Contents

CHAPTER 5
Customizing Hardware Through LabVIEW FPGA

This chapter covers several best practices in addition to advanced tips and tricks for developing high-performance control and monitoring systems with the LabVIEW FPGA Module and CompactRIO. It examines recommended programming practices, ways to avoid common mistakes, and numerous methods to create fast, efficient, and reliable LabVIEW FPGA applications.

FPGA Technology

FPGAs offer a highly parallel and customizable platform that you can use to perform advanced processing and control tasks at hardware speeds. An FPGA is a programmable chip composed of three basic components: logic blocks, programmable interconnects, and I/O blocks.

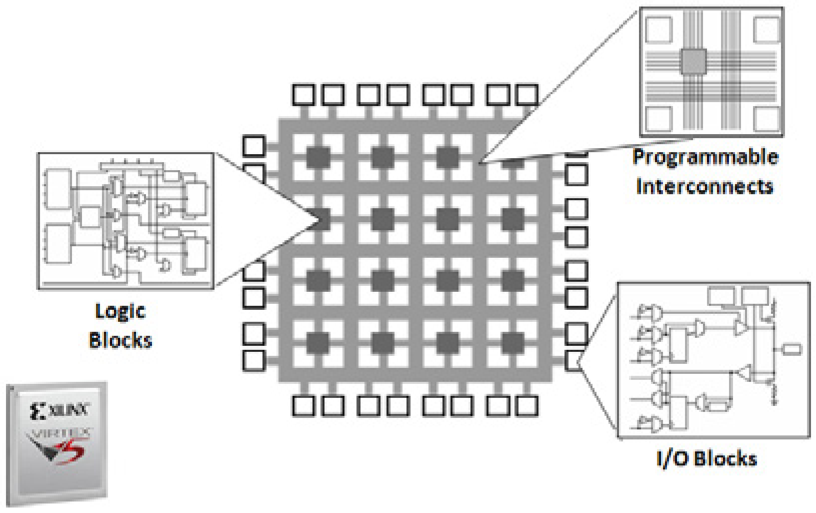


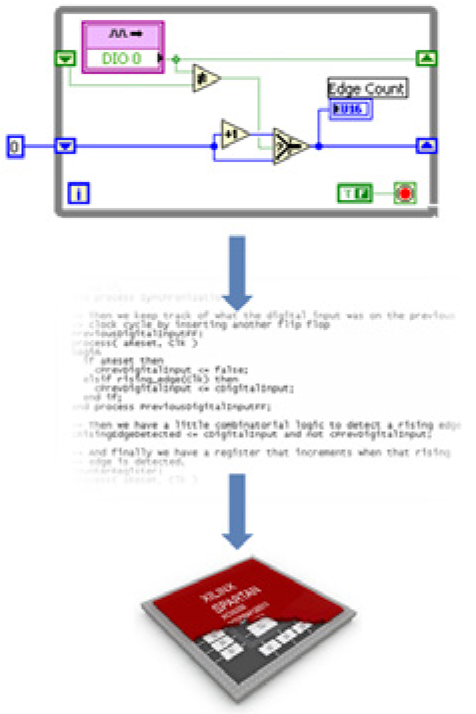
Figure 5.1. An FPGA is composed of configurable logic and I/O blocks tied together with programmable interconnects.

The logic blocks are a collection of digital components such as lookup tables, multipliers, and multiplexers where digital values and signals are processed to generate the desired logical output. These logic blocks are connected with programmable interconnects that route signals from one logic block to the next. The programmable interconnect can also route signals to the I/O blocks for two-way communication to surrounding circuitry. For more information on FPGA hardware components, see the LabVIEW FPGA Help Document: Introduction to FPGA Hardware Concepts.

FPGAs are clocked at relatively lower rates than CPUs and GPUs, but they make up for this difference in clock rate by allowing you to create specialized circuitry that can perform multiple operations within a clock cycle. Combine this with their tight integration with I/O on NI reconfigurable I/O (RIO) devices and you can achieve much higher throughput and determinism as well as faster response times than with a processor-only solution. This helps you tackle high-speed streaming, digital signal processing (DSP), control, and digital protocol applications.

