condition occurs. When the FIFO is empty, the Read method times out.

The method for handling overflow and underflow depends on the importance of a lossless transfer. If it is critical that both the read and write functions cease in the event of an overflow/underflow, then the Timed Out? output of the FIFO Method Node can be used as a condition for termination of both the read and the write loop, as shown in Figure 6-26.

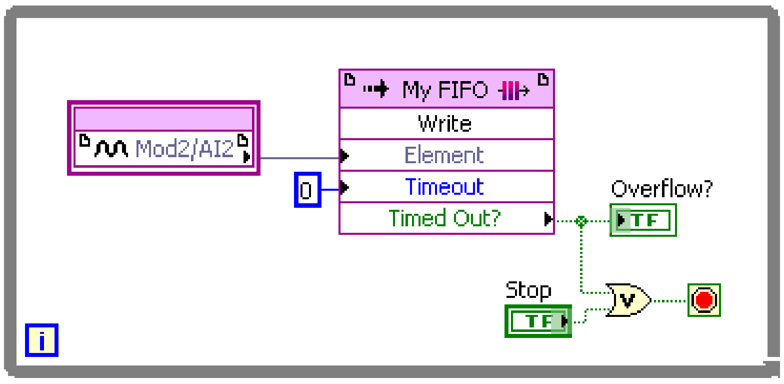


Figure 6-26. Handling Overflow by Terminating Execution

If you are only concerned that the timeout occurred during execution, then you can latch the Timed Out? output of the Read/Write method by using a shift register to store whether or not Timed Out? was ever true.

To prevent Timed Out? from occurring, you can use the Get Number of Elements to Read and Get Number of Elements to Write FIFO methods. Get Number of Elements to Read returns the number of elements remaining to be read from the FIFO. Get Number of Elements to Write returns the number of empty elements that remain in the FIFO. These two values can be monitored to ensure that the FIFO is never completely empty or full.

Separate I/O and Data Processing

Figure 6-27 shows an example of implementation using target-scoped FIFOs to pass data between parallel processes on an FPGA. The top loop handles data acquisition, timing of the acquisition, and writing the data to a FIFO. The second loop reads data from the FIFO and processes the data using a Butterworth Filter. In this technique, the top loop is not slowed by the processing that occurs in the bottom loop.

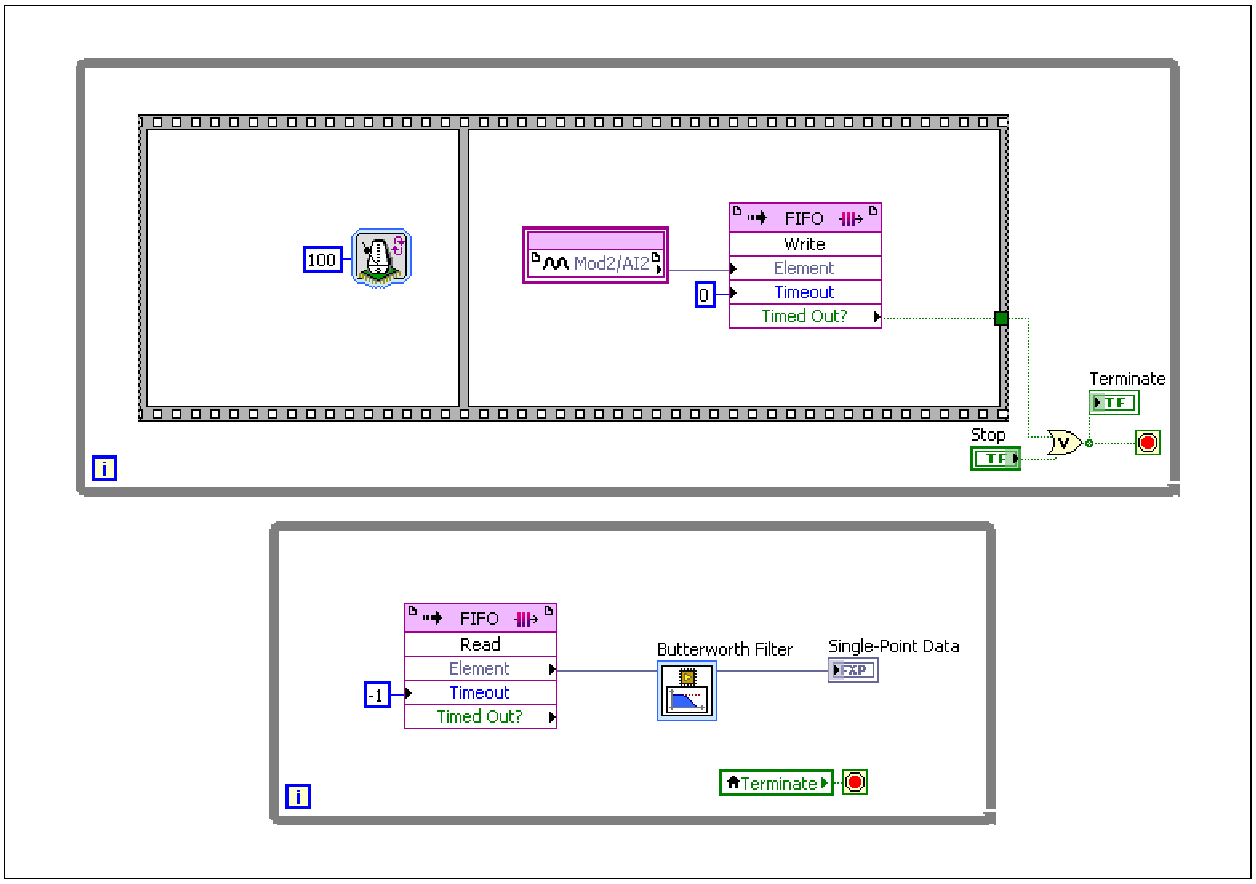


Figure 6-27. Communicating Data Between Parallel Processes with Target-Scoped FIFOs

Clear FIFO

FIFOs retain data even if the FPGA VI stops and restarts. You should empty a target-scoped FIFO at the start of your VI by using a FIFO Method Node configured to use the Clear method, as shown in Figure 6-28.

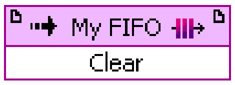


Figure 6-28. FIFO Clear

Study the examples in the NI Example Finder in the **Toolkits and Modules»FPGA»CompactRIO»FPGA Fundamentals»FIFOs** folder to learn more about FPGA FIFOs

G. Comparison of Data Sharing Methods

Selection of a data sharing method should be based upon the needs of your application. Depending on the features that are important to your application, any of the methods discussed may be appropriate. Table 6-3 summarizes the methods that can be used to transfer data among parallel loops and presents common use cases for each.

Table 6-3. Comparison of Data Sharing Methods

Transfer Method	FPGA Resource	Lossy?	Between Clock Domains?	New Data Notification?	Common Use
Variables	Logic	Yes	Yes	No	Control, Simulation
Memory Items	Memory	Yes	No	No	Control, Simulation, Variable-sized data
Flip-Flop FIFOs	Logic	No	No	Yes	Datalogging (FIFOs smaller than 100 bytes)
Look-Up Table FIFOs	Logic	No	No	Yes	Datalogging (FIFOs smaller than 300 bytes)
Block Memory FIFOs	Logic and Memory	No	Yes	Yes	Datalogging (FIFOs 300 bytes or larger)

Note Only variables and block memory FIFOs have the ability to transfer data across different clock domains. The use of different clock domains relates to the use of single-cycle Timed Loops, which is discussed in Lesson 7, *Single-Cycle Timed Loops*.

Self-Review: Quiz

1. Which of the following is *not* a method for passing data between parallel loops on an FPGA?
   a. Local variable
   b. Memory item
   c. FIFO
   d. A wire containing the data

2. Match each method of FIFO implementation with its recommended usage.

Block Memory	For FIFOs smaller than 300 bytes
Flip-Flops	For FIFOs larger than 300 bytes
Look-Up Table	For FIFOs smaller than 100 bytes

3. Which of the following may result if an FPGA VI with a FIFO Read or Write does not monitor the Timed Out? output, assuming that the Timeout is not –1 (infinite)? (Select all answers that apply)
   a. Attempts to write when there is no room in the FIFO may go undetected
   b. Attempts to read from and empty FIFO may be misinterpreted as having produced valid data
   c. The FIFO may reset itself automatically
   d. LabVIEW will report a code generation error
   e. None of the above

Self-Review: Quiz Answers

1. Which of the following is *not* a method for passing data between parallel loops on an FPGA?

   a. Local variable

   b. Memory item

   c. FIFO

   d. A wire containing the data—A wire would cause the loops to execute in sequence instead of in parallel. To pass data between loops, you should use a local or global variable, memory item, or FIFO, depending on the amount of data that you want to transfer, what resources you want to allocate to the transfer, and whether or not you are concerned with transferring all data that is acquired.

2. Match each method of FIFO implementation with its recommended usage.

Block Memory	**For FIFOs larger than 300 bytes**—This implementation uses FPGA memory to store data which only has a minor impact on the amount of logic available for the rest of the FPGA application.
Flip-Flops	**For FIFOs smaller than 100 bytes**—This implementation results in the fastest performance, but is costly in terms of the logic that is used.
Look-Up Table	**For FIFOs smaller than 300 bytes**—This implementation makes use of available look-up tables in the logic section of the FPGA.

3. Which of the following may result if an FPGA VI with a FIFO Read or Write does not monitor the Timed Out? output, assuming that the Timeout is not –1 (infinite)? (Select all answers that apply)

   a. Attempts to write when there is no room in the FIFO may go undetected

   b. Attempts to read from and empty FIFO may be misinterpreted as having produced valid data

   c. The FIFO may reset itself automatically

   d. LabVIEW will report a code generation error

   e. None of the above

Notes

7

Single-Cycle Timed Loops

In this lesson, you learn to improve performance of your FPGA VI by using the single-cycle Timed Loop (SCTL), which executes at the rate of selectable FPGA clocks.

Topics

A. Dataflow in FPGA

Enable Chain

The enable chain is additional logic added to the FPGA code to guarantee that dataflow on the FPGA is consistent with the LabVIEW dataflow paradigm. The enable chain is a series of flip-flops, also known as registers, that run in parallel with the actual flow of data on the block diagram. A flip-flop holds a bit of data and outputs the data on clock edges.

Dataflow within the FPGA

LabVIEW executes code in a dataflow manner. Nodes execute when data is present on all inputs. When the node finishes execution, the outputs of the node pass data to the next node downstream. Figure 7-1 shows an example of the FPGA hardware required to implement a Boolean operation.

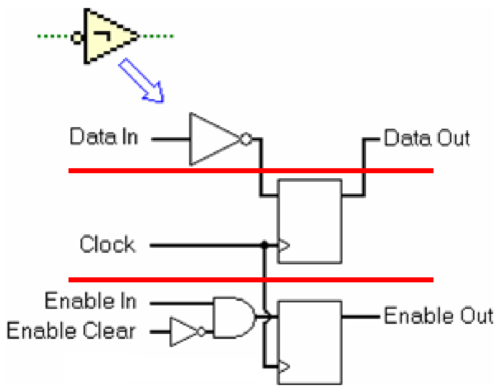


Figure 7-1. A LabVIEW Not Function and the Corresponding FPGA Logic that Implements the Not Function

LabVIEW code is transformed into FPGA logic in three sections—logic, synchronization, and the enable chain. The logic, shown in the upper third of Figure 7-1, corresponds to the actual LabVIEW operation. In this example, the LabVIEW code is a Not function corresponding to an inverter in the FPGA hardware. The synchronization register is shown in the middle of Figure 7-1 guarantees that data is output only on rising edges of the clock. The final portion of FPGA code generated from the LabVIEW code is the enable chain. The enable chain is an additional register that only outputs on the rising edge of the clock. The enable chain guarantees that the FPGA logic executes in the same order depicted on the block diagram.

Due to enable chain overhead, each function or VI takes a minimum of one clock cycle. Some functions, such as analog input operations, can take hundreds of clock cycles, depending upon the complexity of the operation and hardware limitations.

A VI can run only as fast as the sum of the items in a combinatorial path. One advantage of using an FPGA is that code can run in true parallel to another operation. Because you can create code in parallel, it is often best to design code so that as many parallel operations can take place as possible. This uses the same amount of FPGA space and can increase the execution speed by reducing the combinatorial path size.

B. Single-Cycle Timed Loop

The single-cycle Timed Loop is one of the most powerful constructs in FPGA programming. Code inside the single-cycle Timed Loop is more optimized, takes up less space on the FPGA, and executes faster than identical code in a standard While Loop. The single-cycle Timed Loop removes the enable chain from the loop to save space on the FPGA. Because the additional enable chain registers are removed, all operations in a single-cycle Timed Loop can complete in a single clock cycle. In addition, eliminating the enable chain overhead reduces the total space used on the FPGA because the flip-flops used for the enable chain are no longer required. The single-cycle Timed Loop is a great tool for safety-critical and control applications where fast loop rates are important.

Figure 7-2 shows identical code in a standard While Loop and single-cycle Timed Loop. The vertical lines indicate the end of a clock cycle. The code in the While Loop requires four clock cycles to execute, in addition to two clock cycles of loop overhead.

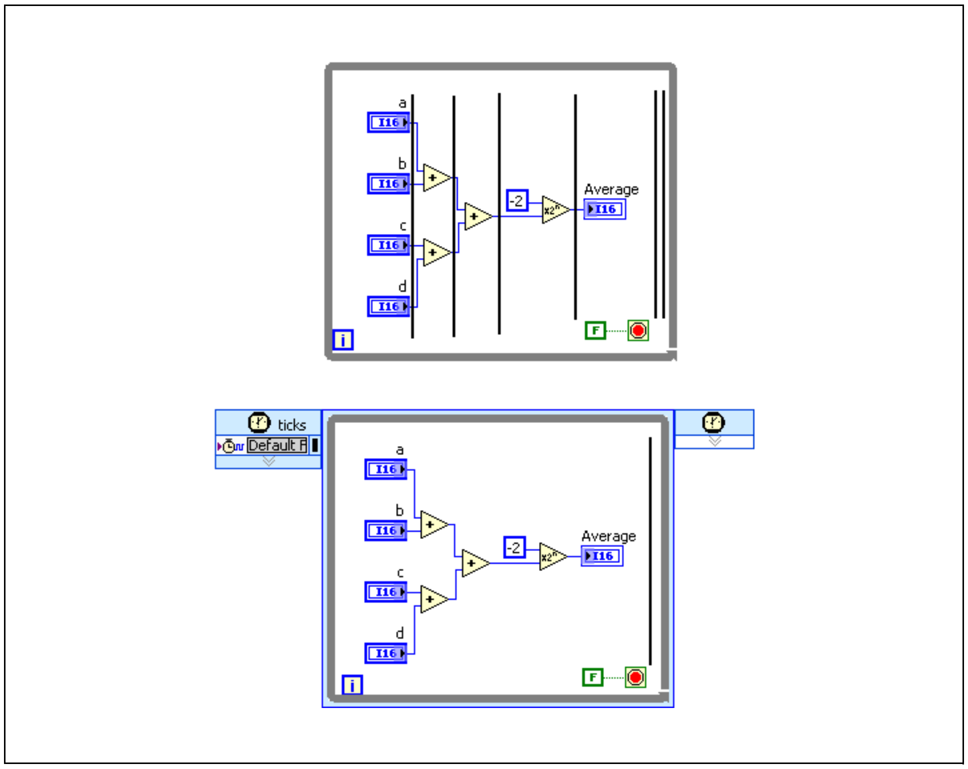


Figure 7-2. Single-Cycle Timed Loop and While Loop Comparison

Because the single-cycle Timed Loop executes in exactly one clock cycle, the clock period must be long enough to allow all the operations to complete in a single cycle. The clock frequency can technically be from 2.5 to 210 MHz; however, the faster the clock frequency, the fewer the computations possible in the single-cycle Timed Loop. It is usually not possible to know prior to compiling if your code will execute in a single-cycle Timed Loop. Some functions and structures, such as analog input I/O, loops, and interrupts, are too slow for a single-cycle Timed Loop and result in a broken Run arrow if you place them in a single-cycle Timed Loop. However, the Run arrow may be solid and compilation still fails because LabVIEW does not know the timing requirements of a chain of commands in the single-cycle Timed Loop until the compile is complete.

Timed Loop Behavior Based On Target

The FPGA Module single-cycle Timed Loop differs from the standard LabVIEW Timed Loop in that the timing of the FPGA Timed Loop corresponds exactly to the clock rate of the FPGA clock you specify. By configuring a single-cycle Timed Loop to use a clock other than the base clock of the FPGA target, you can implement multiple clock domains in an FPGA VI. You can specify the FPGA clock that controls the single-cycle Timed Loop by wiring a value to the Source Name input on the Input Node of the single-cycle Timed Loop or by using the Configure Timed Loop dialog box.

A Timed Loop in a VI running on a Windows-based or Real-Time target will execute one or more subdiagrams, or frames, sequentially for each iteration of the loop at the period you specify. In contrast, the single-cycle Timed Loop in an FPGA VI executes one subdiagram at the same period as an FPGA clock. Use the single-cycle Timed Loop when you want to develop VIs with multirate timing capabilities, precise timing, feedback on loop execution, timing characteristics that change dynamically, or several levels of execution priority. Right-click the structure border to add, delete, insert, and merge frames.

C. FPGA Clocks

FPGA Base Clock

A base clock is a digital signal existing in hardware that you can use as a clock for an FPGA application.

You can configure the FPGA base clock that is associated with an FPGA target by specifying its name and certain properties. The properties available vary according to FPGA target and FPGA base clock selected. Complete the following steps to configure the FPGA base clock associated with an FPGA target:

1. Create a new project or open an existing project.
2. Add an FPGA target to the project.
3. If the FPGA target you use does not automatically add an FPGA base clock to the Project Explorer window, add the FPGA base clock.
4. Right-click an FPGA base clock in the Project Explorer window and select **New FPGA Derived Clock** from the shortcut menu. The FPGA Derived Clock Properties dialog box appears.
5. (Optional) Enter a name in the Name text box. By default, the name is the actual derived frequency.
6. Type the frequency at which you want the derived clock to run in the **Desired Derived Frequency** text box.

   LabVIEW finds the closest frequency that the underlying FPGA supports and displays it in the Derived Frequency text box in the Actual Derived Configuration section. If the FPGA supports the Derived Frequency, the Message text box displays that the frequency is available. Otherwise, it displays the error between the derived frequency and the desired frequency.
7. Click **OK** to accept the derived frequency and close the dialog box. The FPGA-derived clock appears in the Project Explorer window with the derivation ratio in parentheses.

FPGA targets can support more than one base clock, although some targets add only a subset of supported base clocks to the Project Explorer window when you add the target. If the FPGA target you use does not include the FPGA base clock you need in the Project Explorer window when you add the target to a LabVIEW project, you can add a new base clock to the LabVIEW project.

Complete the following steps to add a new base clock to a LabVIEW project:

1. Create a new project or open an existing project.
2. Add an FPGA target to the project.
3. Right-click the FPGA target and select **New»FPGA Base Clock** from the shortcut menu. The FPGA Base Clock Properties dialog box appears.
4. Select an available clock from the Resource pull-down menu.
5. Click **OK**. The new FPGA base clock appears in the Project Explorer window below the FPGA target.

FPGA Derived Clock

A derived clock is a clock you create from a base clock that you can use as a clock for an FPGA application. You can derive additional clocks from FPGA base clocks in a LabVIEW project. Complete the following steps to derive an FPGA clock:

Note Support of FPGA-derived clocks varies by FPGA target. Refer to the specific FPGA target hardware documentation for more information.

1. Create a new project or open an existing project.
2. Add an FPGA target to the project.
3. If the FPGA target you use does not automatically add the FPGA base clock you desire to the Project Explorer window, add the FPGA base clock.
4. Right-click an FPGA base clock in the Project Explorer window and select **New FPGA Derived Clock** from the shortcut menu. The FPGA Derived Clock Properties dialog box appears.

Note If the FPGA target does not support FPGA-derived clocks or the FPGA base clock you add, the **New FPGA Derived Clock** option is dimmed in the shortcut menu. You cannot derive a clock from a component-level IP clock.

5. Type the frequency at which you want the derived clock to run in the Desired Derived Frequency text box or enter values in the Multiplier and Divisor text boxes and read the resulting Message text. If the FPGA target supports the Derived Frequency in the Actual Derived Configuration section, the Message text box displays `Valid Configuration`.
6. Click **OK**. The FPGA-derived clock appears in the Project Explorer window below the FPGA base clock with the derivation ratio in parentheses.

FPGA Top-level Clock

You can define a top-level clock for an FPGA target in the Project Explorer window. The top-level clock is the clock that the FPGA VI uses outside of single-cycle Timed Loops. The FPGA target uses one of the FPGA base clocks by default. If the FPGA target allows, you can configure the FPGA base clock and set it as the top-level clock in the project to control execution rates. If the FPGA target does not allow you to configure the FPGA base clock the way you want, you can use a derived clock. Support of FPGA derived clocks varies by FPGA target. Refer to the specific FPGA target hardware documentation for more information.

The code inside the single-cycle Timed Loop executes at the base clock or derived clock rate, depending on which clock you use as a source. The top-level clock controls the execution rate of the code outside of the single-cycle Timed Loop.

Complete the following steps to select a top-level clock for an FPGA target:

1. Create a new project or open an existing project.
2. Add an FPGA target to the project.
3. If the FPGA target you use does not automatically add an FPGA base clock to the Project Explorer window, add the FPGA base clock. If you want to use the FPGA base clock as the timing source for the top-level clock, skip the following step.
4. (Optional) Create an FPGA derived clock.
5. Right-click the FPGA target in the Project Explorer window and select **Properties** from the shortcut menu. The FPGA Target Properties dialog box appears.
6. Select **Top-Level Clock** in the Category list.
7. Select **Default** if you want to use the default FPGA target clock. Select **Select Configured Clock** and a configured clock from the Configured Clock list if you want to use a clock you configured.
8. Click **OK**.

Each FPGA target provides at least one clock to control the internal operations of the FPGA. FPGA target clocks determine the execution time of the individual VIs and functions on the FPGA VI block diagram. You can compile FPGA VIs with faster clock rates for higher performance. However, not all FPGA VIs can compile properly with faster clock rates. If you select a clock rate that is too fast for the FPGA VI, the Compilation Status window notifies you that the compile failed. You must select a lower clock rate and attempt to compile again.

You can change the top-level FPGA target clock rate for an FPGA target by right-clicking the FPGA target in the Project Explorer window and selecting **Properties** from the shortcut menu. On the Top-Level Clock page of the FPGA Target Properties dialog box, you can set the top-level clock. You also can change the clock rate for a single-cycle Timed Loop within an FPGA VI by double-clicking the Input Node and selecting a clock rate in the Configure Timed Loop dialog box. You can select the FPGA target top-level clock or a clock you derive from the FPGA target base clock.

If you change the top-level FPGA target clock rate or the clock rate of a single-cycle Timed Loop in the FPGA VI, you must recompile the FPGA VI.

D. Single-Cycle Timed Loop Errors

Unsupported Objects

You can use most VIs and functions available when you target an FPGA target in a single-cycle Timed Loop. However, you cannot use the following VIs, functions, and structures in a single-cycle Timed Loop. LabVIEW reports a code generation or compile-time error for the single-cycle Timed Loop when you try to compile an FPGA VI with any of the following VIs, functions, or structures in a single-cycle Timed Loop.

- Analog Period Measurement VI
- Butterworth Filter VI
- Discrete Delay VI
- Divide function
- FIFO Clear function
- For Loop
- FPGA I/O Method Node except with some FPGA targets
- FPGA I/O Property Node except with some FPGA targets
- Interrupt VI

- Look-Up Table 1D VI with the Interpolate data checkbox selected
- Loop Timer Express VI
- Multiple FPGA I/O Nodes configured for the same I/O resource if at least one node is inside the loop and at least one node is outside the loop
- Non-reentrant subVIs if you use multiple instances
- Notch Filter VI
- PID (FPGA) VI
- Quotient & Remainder function
- Reciprocal function
- Rotate 1D Array function
- Sine Wave Generator VI
- Square Root function
- Timed Loop
- Wait Express VI
- Wait on Occurrence function
- While Loop

The FPGA target you use might not support additional VIs or functions. Also, some targets do not support specific I/O items both inside and outside a single-cycle Timed Loop. Refer to the specific FPGA target hardware documentation for more information.

Table 7-1 describes how Timed Loops interact with other components.

Table 7-1. Timed Loop Interaction with Other Components

Type of Timed Loop	Interaction with Other Components
Timed Loops in VIs under My Computer	If you place the Timed Loop in a VI located under My Computer, the Timed Loop displays some terminals that FPGA targets do not support. So, if you open the VI under an FPGA target, the unsupported terminals still appear. If you place the Timed Loop in a VI under an FPGA target, the Timed Loop hides terminals that are not supported. So, if you open the VI under My Computer, the Timed Loop does not display all terminals that My Computer supports.
Timed Loops in VIs under an FPGA Target	When you place a Timed Loop in an FPGA VI, only the Source Name input appears visible by default. Other than Source Name and Error, the inputs available on the Input Node of the Timed Loop have no effect when you use the Timed Loop in an FPGA VI. Error is the only supported output of the Timed Loop in an FPGA VI. Do not add frames before or after the single-cycle Timed Loop frame to try to use the single-cycle Timed Loop as a Timed Sequence structure in an FPGA VI. The LabVIEW FPGA Module does not support Timed Sequence structures.
Indicators in Single-Cycle Timed Loops	You can place indicators in the single-cycle Timed Loop only if you do not have any local variables writing to the indicators.
FPGA I/O Nodes and Timed Loops	You can use the FPGA I/O Node in the single-cycle Timed Loop if the FPGA target allows it. If the FPGA target you use supports I/O in the single-cycle Timed Loop, you can use only the Arbitrate if Multiple Requestors Only and Never Arbitrate arbitration options. If you select Arbitrate if Multiple Requestors Only, you cannot use more than one instance of the FPGA I/O Node for a specific I/O item in the FPGA VI. If you select Never Arbitrate, you can use more than one instance of the FPGA I/O Node for a specific I/O item in the FPGA VI if each instance in a single-cycle Timed Loop is executing at the same rate.
Flat Sequences and Timed Loops	You can use the Flat Sequence or Stacked Sequence structure in the single-cycle Timed Loop. However, the entire sequence of frames must execute in one clock cycle.

Table 7-1. Timed Loop Interaction with Other Components (Continued)

Type of Timed Loop	Interaction with Other Components
SubVIs and Timed Loops	You cannot use more than one instance of a non-reentrant or shared subVI in a single-cycle Timed Loop. You can use multiple instances of a reentrant VI inside a single-cycle Timed Loop as long as the reentrant VI does not use shared resources.
Wait On Occurrence Function and Timed Loops	You cannot use the Wait on Occurrence function in a single-cycle Timed Loop. However, you can use the Set Occurrence function. You then can use the Wait on Occurrence function outside the single-cycle Timed Loop in a While Loop or For Loop.

Long Combinatorial Paths

Long combinatorial paths take more time to execute and limit the maximum clock rate of the clock domain.

Long combinatorial paths are typically a problem in single-cycle Timed Loops because the logic between the input register and the output register must execute within one period of the clock rate you specify. In the single-cycle Timed Loop, enable chain registers within and between components are removed, increasing the length of the combinatorial path between registers. If the code in a combinatorial path does not execute within a clock cycle, LabVIEW returns a timing violation in the Compilation Status window.

Note Deeply-nested Case structures also can cause LabVIEW to return a timing violation in the Compilation Status window.

Timing Violation Analysis Window

Click the **Investigate Timing Violation** button in the Compilation Status window to display the Timing Violation Analysis window. The Investigate Timing Violation button appears only if the compile server encounters timing violations while trying to compile an FPGA VI.

Use this window to identify components in the FPGA application that cannot execute within the application clock rate. Double-click an item in the list or click the **Show Element** button to locate the node on the block diagram.

This window includes the following components:

- **Timing information (s)**—Lists the propagation delay of components in the FPGA VI that cause the timing violation. The units of Total, Logic, and Routing are in nanoseconds.
  - **Paths**—Lists sets of VIs and components that exceed the applicable FPGA clock rate. Each path describes the VIs and components between two internal registers. When items in the table correspond to objects on the block diagrams, such as functions, structures, and subVIs, double-clicking the items in the table highlights the corresponding object on the block diagram.

    Some items in the table are non-diagram components and do not correspond directly to objects on the block diagrams. Non-diagram components include resources, arbitration circuits, component-level IP (CLIP), and other circuits that depend on the target hardware. You may be able to use the internal name of the non-diagram component to correlate the non-diagram component with a block diagram object or CLIP.
  - **Total Delay**—Indicates the sum of Logic and Routing. Because of rounding, the value of Total might differ slightly from the sum of the values of Logic and Routing.
  - **Logic Delay**—Indicates the amount of time in nanoseconds that a logic block takes to execute.
  - **Routing Delay**—Indicates the amount of time in nanoseconds that a signal takes to traverse between FPGA logic blocks.
  - **Max Fanout**—Displays the maximum number of logic block inputs from which a single logic block output connects. This maximum fanout can occur anywhere along the path. High signal fanout contributes to greater routing delays.
- **Show Element**—Highlights on the block diagram the item you select in the Paths list. You also can double-click an item in the Paths list to highlight the item on the block diagram.
- **Show Path**—Highlights on the block diagram all items in the path you select in the Paths list.

Fixing Timing Violations

You can use the following strategies to fix a timing violation:

- If the target supports the Xilinx Options page, change some of the compilation options.
- Reduce long combinatorial paths.
- Use pipelining.
- Reduce the number of nested Case Structures.
- Use smaller data types.
- Recompile the FPGA VI. Because the compilation process non-deterministically maps the FPGA VI to the FPGA, the process does not produce the same results each time you compile an FPGA VI. Therefore, if the FPGA VI did not miss the required clock rate by much, recompiling the FPGA VI might fix the timing violation.
- Reduce the clock rate of the application.

Erroneously Listed Single-Cycle Timed Loops

If the FPGA VI uses a large area on the FPGA, the Xilinx compiler optimizations might map different single-cycle Timed Loops to different look-up tables (LUTs) in the same slice. If two different single-cycle Timed Loops map to the same slice and a timing error occurs in one of them, the Timing Violation Analysis window might indicate the wrong single-cycle Timed Loop has the timing violation.