Because LabVIEW FPGA VIs are synthesized down to physical hardware, the FPGA compile process is different from the compile process for a traditional LabVIEW for Windows or LabVIEW Real-Time application. When writing code for the

FPGA, you write the same LabVIEW code as you do for any other target, but when you select the **Run** button, LabVIEW internally goes through a different process. First, LabVIEW FPGA generates VHDL code and passes it to the Xilinx compiler. Then the Xilinx compiler synthesizes the VHDL and places and routes all synthesized components into a bitfile. Finally, the bitfile is downloaded to the FPGA and the FPGA assumes the personality you have programmed. This process, which is more complicated than other LabVIEW compiles, can take up to several hours depending on how intricate your design is. Later in the chapter, learn more about debugging and simulating your FPGA VI so that you can compile less often.



```
-- process synchronization
-- Then we keep track of what the digital input was on the previous
-- clock cycle by inserting another flip flop
PreviousDigitalInputFF:
process( aReset, Clk )
begin
  if aReset then
    cPrevDigitalInput <= false;
  elsif rising_edge(Clk) then
    cPrevDigitalInput <= cDigitalInput;
  end if;
end process PreviousDigitalInputFF;

-- Then we have a little combinatorial logic to detect a rising edge
cRisingEdgeDetected <= cDigitalInput and not cPrevDigitalInput;

-- and finally we have a register that increments when that rising
-- edge is detected.
CounterRegister:
process( aReset, Clk )
```

Figure 5.2. Behind the scenes, the LabVIEW FPGA compiler translates LabVIEW code into VHDL and invokes the Xilinx compile tools to generate a bitfile, which runs directly on the FPGA.

Establishing a Design Flow

Depending on the complexity of your LabVIEW FPGA application, you may want to quickly write a program and compile it down to hardware, or you may want to leverage the built-in simulator to debug, test, and verify your code without having to compile to hardware every time you make a change. This section of the guide outlines one example of a recommended design flow to enhance your productivity while programming in LabVIEW FPGA.

1. Establish functional and performance requirements (covered in Introduction and Basic Architectures)
2. Design a software architecture (covered in Introduction and Basic Architectures)
3. Implement LabVIEW FPGA code
4. Test and debug LabVIEW FPGA code
5. Optimize LabVIEW FPGA code
6. Compile LabVIEW FPGA code to hardware
7. Deploy your system

The next few sections reflect this recommended design flow, starting with implementing your LabVIEW FPGA code. The first two topics are discussed in the first section of the LabVIEW for CompactRIO Developer's Guide, Introduction and Basic Architectures.

Best Practices for Implementing LabVIEW FPGA Code

When you begin development, you should create any VIs underneath the FPGA target in the LabVIEW project so you can program with the LabVIEW FPGA palette, which is a subset of the LabVIEW palette plus some LabVIEW FPGA-specific functions.

You also should develop your VIs in simulation mode by right-clicking **FPGA Target** and selecting **Execute VI on»Development Computer with Simulated I/O**. By taking this approach, you can quickly iterate on your design and access all of the standard LabVIEW debugging features. If you need to access real-world I/O, change the execution mode to **Execute VI on»FPGA Target**.

Reading and Writing I/O

This section covers the basics of accessing I/O through a LabVIEW FPGA VI. For more detailed information on timing and synchronization with analog and digital I/O modules using LabVIEW FPGA, see Chapter 6: Timing and Synchronization of I/O.

To develop a LabVIEW FPGA application, you need to add your FPGA target to a LabVIEW project in addition to any necessary I/O, clocks, register items, memory items, or FIFOs. The LabVIEW Help document Using FPGA Targets in a LabVIEW Project provides detailed instructions on how to set this up.

Once your LabVIEW project is configured to target an FPGA device, you can drag and drop an I/O channel from the LabVIEW project onto the LabVIEW FPGA VI block diagram to get an I/O node.

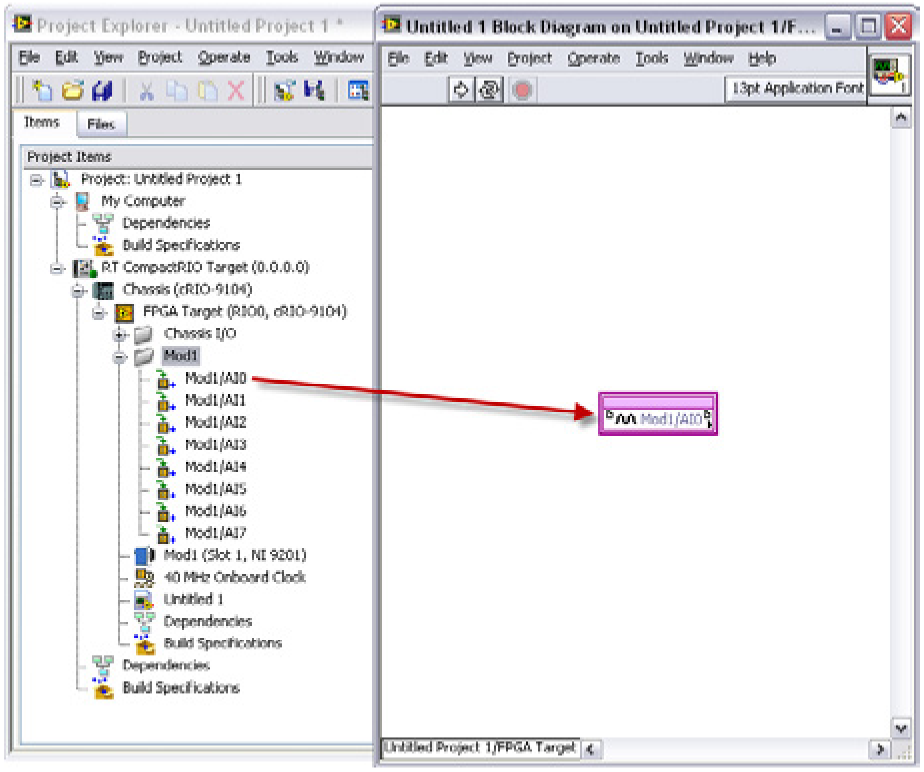


Figure 5.3. Drag and drop an I/O node to the FPGA block diagram.

The FPGA I/O Node returns single-point data when called. For most NI R Series devices and C Series I/O modules, you can use LabVIEW structures and logic to specify the sample rate and triggering. High-speed analog input modules that use delta-sigma converters and have their own onboard clock are an exception. For these modules, you control the sample rate with a Property Node. Table 6.2 in Chapter 6: Timing and Synchronization of I/O provides a list of NI C Series I/O modules that use delta-sigma modulation.

For most I/O nodes, you can use a loop with a Loop Timer Express VI to set the sample rate. The Sample Delay control sets the rate in ticks. Ticks are the pulses on the FPGA clock, which by default is 40 MHz for most CompactRIO targets. For example, implementing a sample delay of 20,000 ticks of the 40 MHz clock renders a sample rate of 2 kHz. You can also specify the sample delay in microseconds or milliseconds by double-clicking the Loop Timer and adjusting the configuration panel. Of course, no matter what rate you specify, the module samples up to only the maximum rate specified in the module documentation.

You also have an option to use a Sequence structure, as shown in Figure 5.4. A Sequence structure can be used to guarantee that every time the LabVIEW FPGA VI is compiled, the timer and I/O Node are executed in the same sequence.

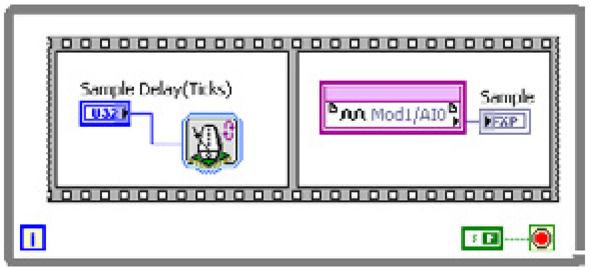


Figure 5.4. The Acquisition Scheme for a Standard Analog Input Module

To implement a triggered application, use a Case structure gated on the trigger condition. This trigger condition can be an input from the host processor or it can be derived from logic directly on the FPGA. With LabVIEW FPGA, you can implement basic or complex triggering schemes such as retriggering, pause triggering, or any type of custom triggering. Figure 5.5 shows an example of an application that takes in a start trigger and monitors the number of samples being collected until it reaches the number specified by the user. The code used in this example can be found in the Turbine Tester project, which can be downloaded from the Introduction and Basic Architectures section at ni.com/compactriodevguide.