E. Optimizing Code within a While Loop

You can also use single-cycle Timed Loops to optimize code in your VI, even if you don’t intend to immediately reiterate the code inside the loop. Figure 7-3 shows how to use a single-cycle Timed Loop to speed up a portion of code even though it is not meant to iterate more than once. Place as much code as possible inside a single-cycle Timed Loop and then wire a True constant to the loop-termination terminal so the single-cycle Timed Loop executes exactly once. Using the single-cycle Timed Loop removes the enable chain from the portion of the FPGA code inside the single-cycle Timed Loop.

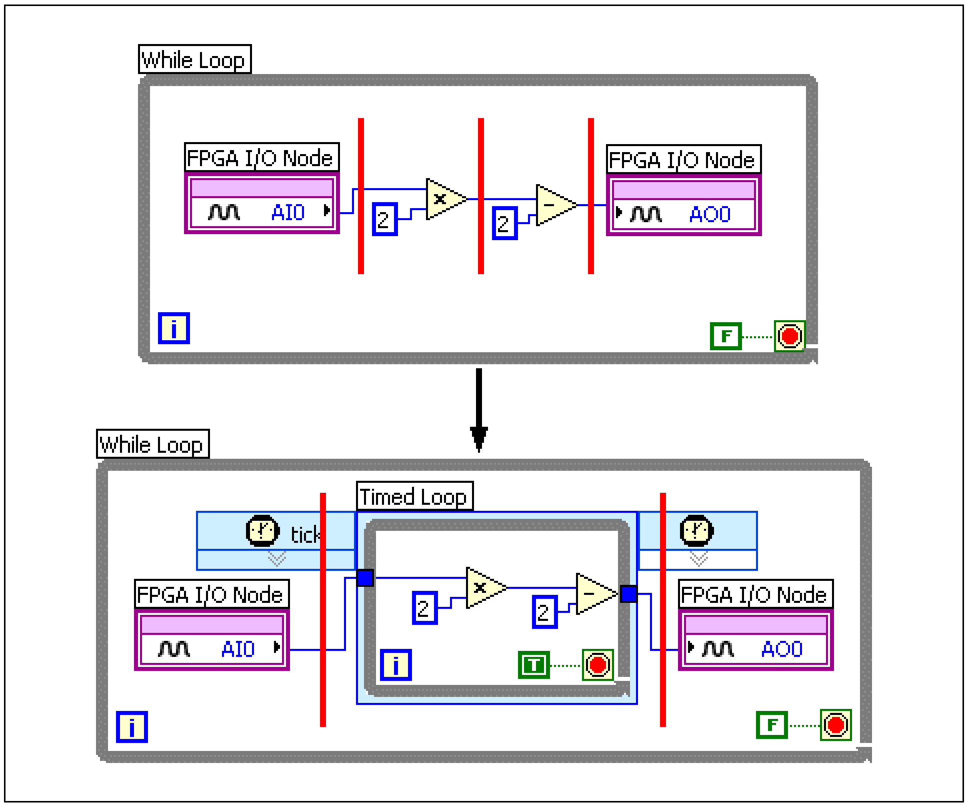


Figure 7-3. Single-Cycle Timed Loop Used to Increase the Speed in a Portion of the Code

Complete the following steps to select a clock other than the top-level clock as the timing source for a single-cycle Timed Loop in an FPGA VI. The single-cycle Timed Loop uses the top-level clock of the FPGA target by default.

Note Support of the single-cycle Timed Loop varies by FPGA target. Refer to the specific FPGA target hardware documentation for more information.

1. Create a new project or open an existing project.
2. Add an FPGA target to the project.
3. If the FPGA target you use does not automatically add the FPGA base clock you desire to the Project Explorer window, add the FPGA base clock. If you want to use the FPGA base clock as the timing source for the single-cycle Timed Loop, skip the following step.
4. (Optional) Create an FPGA derived clock.
5. Create a new VI or open an existing VI under an FPGA target in the Project Explorer window.
6. Add a single-cycle Timed Loop to the block diagram.

7. Select a clock by double-clicking the Input Node of the single-cycle Timed Loop to display the Configure Timed Loop dialog box and selecting one of the following options:
   - **Top-Level Timing Source**—Select this option if you want the Timed Loop to inherit the top-level timing source of the project it is in. You might use this option if you intend to reuse an FPGA VI among multiple FPGA targets.
   - **Select Timing Source—**Select this option if you want to use a timing source in the project other than the top-level clock. Then select a timing source from the **Available Timing Source** list. To use a clock that is not in the list, create a new base clock or create a new derived clock.

Self-Review: Quiz

1. Match each clock to the definition that best fits it.

FPGA base clock	Global clock that the FPGA VI uses outside a SCTL.
FPGA derived clock	Created from a base clock that you can use as additional clocks for an FPGA application.
FPGA top-level clock	Digital signal existing in hardware that you can use as a clock for an FPGA application.

2. How does the single-cycle Timed Loop create a smaller FPGA footprint and execute within one clock tick?
   a. By using other VIs logic functions when they are not in use
   b. By eliminating the enable chain overhead
   c. By passing the data to the real-time controller to process
   d. By skipping some functions and having incomplete functionality

3. Which of the following objects are *not* supported in single-cycle Timed Loops?
   a. Add
   b. For Loop
   c. Loop Timer VI
   d. NI 9211 Analog Input FPGA I/O Node

4. If the code in your SCTL causes a timing violation, which of the following methods can help fix it?
   a. Optimize the code in the SCTL
   b. Move some code out of the SCTL
   c. Select a faster clock for the SCTL timing source
   d. Select a slower clock for the SCTL timing source

Self-Review: Quiz Answers

1. Match each clock to the definition that best fits it.

FPGA base clock	**Digital signal existing in hardware that you can use as a clock for an FPGA application.**
FPGA derived clock	**Created from a base clock that you can use as additional clocks for an FPGA application.**
FPGA top-level clock	**Global clock that the FPGA VI uses outside a SCTL.**

2. How does the single-cycle Timed Loop create a smaller FPGA footprint and execute within one clock tick?
   a. By using other VIs logic functions when they are not in use
   b. By eliminating the enable chain overhead
   c. By passing the data to the real-time controller to process
   d. By skipping some functions and having incomplete functionality

3. Which of the following objects are *not* supported in single-cycle Timed Loops?
   a. Add
   b. For Loop
   c. Loop Timer VI
   d. NI 9211 Analog Input FPGA I/O Node

4. If the code in your SCTL causes a timing violation, which of the following methods can help fix it?
   a. Optimize the code in the SCTL
   b. Move some code out of the SCTL
   c. Select a faster clock for the SCTL timing source
   d. Select a slower clock for the SCTL timing source

Notes

8

Basic Host Integration – PC/Real-Time

In this lesson, you learn how to interface with your FPGA VI from your host PC or real-time controller. You create host VIs to control and pass data between your FPGA and host system.

Topics

A. Windows-Based Host Integration

In previous lessons, you learned how to develop VIs that run on the FPGA target. You also learned how to use Interactive Front Panel Communication to communicate with an FPGA VI running on an FPGA target with no additional programming. With Interactive Front Panel Communication, the host computer displays the FPGA VI front panel window and the FPGA target executes the FPGA VI block diagram.

In this lesson, you learn about Programmatic FPGA Interface Communication. With Programmatic FPGA Interface Communication, you programmatically monitor and control an FPGA VI with a separate host VI running on the host processor.

The host VI might provide a remote graphical user interface, log data, and post process data. The host VI can run in a Windows operating system on a PC or PXI controller; or it can run on a real-time operating system (RTOS) on a PC, PXI controller, Compact Vision System, or CompactRIO controller.

In this lesson, we discuss the Windows-based Host Integration architecture. In this architecture, the FPGA is physically connected to a Windows-based computer. For example, both a PCI-7831R connected to a Windows-based PC and a PXI-7831R connected to a PXI controller running a Windows operating system use the Windows-based Host Integration architecture.

You write a host VI to send information between the host computer and the FPGA target for the following reasons:

- Transfer data between Windows-based computer and FPGA
- User interface
- Do more data processing than you can fit on the FPGA
- Perform operations not available on the FPGA target
- Floating-point math
- Sequence multiple FPGA VIs
- Log data
- Control the timing and sequencing of data transfer

B. Developing a Windows-Based Host VI

A Windows-based Host VI should be created under the My Computer target in the Project Explorer.

Host VI Functions Palette with FPGA Interface Functions

When you install the LabVIEW Real-Time and LabVIEW FPGA Modules in LabVIEW, the FPGA Interface Functions palette is available for VIs located under the My Computer target in the project hierarchy.

Use the FPGA Interface functions to communicate with an FPGA VI from a host VI using the following operations:

- Establish and terminate communication with the FPGA VI
- Download, abort, reset, and run the FPGA VI on the FPGA target
- Read and write data to the FPGA VI
- Additional FPGA Interface functionality

The FPGA Interface functions can return general LabVIEW error codes, specific FPGA interface error codes, or error codes specific to the FPGA target. Refer to the specific FPGA target documentation for a list of target-specific error codes.

- **Open FPGA VI Reference**—Opens a reference to the FPGA VI or bitfile and FPGA target you select in the shortcut menu or that you specify with the resource name input. You must open a reference to the FPGA target before you can communicate between the host VI and the FPGA VI.
- **Read/Write Control**—Reads a value from or writes a value to a control or indicator in the FPGA VI on the FPGA target.
- **Invoke Method**—Invokes an FPGA Interface method or action from a host VI on an FPGA VI. Use methods to do the following: download, abort, reset, and run the FPGA VI on the FPGA target, wait for and acknowledge FPGA VI interrupts, read DMA FIFOs, and write to DMA FIFOs. The methods you can choose from depend on the target hardware and the FPGA VI. You must wire the FPGA VI Reference In input to view the available methods in the shortcut menu.
- **Close FPGA VI Reference**—Stops and resets the FPGA VI running on the FPGA target and closes the reference to the VI. Right-click the Close FPGA VI Reference function on the block diagram and select **Close** from the shortcut menu to close the reference without resetting the FPGA VI running on the FPGA target. The default is Close and Reset, which closes the reference, stops the FPGA VI, and resets the FPGA VI running on the FPGA target.

Open FPGA Reference Function

Begin coding a real-time host program by establishing communication with the FPGA. Add the Open FPGA VI Reference function to the block diagram.

Include a Close FPGA VI Reference function for every Open FPGA VI Reference function in a host VI. When the Open FPGA VI Reference function first executes on the block diagram, it checks whether the compiled FPGA VI already exists on the FPGA target. If the compiled FPGA VI is not on the FPGA target, the Open FPGA VI Reference function downloads the compiled FPGA VI to the FPGA target. To open a reference to the FPGA VI without running it, right-click the Open FPGA VI Reference function on the block diagram and select **Open**.

By default, the Open FPGA VI Reference function opens and runs the compiled FPGA VI on the FPGA target if it is not already running. To open a reference to the FPGA VI without running it, right-click the Open FPGA VI Reference function on the block diagram and select **Open**. You then can run the FPGA VI using the Invoke Method function.

The Open FPGA VI Reference function does not affect the FPGA VI if it is already downloaded and running. If you want to restart the FPGA VI, you can use the Invoke Method function to abort or reset the FPGA VI and use the Invoke Method function to run the FPGA VI again.

Open only one reference to an FPGA VI or a bitfile on one FPGA target at a time, and close one reference before opening another. Opening multiple references to FPGA VIs or bitfiles on the same physical target might cause erroneous behavior without any error messages.

The Open FPGA VI Reference function extracts the bitstream from the compiled FPGA VI or bitfile and stores the bitstream when you save the host VI. The bitstream contains the programming instructions LabVIEW downloads to the FPGA target.

Use the Open FPGA VI Reference function to perform the following tasks:

- Select the FPGA VI or bitfile with which the host VI communicates
- Select the FPGA target on which the FPGA VI runs
- Determine whether the host VI opens and runs the FPGA VI or just opens the FPGA VI

To select the VI, right-click the Open Reference icon and select **Configure Open FPGA VI Reference**. The Configure Open FPGA VI Reference dialog box opens. Select the FPGA target VI. The FPGA VI must be compiled first. The Open Reference VI appearance changes to acknowledge that it references the FPGA target VI.

Read/Write Control Function

The Read/Write Control function reads a value from or writes a value to a control or indicator in the FPGA VI on the FPGA target.

A host VI can control and monitor data passed through the FPGA VI front panel. You cannot access values on any wires on the FPGA VI block diagram that do not have controls or indicators unless the data is stored in a DMA FIFO, which is covered in Lesson 9, *DMA Data Transfers*.

First, use the Open FPGA VI Reference function to open a reference to the FPGA target. Then, wire the FPGA VI Reference Out parameter to the Read/Write Control function to access controls and indicators on the FPGA VI. You can read indicators and write controls, as well as write indicators and read controls. You can expand the Read/Write Control function to read or write multiple controls and indicators. When you run the host VI, the Read/Write Control function reads and writes controls and indicators in the order they appear in the Read/Write Control function on the block diagram.

Tip The Read/Write Control function supports scalar data, such as numeric and Boolean controls, and complex data, such as arrays and clusters. You can program the FPGA VI to bundle scalar data into arrays or clusters and then read or write the arrays or clusters of data as a single block with the host VI to make sure all data is read or written at the same time. You can use the Read/Write Control function to read whole clusters or an individual element of a cluster. If you must read multiple elements of a cluster, read the whole cluster. You can write to a whole cluster, but you cannot write to individual elements of a cluster. Be careful not to overuse arrays because arrays use a lot of space on an FPGA target.

The FPGA Module creates a register map, specific to the FPGA VI, that includes a hardware register for every control and indicator. LabVIEW uses the register map internally to communicate with the FPGA VI directly with Interactive Front Panel Communication and using the host VI with Programmatic FPGA Interface Communication.

Invoke Method Function

The Invoke Method function invokes an FPGA Interface method or action on an FPGA VI. Use methods to do the following:

- Abort
- Acknowledge IRQ
- Download
- Get FPGA VI Execution Mode
- Reset
- Run
- Wait on IRQ
- FIFO—Includes the following methods you can use to read from or write to a Direct Memory Access (DMA) FIFO in the FPGA VI. FIFO is the name of the FIFO item in the project.
  - Configure
  - Read
  - Start
  - Stop
  - Write

The methods you can choose from depend on the target hardware and the FPGA VI. You might have fewer or more methods available depending on the FPGA target. Refer to the specific FPGA target hardware documentation for information about the FPGA Interface methods you can use.

To specify a method, right-click the Invoke Method function and select **Method»x** from the shortcut menu, where **x** is the specific method. You must wire the FPGA VI Reference In input to view the available methods in the shortcut menu.

Close FPGA Reference Function

The Close FPGA Reference function closes the reference to the FPGA VI and, optionally, resets or aborts execution of the FPGA VI. By default, the Close FPGA VI Reference function closes the reference to the FPGA VI and resets the FPGA VI. To configure this function only to close the reference, right-click the function and select **Close** from the shortcut menu.

You have learned how to use the FPGA interface VIs to control and communicate with your FPGA VI. Remember that you should display only the necessary controls and indicators on your FPGA VI because extra front panel objects take additional space on the FPGA.