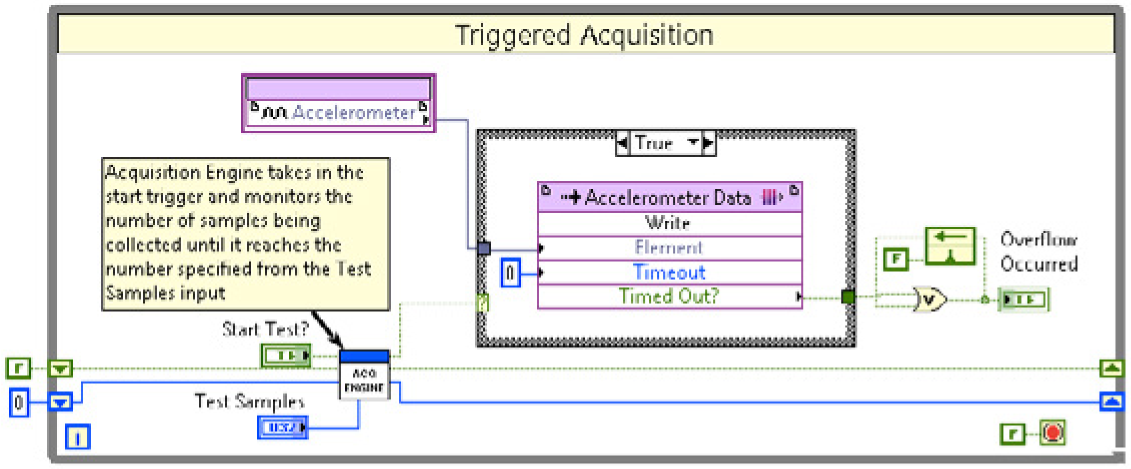

Triggered Acquisition
Accelerometer
True
Accelerometer Data
Write
Element
Timeout
Timed Out?
Acquisition Engine takes in the start trigger and monitors the number of samples being collected until it reaches the number specified from the Test Samples input
Start Test?
ACQ ENGINE
Test Samples
U32
Overflow Occurred

Figure 5.5. Simple One-Shot Trigger

Synchronize Your Loops

For most control and monitoring applications, the timing of when the code executes is important to the performance and reliability of the system. In this motor control example, you have two different clock signals: a sample clock and a PID clock. These are Boolean signals you generate in the application to provide synchronization among the loops. You can trigger on either the rising or falling edge of these clock signals.

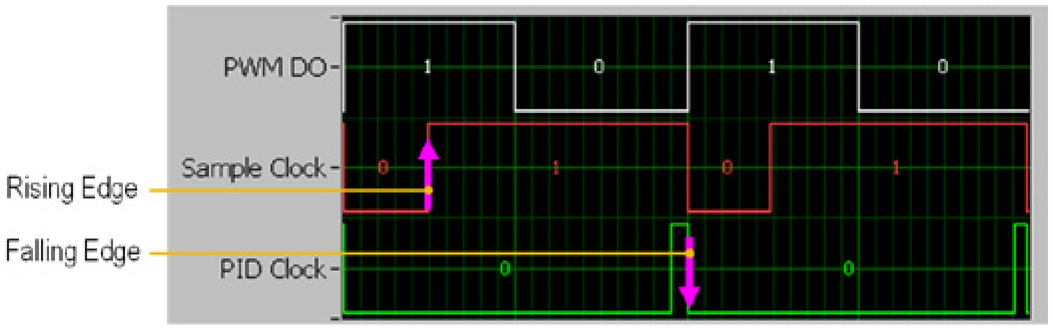


Figure 5.6. Motor Control Example With Two Different Clock Signals

Now consider the LabVIEW FPGA code used to monitor these signals and trigger on either the rising or falling edge.

Typically triggering a loop based on a Boolean clock signal works like this: first wait for the rising or falling edge to occur and then execute the LabVIEW FPGA code that you want to run when the trigger condition occurs. This is often implemented with a Sequence structure that uses the first frame to wait for the trigger and the second frame to execute the triggered code, as shown in Figure 5.7.

Rising Edge Trigger: In this case, you are looking for the trigger signal to transition from False (or 0) to True (or 1). This is done by holding the value in a shift register and using the Greater Than? function. (Note: A True constant is wired to the iteration terminal to initialize the value and avoid an early trigger on the first iteration.)

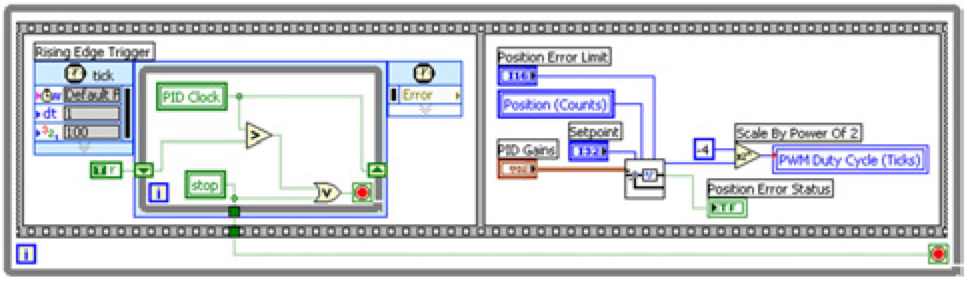


Figure 5.7. Rising Edge Trigger Example

Falling Edge Trigger: In this case, use a Less Than? function to detect the transition from True (or 1) to False (or 0). (Note: A False constant is wired to the iteration terminal to initialize the value.)

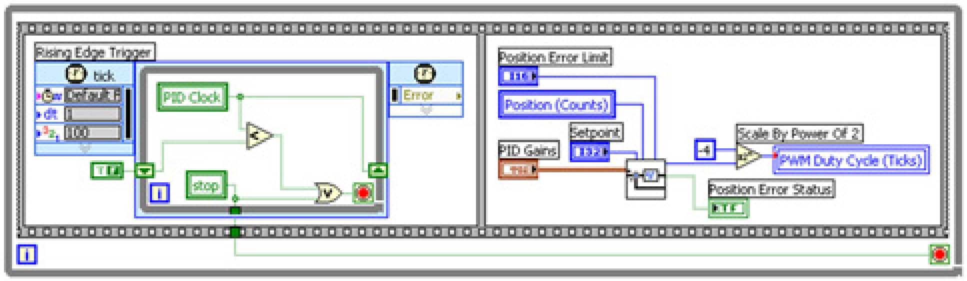


Figure 5.8. Falling Edge Trigger Example

Analog Level Trigger: Use a Greater Than? function to detect when the analog signal is greater than your analog threshold level, and then use the Boolean output of the function as your trigger signal. This case actually is a rising or falling edge detector because you are using the Not Equal? function to detect any transition.

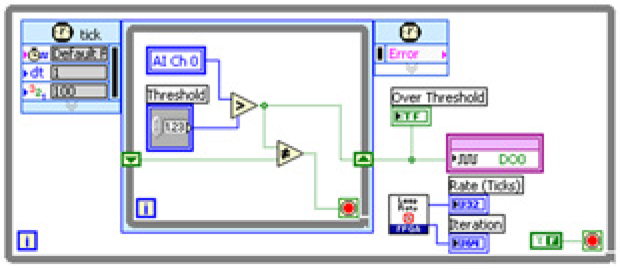


Figure 5.9. Analog Level Trigger Example

Now examine another common triggering use case—latching the value of a signal when a trigger event occurs.

Tip: Latch Values

In this case, you use a rising edge trigger to latch the Analog Input value from another loop into the Latched Analog Input register. This value is held constant until the next trigger event occurs. In this example, the actual analog input operation is occurring in another loop, and you are using a local variable for communication between the loops. (Note: Local variables are a good way to share data between asynchronous loops in LabVIEW FPGA.)

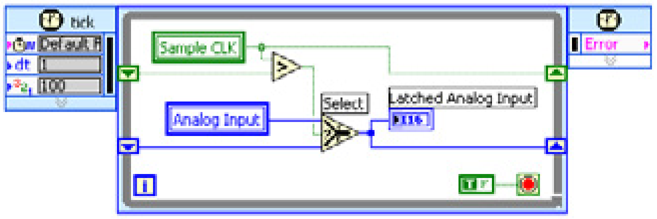


Figure 5.10. In this example, the actual analog input operation is occurring in another loop, and you are using a local variable for communication between the loops.

Creating Modular, Reusable SubVIs

Writing modular code is almost always a good idea, whether you are designing an application for a Windows, a real-time, or an FPGA device. SubVIs make code easier to debug and troubleshoot, easier to document and track changes, and typically cleaner, easier to understand, and more reusable. An example of a LabVIEW FPGA subVI is shown in Figure 5.11. This subVI counts the number of samples acquired once a trigger condition is met.