C. Introduction to Real-Time

FPGA in a Real-Time system is another major architecture using the LabVIEW FPGA Module and the LabVIEW Real-Time Module. This is the architecture for PXI and CompactRIO platforms that run LabVIEW Real-Time. In this architecture, the FPGA is physically connected to a Real-Time target. For example, both an FPGA connected to a CompactRIO Real-Time Controller and a PXI-7831R connected to a PXI controller running a Real-Time OS use this architecture. This architecture has the following components:

- LabVIEW FPGA VI—Implements custom I/O or ultra high-speed control. Because your LabVIEW code executes directly in hardware, you can achieve hardware-level (ns) determinism.
- LabVIEW Real-Time Module—Adds deterministic floating-point processing and control algorithms on a dedicated processor.
- LabVIEW for a Windows-based Host VI—Contains the user interface for your real-time system.

It is common for three processors to run simultaneously and communicate with each other in this architecture—the FPGA, the CompactRIO real-time controller, and remote machines, as shown in Figure 8-1. The external host program might provide a remote graphical user interface, log data, and post process data. The host program can run in a Windows operating system on a PC or PXI controller; or it can run on a real-time operating system on a PC, PXI controller, Compact Vision System, or CompactRIO controller. The RT host controller communicates with the CompactRIO RT processor over the network using shared variables or low-level languages such as TCP/IP.

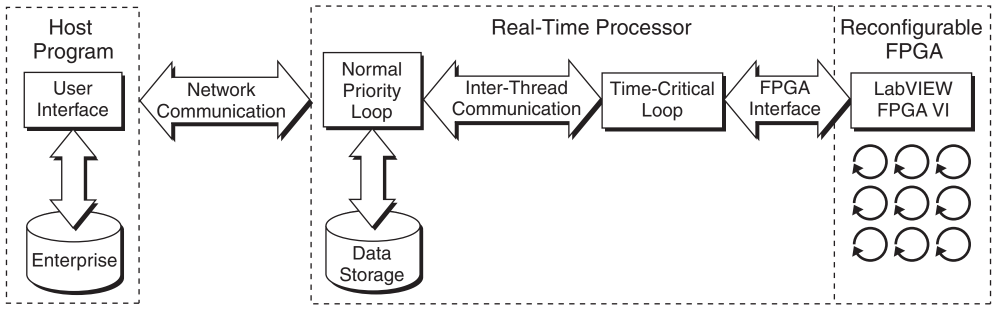


Figure 8-1. FPGA for Real-Time Application Software Architecture

The real-time program might have two or more loops—one time-critical loop for floating-point control, signal processing, analysis, and point-by-point decision making; and one or more normal-priority loops for embedded data logging, remote panel Web interface, and Ethernet/serial communication. The time-critical loop has higher priority, so the normal-priority loop executes when the time-critical loop waits. Refer to the *LabVIEW Real-Time 1* course for more information about multi-threading and setting priorities.

The VI running on the RT processor communicates with the FPGA through front panel controls and indicators and other techniques discussed later. These front panel objects are represented as data registers within the FPGA.

The FPGA core application tasks include input, output, communication, and control.

The architecture and tasks described previously can vary depending on your application requirements. Figure 8-2 shows four possible run-time configurations for an embedded Real-Time controller.

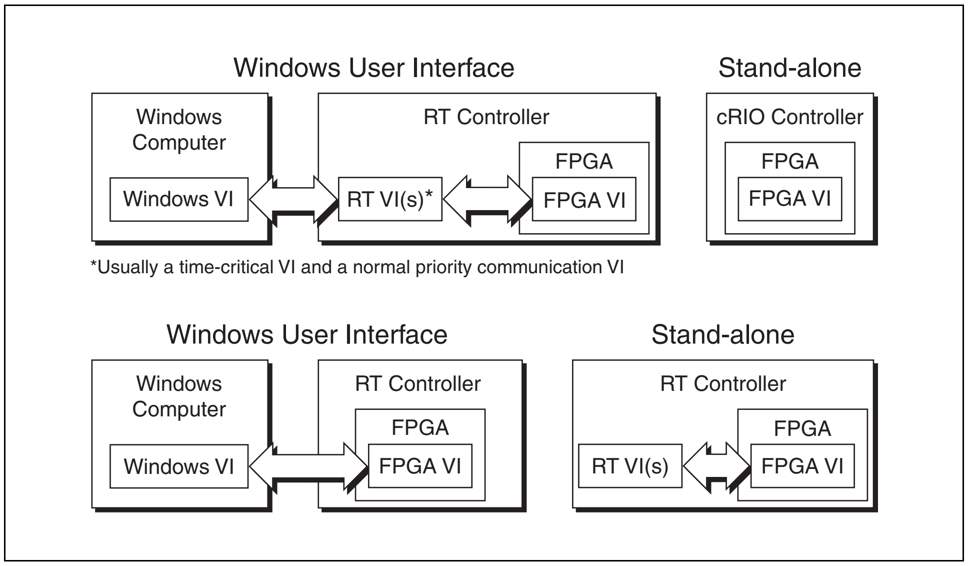


Figure 8-2. Possible FPGA for Real-Time Configurations

When you complete the system of Windows-based PC, Real-Time controller and FPGA, you create a hierarchy of host programs. The supervisory program running on the PC is the host for the Real-Time Controller, but the Real-Time controller VI is a host program for the FPGA.

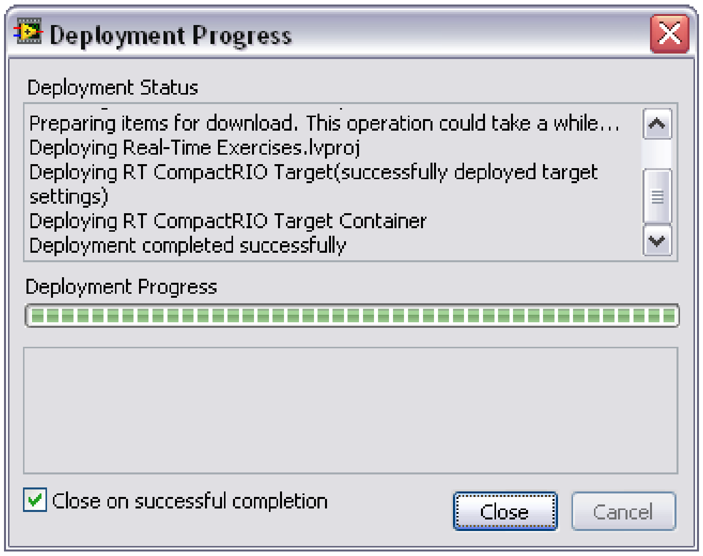
This course only introduces a few concepts of real-time programming in LabVIEW. Refer to the *LabVIEW Real-Time 1* course for further instruction in real-time operating systems and the LabVIEW Real-Time Module. You can use LabVIEW Real-Time to develop more sophisticated Real-Time applications. Some advantages of using a Real-Time host include:

- Code execution is more deterministic
- Ability to set tasks to run at different priorities, including a time-critical priority
- Ability to separate deterministic tasks from non-deterministic tasks
- Real-time OS is more stable than Windows OS
- Real-time targets can have 1MHz clocks (Windows OS has a 1kHz clock)

D. Developing an RT Host VI

An RT VI is created under a Real-Time target in the Project Explorer. The RT host VI communicates with the FPGA VI using the same FPGA Interface functions discussed earlier in the *Developing a Windows-Based Host VI* section.

When you finish developing the host VI, click the **Run** button. The Deployment Progress window, shown in Figure 8-3, opens. When the VI finishes deploying, close the Deploying Status dialog box to run the application.



```
Preparing items for download. This operation could take a while...
Deploying Real-Time Exercises.lvproj
Deploying RT CompactRIO Target(successfully deployed target
settings)
Deploying RT CompactRIO Target Container
Deployment completed successfully
```

Figure 8-3. Deployment Progress Window

Using Front Panel Communication in LabVIEW Real-Time

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

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

You also can use front panel communication to debug RT 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 the *LabVIEW Real-Time 1* course for information about debugging real-time applications.

Front panel communication is a good communication method to use during development because you can quickly monitor and interface with RT VIs. However, 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 time-critical sections of code. If no front panel terminals are in the time-critical section of code, then the time-critical section of code will run deterministically.

E. Developing a Windows-Based VI

The Windows-based VI is used in the FPGA for Real-Time architecture to communicate between the Windows-based computer and the Real-Time target. Possible tasks handled by the Windows-based VI include:

- Act as a user-interface
- Send data and commands to RT host VI
- ActiveX and .NET functionality
- Database connectivity
- Log data
- Data processing

Network Communication

You can use network communication protocols to communicate between your Windows-based VI running on your Windows-based computer and the RT VI running on the RT target. Each protocol has its advantages and disadvantages. In this course, we briefly discuss the network-published shared variable protocol with the RT FIFO enabled.

You can use network-published shared variables to share data between VIs running on different targets across a network. 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. For datalogging applications, you can use timestamps to programmatically ensure that each value is logged once and only once.

Using network communication allows you to:

- Run a VI on the RT Target and a VI on the Windows-based system
- Control the data exchanged
  - Which front panel objects on the PC get updated and when
  - Which RT target VI components are visible on the front panel of the Windows-based VI
- Control timing and sequencing of the data transfer
- Perform additional data processing or logging

Network communication is very powerful and there are various approaches to configure network communication for a real-time application. For more information on other methods of transferring data between a Windows-based computer and RT target, shared variable configuration, shared variable deployment and undeployment, and LabVIEW Real-Time programming practices, refer to the *LabVIEW Real-Time 1* course.

F. Prepare RT Host for Final Application

When you complete the development work on an RT VI, you may want to deploy it. Because front panel communication can be nondeterministic, you usually want to compile the RT VI into a stand-alone application that runs on the RT target. 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.

For more information on deploying real-time applications, creating a build specification, communicating with deployed applications, and system replication, refer to the *LabVIEW Real-Time 1* course.

Self-Review: Quiz

1. Match the task with the VI in an FPGA/RT architecture.

Logging data to a file	A. FPGA VI
I/O operations	B. RT Host VI
Floating-point math	C. Windows-based User Interface VI
User interface	
40MHz Logic	
Inserting data into an enterprise database	
Separating deterministic and non-deterministic tasks	

2. Which of the following should you use in the host VI to communicate with the FPGA VI?
   a. Array functions
   b. Network-published shared variables
   c. FPGA Interface VIs
   d. TCP/IP VIs

3. Which of the following are necessary in a host VI that reads and writes values of controls/indicators on the FPGA VI?
   a. Open FPGA VI Reference
   b. Read/Write Control
   c. Invoke Method
   d. Close FPGA VI Reference

4. You should use Interactive Front Panel Communication in your final RT application.
   a. True
   b. False

Self-Review: Quiz Answers

1. Match the task with the VI in an FPGA/RT architecture.

Logging data to a file	**B. RT Host VI** **C. Windows-based User Interface VI***
I/O operations	**A. FPGA VI**
Floating-point math	**B. RT Host VI** **C. Windows-based User Interface VI**
User interface	**C. Windows-based User Interface VI**
40MHz Logic	**A. FPGA VI**
Inserting data into an enterprise database	**C. Windows-based User Interface VI**
Separating deterministic and non-deterministic tasks	**B. RT Host VI**

* Using the NI 9802 cRIO module, the FPGA VI can log data to a file.

2. Which of the following should you use in the host VI to communicate with the FPGA VI?

   a. Array functions

   b. Network-published shared variables

   c. FPGA Interface VIs

   d. TCP/IP VIs

3. Which of the following are necessary in a host VI that reads and writes values of controls/indicators on the FPGA VI?

   a. Open FPGA VI Reference

   b. Read/Write Control

   c. Invoke Method

   d. Close FPGA VI Reference

4. You should use Interactive Front Panel Communication in your final RT application.

    a. True

    b. False

Notes

Notes

9

DMA Data Transfers

In the previous lesson, we discussed how to perform basic communication between a host VI and an FPGA VI. In this lesson, we further explore communication and introduce synchronization and data transfer techniques between the host VI and FPGA VI.

Topics

A. LabVIEW FPGA and Host Communication

Communication between the FPGA VI and host VI consists of exchanging information and synchronizing the two applications. On their own, the VIs on the host and the FPGA run asynchronously. To synchronize the VIs or to use timing information from the FPGA VI in the host VI, you must add synchronization code to your VIs.

The level of synchronization required depends on the application. You might require synchronization if the timing of the FPGA VI must control the timing of the application running on the RT controller. You might also require synchronization if you must acquire data at a known rate and transfer all the data without loss to the host application for processing or data logging.

Asynchronous applications are suitable for slower control loops or control applications where the synchronization of I/O to the control algorithm is not critical. For example, if you run the acquisition loop at a higher rate than the control loop on your RT controller, most cases do not require synchronization. When you read the most recently acquired input value, output values from the control algorithm update on the I/O cycle.

Limitations of Front Panel Controls/ Indicators

In Lesson 8, *Basic Host Integration – PC/Real-Time*, you learned about using front panel controls and indicators to pass data between the FPGA VI and the host VI. This method of communication only transfers the most current data stored on the control or indicator, which could result in data being lost if data is being written to the control or indicator faster than it is being read.

In Lesson 3, *FPGA Programming Basics*, you learned how to optimize your code by reducing the number of controls and indicators on your front panel. This optimization helps to reduce the size of your FPGA code because each control or indicator requires that FPGA logic be allocated to it. It is especially important to minimize the number of large controls and indicators since they require that even more space be allocated. Mixed-data clusters and numeric arrays are two examples of large front panel objects that should be avoided if possible.

Front panel controls and indicators transfer data between the FPGA VI and the host VI using the host CPU. As a result, the speed of data transfer is highly dependent upon the CPU speed and availability. A slower CPU or lack of resources would result in slower data transfer from the FPGA target to the host.

When a writer loop runs faster than a reader loop, you can lose data. Therefore, you need a technique to store data in memory until it is read. You can do this with a buffer. A buffer is an area of computer memory that stores multiple data items.

B. DMA FIFOs

The best way to buffer data between the FPGA and the host is to use Direct Memory Access (DMA) transfers. Direct memory access (DMA) FIFOs transfer data from the FPGA directly to memory on the host (Windows-based or RT). DMA FIFOs can stream large amounts of data to and from the host VI. This allows the FPGA to use the host RAM as if it were its own. This offers significant performance advantages over using the CPU to read from or write to controls and indicators on the FPGA VI.

With DMA FIFOs the host computer processor can perform calculations while the FPGA target transfers data to the host computer memory. FPGA targets that support DMA FIFOs have direct access to write to memory on the host computer without involving the host computer processor. Without DMA FIFOs, you can transfer data only through the host computer processor. LabVIEW performs DMA transfers through bus mastering. FPGA targets that support DMA FIFOs can master the PCI bus. The FPGA target controls the PCI bus and accesses memory directly without needing to access the host processor.

A DMA FIFO consists of two parts, as shown in Figure 9-1, an FPGA part and a host computer part.

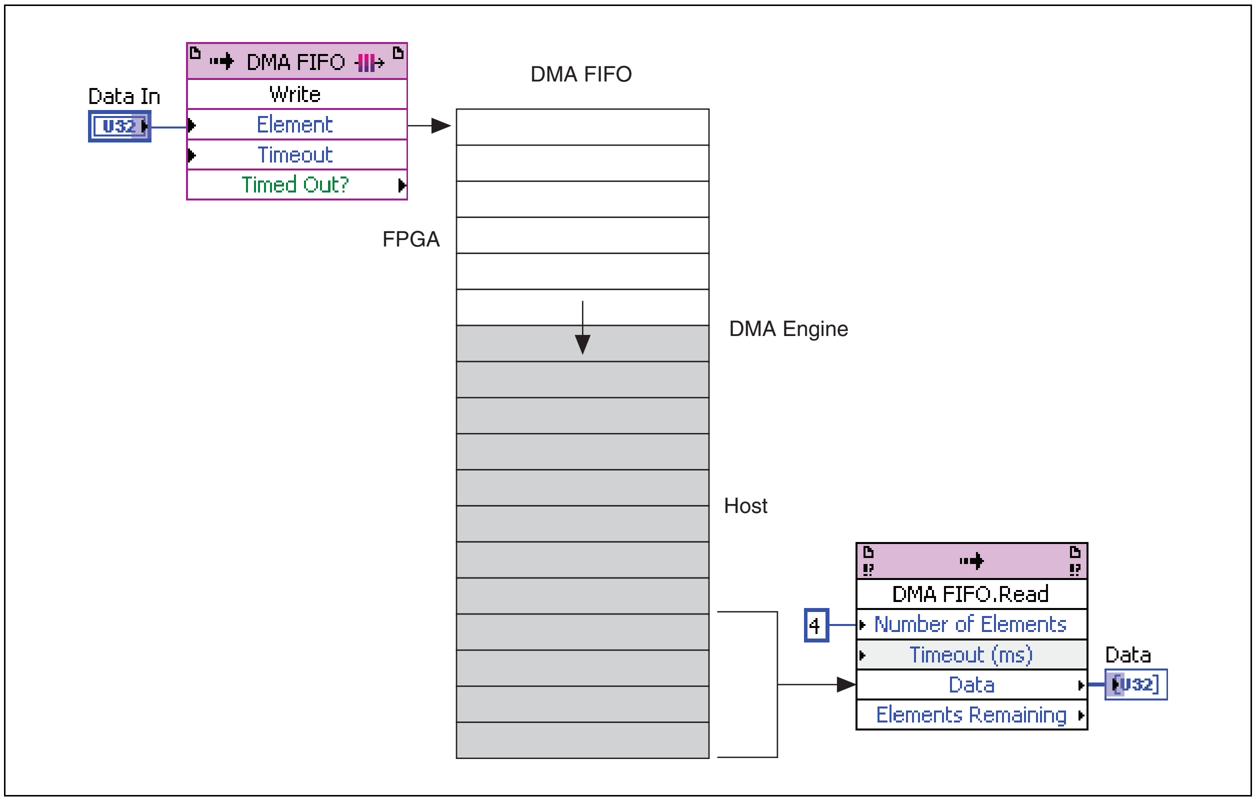


Figure 9-1. Two Parts of the DMA FIFO

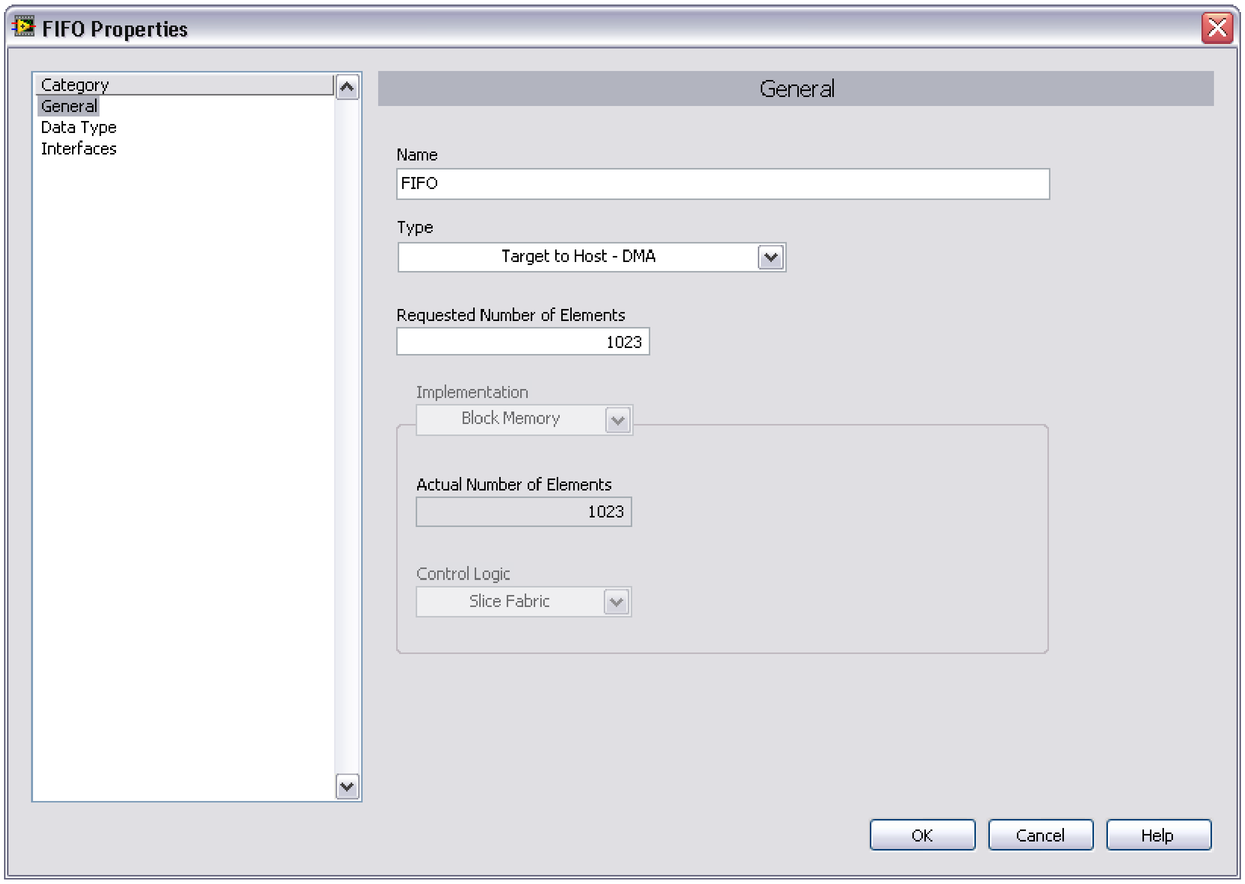
The FPGA part of the DMA FIFO is implemented on the FPGA. The FPGA VI writes the FIFO one element at a time with the DMA FIFO Write method or reads the FIFO one element at a time with the DMA FIFO Read method. The host computer part of the DMA FIFO is stored in memory on the host computer. LabVIEW uses a DMA controller to connect the two parts. The DMA controller includes driver software and hardware logic. When the DMA controller runs, it transfers data between the two parts of the FIFO automatically so they act as one FIFO array. The host VI reads from or writes to the FIFO one or more elements at a time with the Invoke Method function.

The transfer of data to and from the host buffer by the host computer is typically the slowest part of the transfer process. The rate at which data is transferred is dependent upon CPU speed and availability.

As mentioned previously, data is transferred to and from the FPGA FIFO by the FPGA code on a point by point basis. The amount of data that can be held in the FPGA FIFO is limited based upon the amount of memory available on the FPGA and the user-defined size of the FPGA FIFO. The host computer transfers a user-defined number of data elements to and from the host buffer. By default, the host buffer can contain 10,000 elements. This value can be modified by using a Configure Method in the block diagram of the host VI.

Creating a DMA FIFO

You can create and modify a DMA FIFO in the same way that you created FPGA FIFOs in Lesson 6, *Data Sharing on FPGA*. In the Project Explorer window, right-click on the FPGA target and select **New»FIFO** to display the FPGA FIFO Properties dialog box shown in Figure 9-2.



Name: FIFO
Type: Target to Host - DMA
Requested Number of Elements: 1023
Implementation: Block Memory
Actual Number of Elements: 1023
Control Logic: Slice Fabric

Figure 9-2. DMA FIFO Properties Dialog

The Type option specifies the type of FIFO to use. Select **Target-Scoped** if you want to use a FIFO to transfer data within the FPGA VI. Select **Host to Target - DMA** or **Target to Host - DMA** to use a DMA FIFO to transfer data between the host VI and target. When you select either type of DMA FIFO, the Implementation option is disabled, because block memory will always be used.

Number of Elements specifies the number of elements the FIFO can hold. The maximum number of elements the FIFO can hold depends on the amount of space available on the FPGA. If the FPGA does not have enough space for the number of elements you specify, the FPGA VI fails to compile and an error message appears. If you select **DMA - Host to Target** or **DMA - Target to Host** in the Type pull-down menu, Number of Elements specifies the size of the FPGA part of the DMA FIFO.

For DMA FIFOs, you can specify a size that is one less than a power of two for Target to Host - DMA and five more than a power of two for Host to Target - DMA. The General page displays a size of 2^M–1 or 2^M+5, where M is the address width. LabVIEW coerces Requested Number of Elements to the next higher valid value. If the FPGA does not have enough space for the coerced Requested Number of Elements, the FPGA VI fails to compile.

Refer to the FPGA FIFO section of Lesson 6, *Data Sharing on FPGA*, for information on the other options available in this dialog box.

FPGA VI

To read or write data to a DMA FIFO from the FPGA VI, drag the FIFO to the block diagram from the Project Explorer. This creates a FIFO Method Node on your block diagram. If you configured the FIFO as Host to Target - DMA, then the FIFO Method Node will be configured to read data from the FIFO. If you configured the FIFO as Target to Host - DMA, then the node will be configured to write data. The Write and Read method nodes are shown in Figure 9-3.

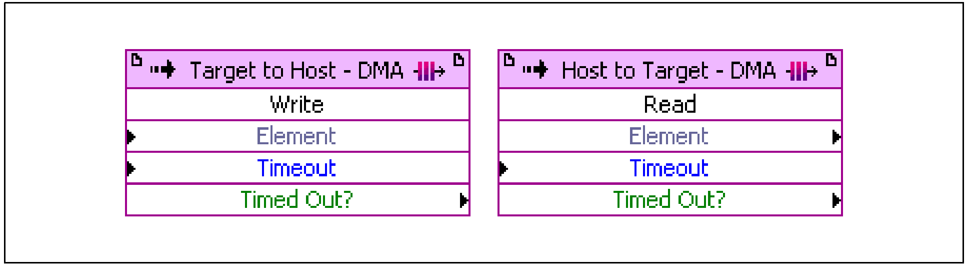


Figure 9-3. DMA FIFO - FPGA VI Write or Read

You can also add a FIFO Method Node to your block diagram from the Memory & FIFO palette. If you use this method, right-click the function and choose **Select FIFO»x**, where **x** is the name of the FIFO that you created in the Project Explorer. By default, the FIFO Method Node is set to write data. If you have configured the FIFO as Target to Host - DMA, then this is the correct method to use. If you configured the FIFO as Host to Target - DMA, then you must change this node to use the Read method. Right-click on the node and select **Select Method»Read**.

Host VI

Most of the host VI interaction with a DMA FIFO takes place using an Invoke Method node, which is located on the FPGA Interface palette on the host. In order to access the DMA FIFO that you configured under the FPGA target, you must complete the following steps:

1. Open a reference to an FPGA VI or bitfile.

   The FPGA target, FPGA VI, and host VI must be in the same LabVIEW project if you want to open a reference to an FPGA VI. The host VI does not need to be in a project if you open a reference to a bitfile. If you open a reference to an FPGA VI, the project must include a DMA FIFO item under the FPGA target and the FPGA VI must include a FIFO Write function on the block diagram that writes to the DMA FIFO item.
2. Add an Invoke Method Node to the block diagram of the host VI in the dataflow where you want the host VI to read the DMA FIFO. Make sure the host VI runs the FPGA VI before you read the DMA FIFO. Wire the FPGA VI Reference In input.
3. Click the Invoke Method Node and select **FIFO**, where FIFO is the name of the FIFO item in the project. This will list the methods that are available for the FIFO that you created under the FPGA target.

The Invoke Method node calls an FPGA Interface method or action from a host VI on an FPGA VI. This method can be used to read from or write to DMA FIFOs.

To specify a method, right-click the Invoke Method function and choose **Select Method»FIFO»x** from the shortcut menu, where FIFO is the name of the FIFO item in the project and **x** is the method that you want to run.

The following FIFO methods are available for execution on the host VI.

- **Configure**—Specifies the depth of the host memory part of the DMA FIFO.
- **Read**—Reads elements of the DMA FIFO from the host memory part of the FIFO.
- **Start**—Begins DMA data transfer between the FPGA target and the host computer.
- **Stop**—Stops the DMA data transfer between the FPGA target and the host computer.
- **Write**—Writes elements to the DMA FIFO from the host VI.

Configure

The Depth parameter of this method specifies the number of elements in the host memory part of the DMA FIFO. If you do not wire this parameter, the Invoke Method function uses a default of 10,000 elements. If you place the FIFO Configure method after a FIFO Start or FIFO Read method in the dataflow, the Invoke Method function sets the new depth when the next FIFO Start or FIFO Read method executes.

Read

This method reads elements of the DMA FIFO from the host memory part of the FIFO. The Read method returns Data when the Number of Elements is available. If the Timeout period ends before the Number of Elements is available, Data is empty

- **Number of Elements**—Determines the number of elements you read from the DMA FIFO.
- **Timeout (ms)**—Specifies the number of milliseconds the Invoke Method function waits before timing out. The default is 5000 milliseconds. Set this parameter to `-1` if you want the Invoke Method function to wait indefinitely for the number of elements.
- **Data**—Returns the data contained in the host memory part of the DMA FIFO. The number of elements contained in Data is either the same as Number of Elements or zero if the function times out.
- **Elements Remaining**—Returns the number of elements remaining in the host memory part of the DMA FIFO.

Start

This is an optional method that can be used to begin the DMA data transfer between the FPGA target and the host computer. The FIFO Method Nodes configured with the Read or Write methods automatically start DMA data transfer. You might want to use this method if you want to start data transfer with the DMA FIFO before you read the first element of the FIFO.

Stop

This is an optional method that can be used to stop the DMA data transfer between the FPGA target and the host computer. This method empties all data from the host memory and FPGA parts of the FIFO. Most applications do not require using the Stop method.

Write

This method writes elements to the DMA FIFO from the host VI. The Write method returns Empty Elements Remaining when the data is written or when the Timeout (ms) period ends.

- **Data**—Specifies the data that you want to transfer to the FPGA target.
- **Timeout (ms)**—Specifies the number of milliseconds the Invoke Method function waits before timing out. The Invoke Method function times out if the host part of the FIFO does not contain enough space to write Data to by the time the number of milliseconds you specify elapse. The default is 5000 milliseconds. Set this parameter to `-1` if you want the Invoke Method function to wait indefinitely.
- **Empty Elements Remaining**—Returns the number of empty elements remaining in the host memory part of the DMA FIFO.