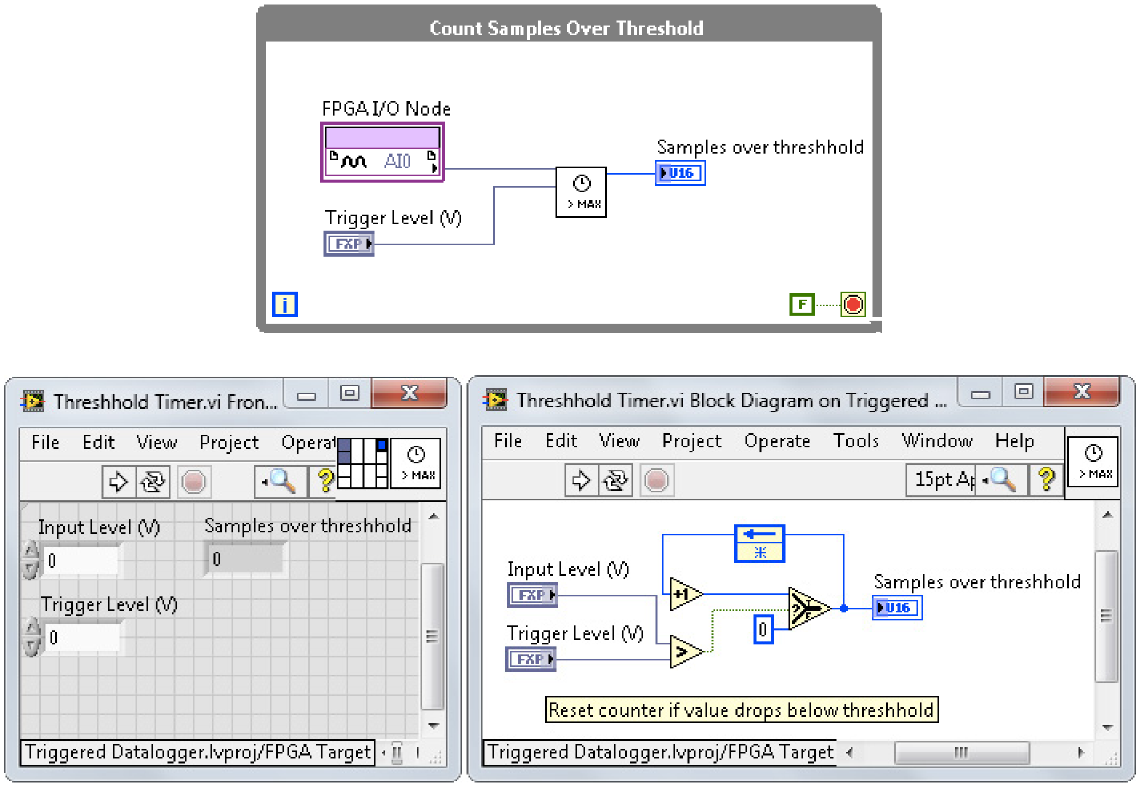

Count Samples Over Threshold
FPGA I/O Node
AI0
Samples over threshhold
U16
Trigger Level (V)
FXP
> MAX
Threshhold Timer.vi Fron...
File Edit View Project Operat
Input Level (V)
Samples over threshhold
Trigger Level (V)
Triggered Datalogger.lvproj/FPGA Target
Threshhold Timer.vi Block Diagram on Triggered ...
File Edit View Project Operate Tools Window Help
15pt Ap
Input Level (V)
Trigger Level (V)
Samples over threshhold
Reset counter if value drops below threshhold
Triggered Datalogger.lvproj/FPGA Target

Figure 5.11. A subVI is used to count the number of samples acquired once a trigger condition is met.

Items to Avoid Placing Into SubVIs

When creating subVIs, you should consider keeping some items outside your subVI, specifically I/O nodes and Loop Timer or Wait functions.

Placing I/O nodes outside subVIs makes them more modular and portable and makes the top-level diagram more readable. This also reduces extraneous I/O node instances that might otherwise be included multiple times in the subVI, resulting in unnecessary gate usage. When accessing shared resources in LabVIEW FPGA, the compiler adds extra arbitration logic necessary to handle multiple callers. Arbitration is described in more detail on page 86.

Another best practice is to avoid using Loop Timer or Wait functions within your modular subVIs. If the subVI has no delays, it executes as fast as possible and avoids slowing down the caller. Also, if you need to move your subVI into a single-cycle Timed Loop (SCTL) for optimization purposes, you must remove any delay functions since they are not supported.