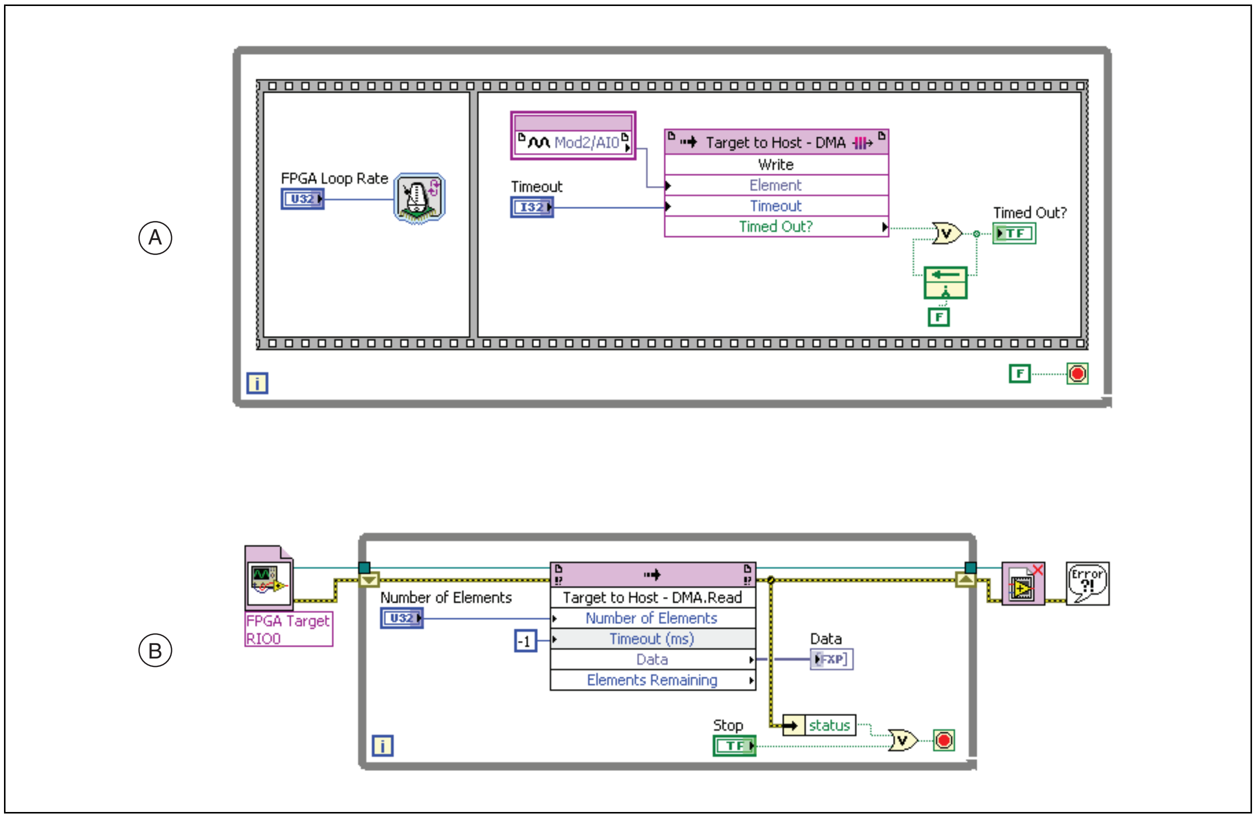
Target to Host Architectures

There are three primary architectures for reading data from the DMA FIFO on the host VI, depending on the level of control required for the rate at which the Read method executes and the amount of data obtained by the Read method.

- **Blocking**—The host DMA FIFO Read method waits indefinitely to read a fixed number of elements from the FIFO.
- **Polling**—The host DMA FIFO Read method obtains all of the available elements in the FIFO at a user-defined rate.
- **Polling with a Fixed Number of Elements**—The host DMA FIFO Read method obtains a fixed number of elements at a user-defined rate.

Blocking

For the blocking method of reading data from a DMA FIFO, the user defines the number of elements that will be obtained by the DMA FIFO Read method, as shown in Figure 9-4, where the top portion of code executes on the FPGA and the bottom portion executes on the host VI.



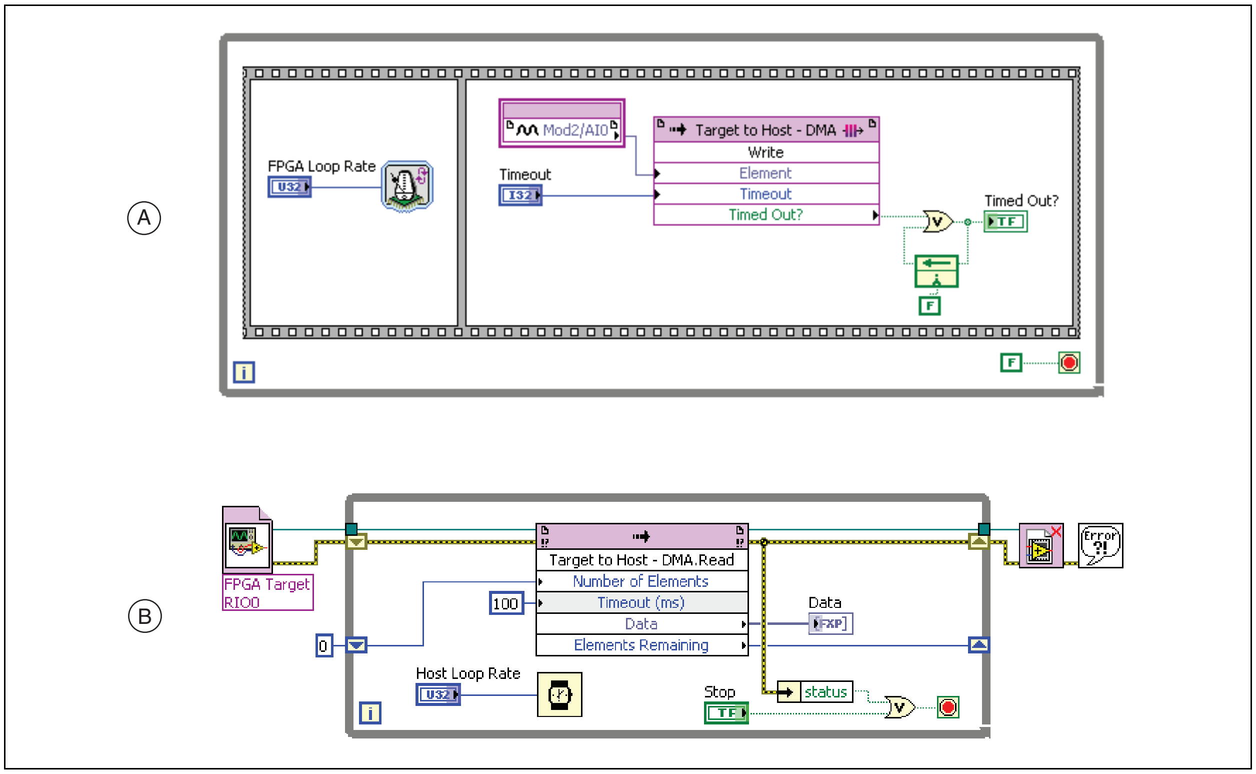
A FPGA Block Diagram	B Host Block Diagram

Figure 9-4. Blocking Method

The user specifies a Timeout value of –1 so that the method will wait indefinitely for the specified number of samples to become available. The DMA FIFO Read method actively uses the CPU while waiting for the desired amount of data to become available

Polling

For the polling method of reading data from a DMA FIFO, the user defines the rate at which data will be obtained by the DMA FIFO Read method. This implementation determines the number of elements that are available in the FIFO and reads them at a user-defined rate, as shown in Figure 9-5.



A	FPGA Block Diagram	B	Host Block Diagram

Figure 9-5. Polling Method

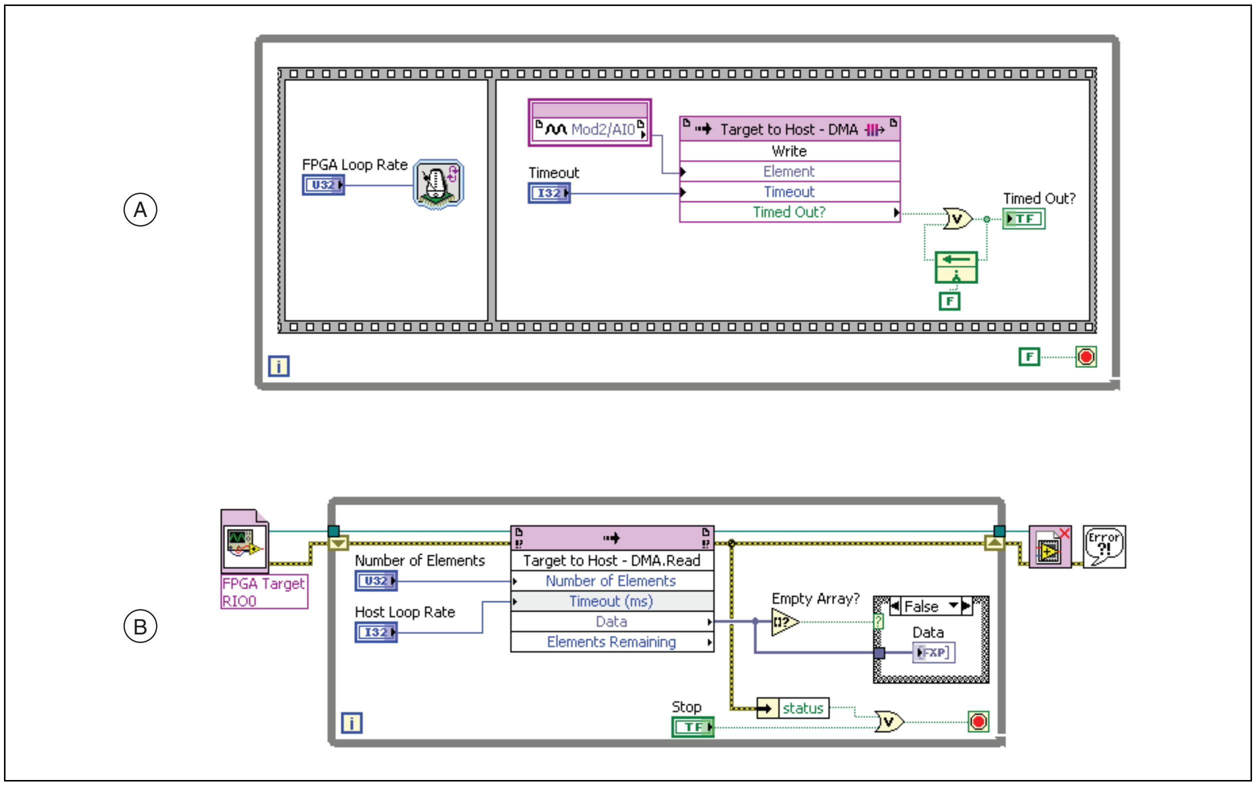
The first iteration of the loop returns the number of elements available in the FIFO and the next iteration will read that many elements on the next iteration of the loop. Wiring a zero to the Number of Elements for the first iteration quickly determines the number of elements that have already been written to the FIFO. For the first iteration, an empty array is written to Data.

Every iteration of the Host VI will execute the read without waiting for data to be written. If there is no data in the FIFO, then the value of Elements Remaining will be zero.

Since the DMA FIFO Read method actively uses the CPU while waiting for data to be returned, the polling method will free up the CPU more since the Number of Elements input to the DMA FIFO Read method will never be greater than the number of elements that are already there and the function will never be waiting for more data to be written to the FIFO.

Polling with a Fixed Number of Elements

For the polling with a fixed number of elements method of reading data from a DMA FIFO, the user defines the rate at which data will be obtained by the DMA FIFO Read method and the number of elements that will be returned, as shown in Figure 9-6.



A FPGA Block Diagram	B Host Block Diagram

Figure 9-6. Polling with a Fixed Number of Elements Method

By specifying a Timeout value for the DMA FIFO Read method, we are controlling the rate at which the DMA FIFO Read executes. Specifying the Number of Elements that are read from the FIFO controls the amount of data that is returned in each iteration of the loop.

For this example, no processing is taking place aside from displaying the data on the front panel. Checking whether or not the Data array is empty tells us whether or not the DMA FIFO Read timed out. If there is no data returned, then the read timed out and we do not have to execute the processing code, which should reside in the False case of the Case Structure. You could handle instances in which no data was returned in the True case.

It is generally easier to process a fixed amount of data being returned at a fixed rate than to process a fixed amount of data at a variable rate (blocking) or a variable amount of data at a fixed rate (polling). This architecture provides the benefit of being able to control both aspects of the acquisition.

C. Lossless Data Transfer

When transferring data between the FPGA and the Host using a DMA FIFO, it is still possible to lose data if you do not take time to consider the size of the FIFO that you are defining and how you will develop your code to ensure that data is not lost.

Selecting FIFO Size

When you configure a FIFO in the FIFO Properties dialog box, you specify the number of elements that can be contained in the FPGA part of the FIFO. To save resources, specify the smallest depth that is reasonable for your application. Compare the rate at which data points are put into the FPGA FIFO to the rate at which the host application will be able to read points out. The slower the host VI is compared to the FPGA VI, the larger a FIFO you must allocate.

The default size of the host computer part of the FIFO is 10,000 elements. You can specify the depth of the host computer part of the FIFO in a host VI with the Invoke Method function set to the Configure method.

Make sure the host computer and FPGA parts of the FIFO are large enough that they do not fill in the instance of the largest delay in the host VI. For example, PCI bus traffic can cause delays in automatic transfers from the FPGA target to the host. Typically, the faster the transfer rate, the more depth you need.

Ensuring Lossless Data Transfer

To ensure lossless data transfer in your application, you must monitor overflow (too much data for the buffer to handle) or underflow (not enough data is being sent). Overflow occurs when the FIFO is full when the DMA FIFO Write method is called. Underflow occurs if the FIFO Read times out waiting for data on the host side. To guarantee lossless data transfer between an FPGA and host, overflow and underflow conditions must not occur.

Overflow

If the FIFO is ever filled as a result of data being written faster than it is read, a data overflow occurs and you have potentially lost data points. If the FIFO fills, you can use one or more of the following techniques to solve the problem:

- Reduce the rate at which you write data to the FIFO
- Increase the Requested Number of Elements to read on the host
- Increase the rate that the host reads data
- Increase FIFO buffer sizes on the FPGA and/or host

If possible, you should also attempt to reduce the load on the CPU. The speed of the CPU and the presence of competing tasks both have a negative impact on the transfer rate from the host buffer to application memory.

Underflow

If underflow occurs, there is not enough data present to read before the timeout expired and the host DMA FIFO Read method generates an error. You can use one or more of the following techniques to solve the problem:

- Increase the timeout of the DMA FIFO Read method
- Reduce the rate at which the host reads data
- Reduce the Requested Number of Elements to read

Recovering from a FIFO Overflow/Underflow

By monitoring overflow and underflow during testing, you can detect and fix either of these conditions. To detect a FIFO overflow, you can monitor the Timed Out? output of the FIFO Method node in the FPGA VI, as shown in Figure 9-7.