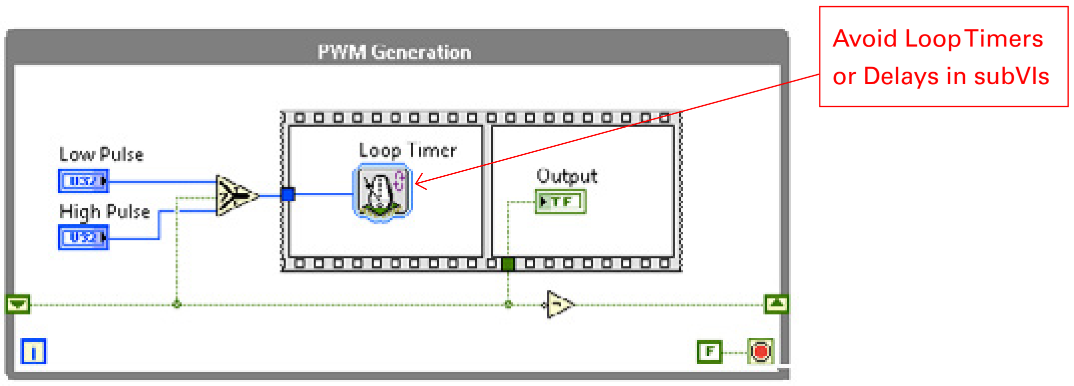


Figure 5.12. Avoid using Loop Timer or Wait functions within your modular subVIs.

The left side of Figure 5.13 shows how you can adapt PWM code to use a Tick Count function rather than a Loop Timer function. Using a feedback node to hold an elapsed time count value, you turn the output on and off at the appropriate times and reset the elapsed time counter at the end of the PWM cycle. The code may look a bit more complicated, but you can drop it inside a top-level loop without affecting the overall timing of the loop—it is more portable.

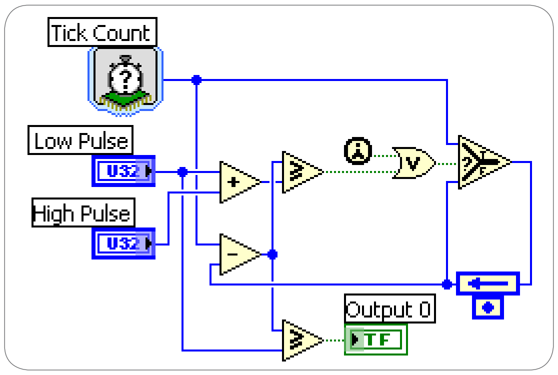


Figure 5.13. Adapt PWM code to use a Tick Count function rather than a Loop Timer function.

Understanding the Trade-Offs Between Reentrant and Nonreentrant SubVIs

Reentrancy is a setting in the subVI Execution properties. In LabVIEW FPGA, subVI execution is set to reentrant by default. Reentrant creates multiple copies of the subVI within the FPGA logic. This enables you to execute multiple copies of the subVI in parallel with distinct and separate data storage.

In the example in Figure 5.14, your subVI is set to reentrant, so both of these loops run simultaneously and any internal shift registers, local variables, or VI-scoped memory data points are unique to each instance.

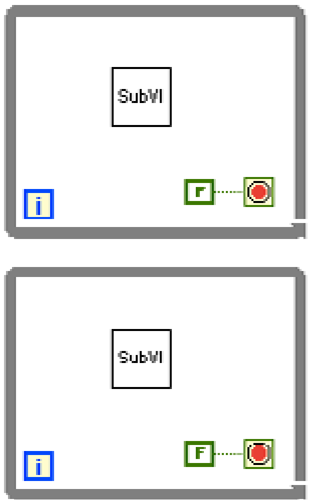


Figure 5.14. A subVI is set to reentrant, so all four of these loops run simultaneously, and any VI-scoped memory data, internal shift registers, or local variables are unique to each instance.

In the case of LabVIEW FPGA, this also means that each copy of the function uses its own FPGA slices, so reentrancy is effective for code portability but has the potential to use more gates. If you are trying to save FPGA resources, you may want to change the execution properties to nonreentrant execution. However, for small subVIs, you could potentially use more FPGA fabric in this case because the compiler adds more logic to the FPGA fabric to handle arbitration. The section below describes techniques for avoiding arbitration.

Avoiding Arbitration

Arbitration occurs when your LabVIEW FPGA code contains a shared resource, such as an I/O node or a nonreentrant subVI, that has multiple readers or writers. In these situations, the compiler adds more logic to the FPGA fabric to handle this arbitration. If you are concerned about minimizing the amount of FPGA fabric you are using, try to avoid arbitration. Find more information on how arbitration works in the LabVIEW FPGA Help Document: Managing Shared Resources.