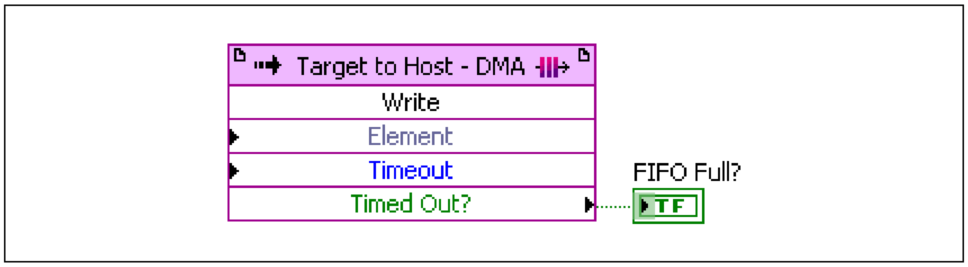


Figure 9-7. Detecting a FIFO Overflow

When a FIFO underflow occurs, the DMA Read method returns error –50400. To detect a FIFO underflow, you can monitor the error output from the host DMA FIFO Read method, checking for error –50400, as shown in Figure 9-8.

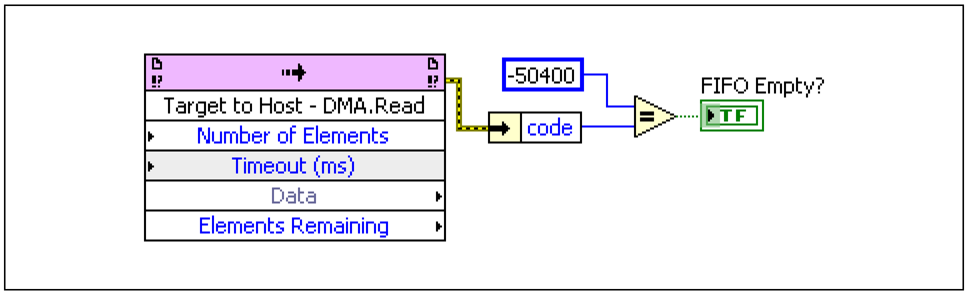


Figure 9-8. Detecting a FIFO Underflow

In addition to monitoring, you may want to take some action such as stopping the application if either problem occurs. You can use a Compound Or function to stop execution of both the FPGA VI and the host VI when an overflow or underflow condition occurs.

When you execute the FPGA VI on the development computer, FIFOs reset when the VI stops and restarts. When you change the execution mode to execute the VI on the FPGA target, FIFOs do not reset when the FPGA VI stops and restarts. To prevent old data from interfering with current operations you should clear the contents of the FIFO. There are two ways to clear the contents of the FIFO from the host VI:

- Reset the FPGA VI to discard the data in the FIFO.
- Flush the buffer to read the remaining elements of the FIFO.

Resetting the FPGA VI resets the FPGA controls and indicators to their default values and clears the FIFO. You can reset the FPGA VI by using an Invoke Method configured to call the Reset method, as shown in Figure 9-9 or by right-clicking on the Close FPGA VI Reference function and selecting **Close and Reset if Last Reference**, as shown in Figure 9-10.

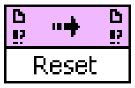


Figure 9-9. Reset Method

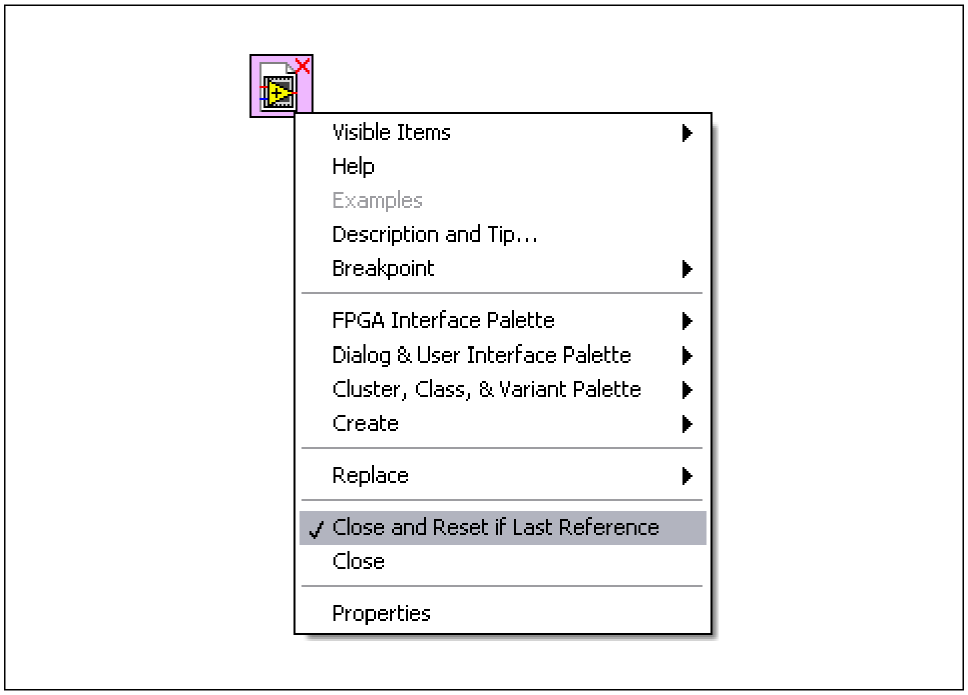


Figure 9-10. Close and Reset if Last Reference

However, there may be data remaining in the FIFO that you do not want to lose on stop. Figure 9-11 shows a technique to recover the remaining data and flush the buffer on the host.

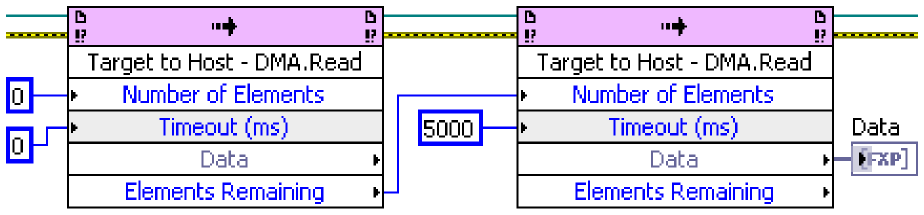


Figure 9-11. Flushing the Buffer

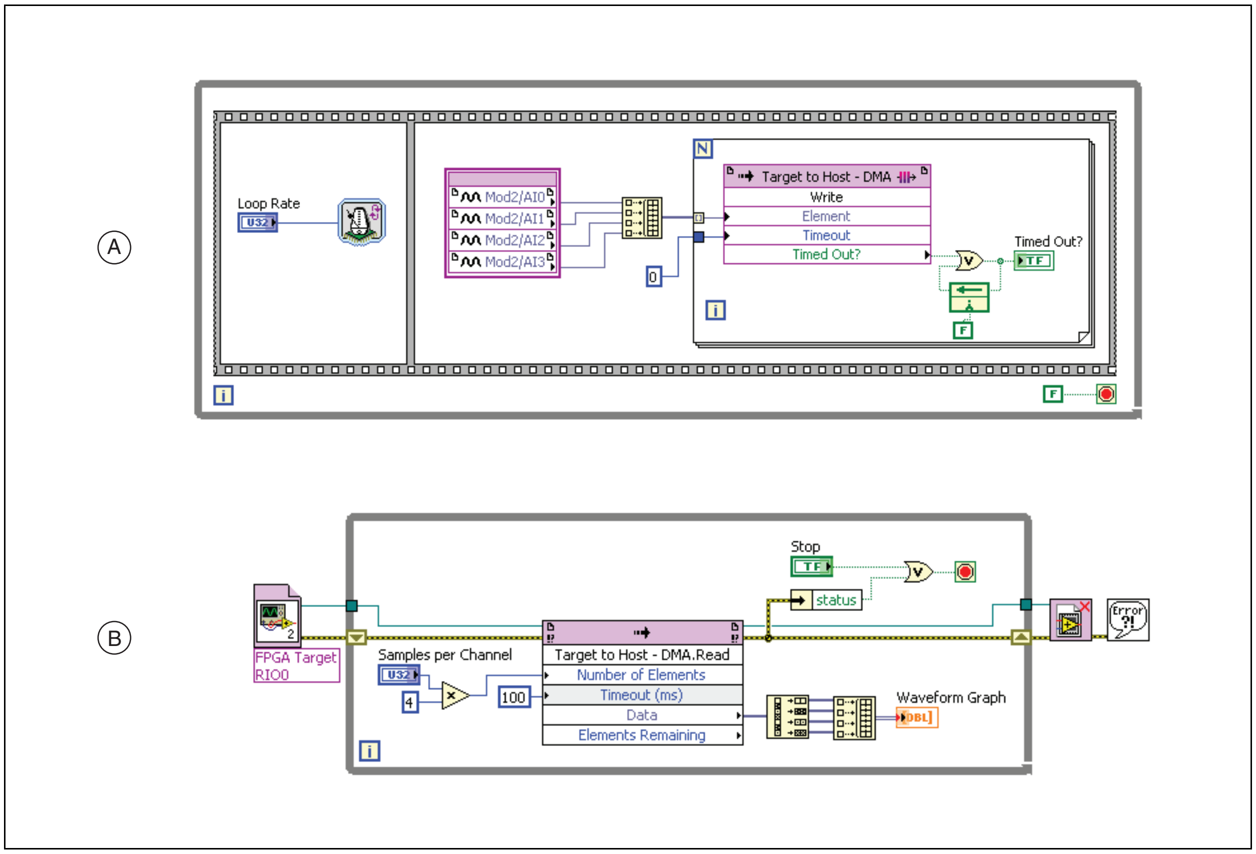
Call two DMA FIFO Read methods in sequence from the host VI. Set the first node to read `0` elements with `0` timeout to get the number of elements remaining in the FIFO. Wire the Elements Remaining output to the second DMA FIFO Read method and wire the Data output to an appropriate process.

D. Interleaving

A limited number of DMA channels are available on your FPGA device. Refer to the documentation or the product page on `ni.com` for your device to determine the number of DMA channels that are provided. Since DMA channels are limited, it may be necessary to write data from multiple sources to a single DMA channel. One way to do this is to build an array of data elements from the different sources and write them to the DMA channel. This technique is referred to as interleaving.

Note The data that you interleave should all be of the same data type to avoid unintentional coercion when you write to the FIFO.

Interleaving enables you to write data from multiple I/O resources on an FPGA to a single DMA channel, and then read them back in a host VI as shown in Figure 9-12.



A FPGA Block Diagram	B Host Block Diagram

Figure 9-12. Interleaving Data from Multiple I/O Resources

On the FPGA VI, four channels of data are combined into an array. The array is passed into a For Loop, which indexes the array, decimating the values and passing them individually and sequentially into a DMA FIFO.

On the host VI, the Samples per Channel input is multiplied by the number of channels used in the FPGA VI to determine the Number of Elements that the Read method will obtain from the DMA FIFO. The array of data that is returned is then divided into multiple arrays, each representing a single channel of the acquisition. Once you have separated the array of data returned by the DMA FIFO Read method, you can perform any additional processing on the data for each channel. In this case, the data is built into a two-dimensional array and displayed on the front panel.

Self-Review: Quiz

1. Identify whether each of the following statements describes communication using front panel controls/indicators or a DMA FIFO.
    a. Well-suited to streaming large amounts of data
    b. Consumes significant CPU resources
    c. Only the most recent data is transferred
    d. Suitable for use in asynchronous applications

2. You have developed an application that acquires data and uses a DMA FIFO to transfer data to a host VI. However, the DMA FIFO repeatedly fills, resulting in an overflow condition. Which of the following techniques can be used to address this problem? (Multiple Answers)
    a. Reduce the speed of acquisition on the FPGA VI
    b. Increase the number of elements to read on the host
    c. Decrease the rate at which the host reads data
    d. Increase the rate at which the host reads data

3. Match each target to host architecture to the description that best fits it.

Polling with Fixed Number of Elements	Host VI DMA FIFO Read reads a fixed number of elements at a user-defined rate
Polling	Host VI DMA FIFO Read waits indefinitely to read a fixed number of elements from the FIFO
Blocking	Host VI DMA FIFO Read reads all of the available elements in the FIFO at a user-defined rate

Self-Review: Quiz Answers

1. Identify whether each of the following statements describes communication using front panel controls/indicators or a DMA FIFO.
   a. Well-suited to streaming large amounts of data—**DMA FIFO**
   b. Consumes significant CPU resources—**front panel controls/indicators**
   c. Only the most recent data is transferred—**front panel controls/indicators**
   d. Suitable for use in asynchronous applications—**DMA FIFO**

2. You have developed an application that acquires data and uses a DMA FIFO to transfer data to a host VI. However, the DMA FIFO repeatedly fills, resulting in an overflow condition. Which of the following techniques can be used to address this problem? (Multiple Answers)
   a. Reduce the speed of acquisition on the FPGA VI
   b. Increase the number of elements to read on the host
   c. Decrease the rate at which the host reads data
   d. Increase the rate at which the host reads data

3. Match each target to host architecture to the description that best fits it.

Polling with Fixed Number of Elements	**Host VI DMA FIFO Read reads a fixed number of elements at a user-defined rate**
Polling	**Host VI DMA FIFO Read reads all of the available elements in the FIFO at a user-defined rate**
Blocking	**Host VI DMA FIFO Read waits indefinitely to read a fixed number of elements from the FIFO**

Notes

10 Modular Programming

In this lesson, you learn how to use subVIs most efficiently in your FPGA application. You learn when to set your VIs as re-entrant or non-reentrant, depending on your FPGA needs. You also learn about FPGA controls so that you can reference FIFOs, memory, and I/O nodes in subVIs.

Topics

A. Review of SubVIs

For more about the information in the following sections, refer to the *Getting Started with LabVIEW* manual.

Understanding Modularity

Modularity defines the degree to which a program is composed of discrete modules such that a change to one module has minimal impact on other modules. Modules in LabVIEW are called subVIs.

A VI within another VI is called a subVI. A subVI corresponds to a subroutine in text-based programming languages. When you double-click a subVI, a front panel and block diagram appear, rather than a dialog box in which you can configure options. The front panel includes controls and indicators. The block diagram includes wires, front panel icons, functions, possibly subVIs, and other LabVIEW objects that also might look familiar. The upper right corner of the front panel window and block diagram window displays the icon for the VI. This icon is the same as the icon that appears when you place the VI on the block diagram.

As you create VIs, you might find that you perform a certain operation frequently. Consider using subVIs or loops to perform that operation repetitively.

Building the Icon and Connector Pane

After you build a VI front panel and block diagram, build the icon and the connector pane so you can use the VI as a subVI. The icon and connector pane correspond to the function prototype in text-based programming languages. Every VI displays an icon in the upper right corner of the front panel and block diagram windows.

A VI icon is a graphical representation of a VI. It can contain text, images, or a combination of both. If you use a VI as a subVI, the icon identifies the subVI on the block diagram of the VI. If you add the VI to a palette, the VI icon also appears on the Functions palette. You can double-click the icon in the front panel window or block diagram window to customize or edit it.

Note Customizing the icon is recommended, but optional. Using the default LabVIEW icon does not affect functionality.

You also must build a connector pane to use the VI as a subVI. The connector pane is a set of terminals that correspond to the controls and indicators of that VI, similar to the parameter list of a function call in text-based programming languages. The connector pane defines the inputs and outputs you can wire to the VI so you can use it as a subVI. A connector pane receives data at its input terminals and passes the data to the block diagram code through the front panel controls and receives the results at its output terminals from the front panel indicators.

Setting Required, Recommended, and Optional Inputs and Outputs

In the **Context Help** window, the labels of required terminals appear bold, recommended terminals appear as plain text, and optional terminals appear dimmed. The labels of optional terminals do not appear if you click the **Hide Optional Terminals and Full Path** button in the Context Help window.

You can designate which inputs and outputs are required, recommended, and optional to prevent users from forgetting to wire subVI terminals.

Right-click a terminal in the connector pane and select **This Connection Is** from the shortcut menu. A checkmark indicates the terminal setting. Select **Required**, **Recommended**, or **Optional**. You also can select **Tools»Options»Front Panel** and put a checkmark in the **Connector pane terminals default to required** checkbox. This option sets terminals on the connector pane to Required instead of Recommended. This applies to connections made using the wiring tool and to subVIs created using **Create SubVI**.

For terminal inputs, required means that the block diagram on which you placed the subVI will be broken if you do not wire the required inputs. Required is not available for terminal outputs. For terminal inputs and outputs, recommended or optional means that the block diagram on which you placed the subVI can execute if you do not wire the recommended or optional terminals. If you do not wire the terminals, the VI does not generate any warnings.

Inputs and outputs of VIs in vi.lib are already marked as **Required**, **Recommended**, or **Optional**. LabVIEW sets inputs and outputs of VIs you create to Recommended by default. Set a terminal setting to required only if the VI must have the input or output to run properly.

Creating a SubVI from an Existing VI

You can simplify the block diagram of a VI by converting sections of the block diagram into subVIs. Convert a section of a VI into a subVI by using the Positioning tool to select the section of the block diagram you want to reuse and selecting **Edit»Create SubVI**. An icon for the new subVI replaces the selected section of the block diagram. LabVIEW creates controls and indicators for the new subVI, automatically configures the connector pane based on the number of control and indicator terminals you selected, and wires the subVI to the existing wires.

The new subVI uses a default pattern for the connector pane and a default icon. Double-click the subVI to edit the connector pane and icon, and to save the subVI.

Note Do not select more than 28 objects to create a subVI because 28 is the maximum number of connections on a connector pane. If your front panel contains more than 28 controls and indicators that you want to use programmatically, group some of them into a cluster and assign the cluster to a terminal on the connector pane.

B. Using SubVIs on the FPGA

LabVIEW allows you to encapsulate common sections of code as subVIs to facilitate their reuse on the block diagram. While you can run only one top-level FPGA VI, you can use multiple VIs on the FPGA by placing subVIs on the block diagram of the top-level FPGA VI.

Front panel objects in subVIs do not communicate directly with the host VI and therefore do not consume additional space on the FPGA. SubVI controls and indicators that must hold state information from one call to another use registers to store data. SubVI controls and indicators that only pass data into and out of a subVI consume no logic resources on the FPGA.

C. Reentrancy and Non-reentrancy in FPGA

You can configure a subVI as a single instance shared among multiple callers, also known as a non-reentrant subVI. You also can configure a subVI as reentrant to allow parallel execution. By default, VIs created under an FPGA target are reentrant. To make a subVI non-reentrant, select **Execution** from the Category pull-down menu of the VI Properties dialog box and remove the checkmark from the **Reentrant execution** checkbox.

Note VIs not created under an FPGA target or created using the FPGA Module 8.2.x or earlier are non-reentrant by default. Moving or copying VIs to or from an FPGA target does not change the reentrancy setting of the VI.

If you use a non-reentrant subVI in an FPGA VI, only a single copy of the subVI becomes a hardware resource and all callers share the hardware resource. If you use a reentrant subVI in an FPGA VI, each instance of the subVI on the block diagram becomes a separate hardware resource. For example, if you have five instances of an event counter configured as a reentrant subVI on the block diagram, LabVIEW implements five independent copies of the event counter hardware on the FPGA.

In general, configuring a subVI as reentrant improves speed because instances of the subVI execute in parallel. On the other hand, configuring a subVI as non-reentrant generally saves space because each instance of the subVI shares the same FPGA hardware. However, when you use multiple instances of a non-reentrant subVI, extra FPGA resources are used for arbitration. In some cases, the arbitration resources can cause a non-reentrant configuration to consume as much or more space on the FPGA as a reentrant configuration.

Table 10-1 summarizes the typical advantages and disadvantages of non-reentrant and reentrant subVIs.

Note When possible, avoid shared resources in reentrant subVIs. If you use a shared resource in a reentrant subVI, each instance of the subVI must use arbitration to access the shared resource, which can impede parallel execution.

Table 10-1. Advantages and Disadvantages of Reentrant and Non-reentrant SubVIs

VI Type	FPGA Speed	FPGA Utilization
Reentrant	Higher—Multiple calls to the same subVI run in parallel	Can be higher because each instance of the subVI on the block diagram uses space on the FPGA. However, reentrant subVIs do not use FPGA resources for arbitration.
Non-reentrant	Lower—Each call to the subVI waits until the previous call ends	Can be lower because only one instance of the subVI exists on the FPGA no matter how many times you use it. However, non-reentrant subVIs use FPGA resources for arbitration.

D. Name Controls for Passing to subVIs

The FPGA Module includes the following name controls you can use to create reusable subVIs with I/O, FIFO, memory, and clocks. Use these FPGA name controls to reference I/O, FIFOs, memory, and clocks in subVIs.

- FPGA I/O control—Use FPGA I/O controls to create reusable subVIs with configurable I/O items.
- FPGA Clock control—Use FPGA clock controls to pass FPGA I/O clocks to a subVI.
- FIFO control—Use FIFO controls to specify a FIFO item on the block diagram.
- Memory control—Use memory controls to specify a memory item on the block diagram.

You also can use the following constants to create reusable subVIs with I/O, FIFO, memory, and clocks.

- FPGA I/O constant
- FPGA Clock constant
- FIFO constant
- Memory constant

The following restrictions apply to FPGA name controls:

- You cannot change the value of FPGA name controls and indicators while an FPGA VI runs on development computer or when using Interactive Front Panel communication.
- You cannot access FPGA controls and indicators from the FPGA Interface.
- You can bundle FPGA controls into clusters with other FPGA controls, but not with other data types.

You can change the value of an FPGA control on the front panel only when the VI is not running. This is because FPGA control refnums must be constant references during run-time.

E. Testing FPGA SubVIs

Test subVIs as top-level VIs first. You should test the logic of the FPGA subVI by executing the FPGA subVI as a top-level FPGA VI on a development computer with simulated I/O. Then test the FPGA subVI by executing the FPGA subVI as a top-level FPGA VI on the FPGA target.

After you finish testing the FPGA subVI as a top-level FPGA VI, test the top-level FPGA VI that calls the FPGA subVI.

F. LabVIEW FPGA IPNet

The LabVIEW FPGA IPNet is an online resource for browsing, understanding, and downloading LabVIEW FPGA functions or IP (intellectual property). LabVIEW FPGA IPNet contains a collection of FPGA IP and examples gathered from the LabVIEW FPGA function palette, internal National Instruments developers, and the LabVIEW FPGA community. You should use this resource to acquire IP that you need for your application, download examples to help learn programming techniques, and explore the depth of IP offered by the LabVIEW FPGA platform. In addition to exploring what is offered here, you can also share your LabVIEW FPGA IP or submit an update to existing IP for the LabVIEW community.

Self-Review: Quiz

1. Controls and indicators on an FPGA subVI are exposed to the host VI.
   a. True
   b. False

2. Which of the following are true for reentrant subVIs in LabVIEW FPGA?
   a. Default configuration of VIs created under an FPGA target
   b. Each instance on the block diagram becomes a separate hardware resource
   c. Optimized for FPGA size
   d. Each call to the subVI waits until the previous call ends
   e. Multiple calls to the same subVI run in parallel

3. Which of the following are FPGA name controls that can be passed into FPGA subVIs?
   a. FPGA I/O control
   b. FPGA Clock control
   c. FPGA FIFO control
   d. FPGA Memory control

Self-Review: Quiz Answers

1. Controls and indicators on an FPGA subVI are exposed to the host VI.
   a. True
   b. False

2. Which of the following are true for reentrant subVIs in LabVIEW FPGA?
   a. Default configuration of VIs created under an FPGA target
   b. Each instance on the block diagram becomes a separate hardware resource
   c. Optimized for FPGA size
   d. Each call to the subVI waits until the previous call ends
   e. Multiple calls to the same subVI run in parallel

3. Which of the following are FPGA name controls that can be passed into FPGA subVIs?
   a. FPGA I/O control
   b. FPGA Clock control
   c. FPGA FIFO control
   d. FPGA Memory control

Notes

A

Pipelining

Another important technique for improving FPGA performance is pipelining. Pipelining breaks up code within a loop so that operations are performed in different cycles of the same loop. You therefore reduce the code length within each iteration. This technique is particularly useful to reduce the length of each run through the SCTL.

Topics

A. Pipelining

Pipelining is a technique where you pass the results from one calculation to the next on the following iteration of a loop. The process of pipelining code begins with identifying combinatorial paths in your code. A combinatorial path is a set of logic between the output of one register and the input of another register. Because data in the registers is updated with every rising edge of the clock, if there are too many operations between two registers, a VI compilation may fail due to a timing error. Figure A-1 shows an example of a combinatorial path inside a single-cycle Timed Loop.

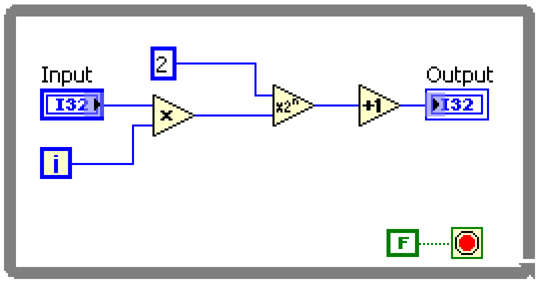


Figure A-1. VI with no Pipelining

Pipelining shortens the length between the output and input registers of a While Loop so that your VI meets timing requirements. You can use shift registers to run portions of your combinatorial path in different cycles of your loop. Pipelining is especially important in single-cycle Timed Loops where the entire path is required to execute in one clock cycle. Figure A-2 illustrates a pipelined version of Figure A-1.

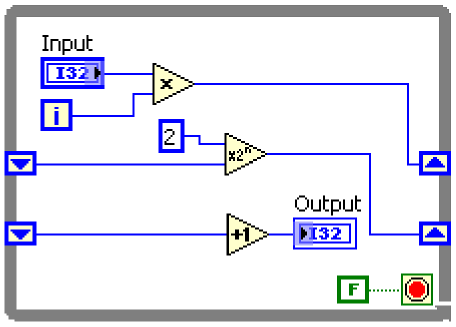


Figure A-2. Use Pipelining to Eliminate Combinatorial Path

Pipelining increases system latency because the input of a function is based on the output of a previous cycle of the loop. However, the latency disappears when the pipe is full. After only a few loop cycles, pipelined code is significantly more efficient than identical code in a normal loop. Figure A-3 illustrates latency due to pipelining.

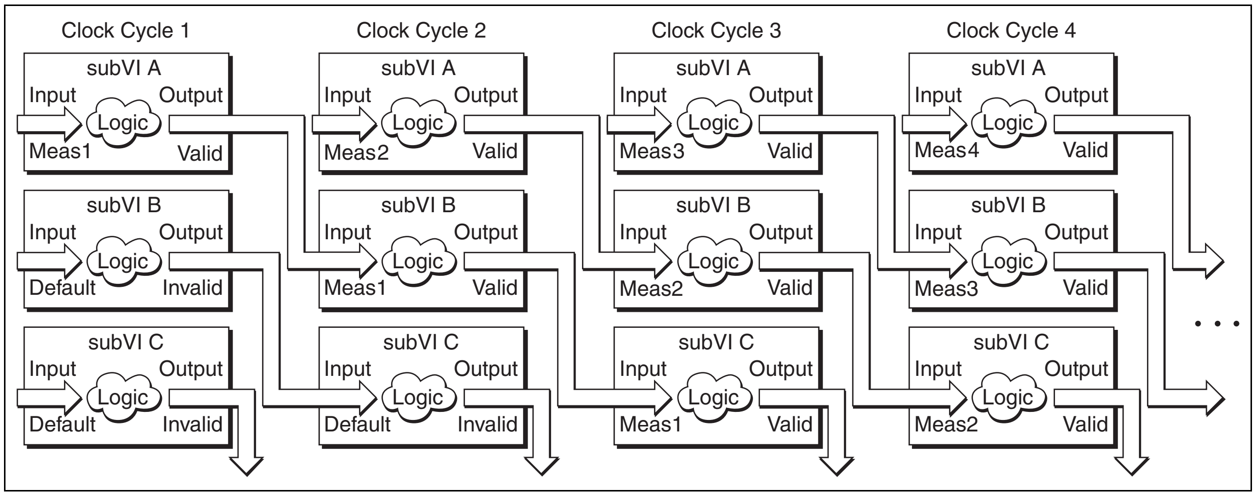


Figure A-3. Increased Latency Due to Pipelining

After Clock Cycle 1, the output of subVI A is valid and the output of subVIs B and C are invalid. After Clock Cycle 2, subVIs A and B have valid output and subVI C has invalid output. After the Clock Cycle 3 and all subsequent clock cycles, all output will be valid.

Feedback Nodes

Feedback Nodes are identical in functionality to a shift register, and are often preferable from a user standpoint because they look similar to the initial code. Figure A-4 shows an example of using Feedback Nodes instead of shift registers.

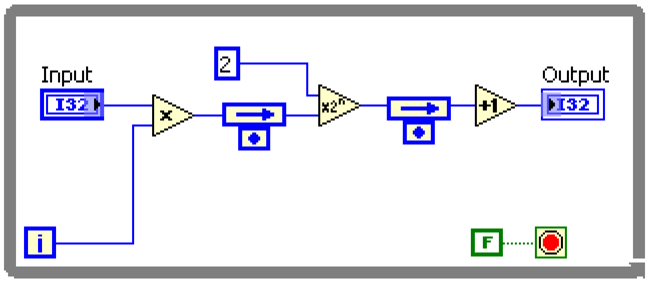


Figure A-4. VI with Feedback Nodes

Feedback Nodes use a value wired to the initializer terminal as the initial value for the first iteration or execution. The Feedback Node then stores the previous iteration result for each subsequent execution. If you do not wire a value to the initializer terminal, the Feedback Node uses the default value for the data type and continues building on previous results in subsequent executions.

You can use a Feedback Node to implement a pipeline and reduce long combinatorial paths. When you use the Feedback Node inside a Case structure, the Feedback Node updates data only on clock cycles when the owning subdiagram executes.

The Feedback Node is implemented as a register and requires logic resources in proportion to the width of the data type. Using the initialization terminal slightly increases logic resource usage.

Drawbacks

When you implement a pipeline, the output of the final step lags behind the input by the number of steps in the pipeline and the output is invalid for each clock cycle until the pipeline fills. The number of steps in a pipeline is called the pipeline depth, and the latency of a pipeline, measured in clock cycles, corresponds to its depth. For a pipeline of depth *N*, the result is invalid until the N^{th} loop iteration, and the output of each valid loop iteration lags behind the input by *N–1* iterations.

Notes

Notes

B

Additional Information and Resources

This appendix contains additional information about National Instruments technical support options and LabVIEW FPGA 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. areas. Visit `ni.com/training` for more information about the NI certification program.