You can avoid arbitration by changing the arbitration settings within LabVIEW as described in the help document above, but a more reliable technique is designing your application in a way that prevents arbitration. Some of these techniques are listed below:

- Minimize shared resources
- Use double-sided resources in time-critical code (such as FIFOs)
- Avoid multiple readers and/or writers to double-sided resources

Another technique is to create multiple global or local variables when you are in a situation that requires multiple writers. For example, consider having a LabVIEW FPGA VI that uses one loop for reading from an analog input node and two additional loops for processing the analog data. In this situation, you can create two separate global or local variables, say A and B, to write to inside your analog input loop. In your first processing loop, you can read from variable A, and in your second processing loop, you can read from variable B.

Creating Counters and Timers

If you need to trigger an event after a period of time, use the Tick Count function to measure elapsed time as shown in Figure 5.15. Do not use the iteration terminal that is built into While Loops and SCTLs because it eventually saturates at its maximum value. This happens after 2,147,483,647 iterations of the loop. At a 40 MHz clock rate, this takes only 53.687 seconds. Instead, make your own counter using an unsigned integer and a feedback node as well as the Tick Count function to provide time based on the 40 MHz FPGA clock.

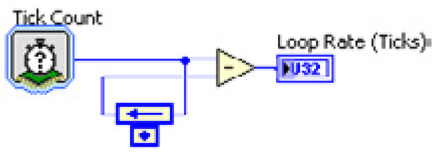


Figure 5.15. Use the Tick Count function to measure elapsed time.

Because you use an unsigned integer for the counter value, your elapsed time calculations remain correct when the counter rolls over. This is because if you subtract one count value from another using unsigned integers, you still get the correct answer even if the counter overflows.

Another common type of counter is an iteration counter that measures the number of times a loop has executed. Unsigned integers are typically preferred for iteration counters because they give the largest range before rolling over. The unsigned 64-bit integer data type you are using for the counter provides a huge counting range—equivalent to about 18 billion-billion. Even with the FPGA clock running at 40 MHz, this counter will not overflow for more than 14,000 years.

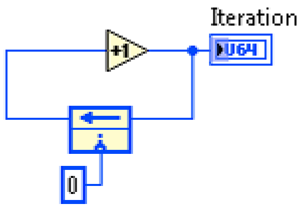


Figure 5.16. Unsigned integers are typically preferred for iteration counters because they give the largest range before rolling over.

Data Communication

Data communication in LabVIEW FPGA falls into two categories: interprocess and intertarget. Interprocess communication generally corresponds to sharing data between two or more loops on the FPGA target. Intertarget data communication is sharing data between the FPGA target and host processor. For both cases, you should consider whether you are communicating current value data, messages, or commands or are streaming data before deciding which mechanism to use. These data communication types are described in more detail in the Introduction and Basic Architectures chapter.

Interprocess Data Communication

If you have multiple loops on your FPGA target, you might want to share data between them like you would in a real-time or Windows-based program. LabVIEW FPGA includes several mechanisms for sharing data between loops.

Variables

LabVIEW FPGA features local and global variables for sharing current values or tags between two or more loops. With variables, you can access or store data in the flip-flops of the FPGA. Variables store only the latest data that is written. They are a good choice if you do not need to use every data value you acquire.

Memory Items

Another method for sharing the latest values is to use available memory items. Taking advantage of memory items consumes few logic blocks but uses the onboard memory. LabVIEW FPGA devices have two types of memory items: target scoped and VI defined. You can access target-scoped memory items through all VIs under the FPGA target. VI-defined memory items are scoped to the VI that defines them. Figure 5.17 shows a block diagram using a Memory Method Node configured for a target-scoped memory item. This VI reads data from memory, increments the data, and then overwrites the same memory location with the new data.

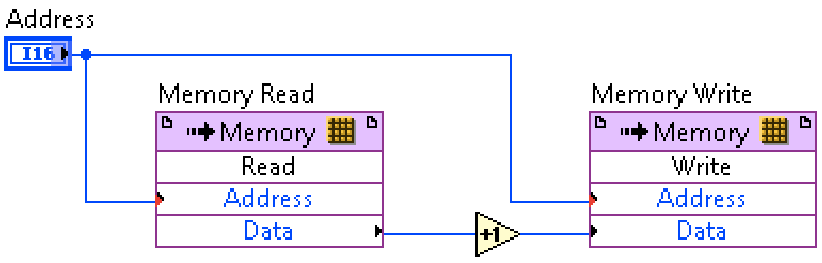


Figure 5.17. Read and write to a memory item using a Memory Method Node.

VI-scoped memory is a powerful tool for applications that need to store arrays of data. In general, you should always avoid using large front panel arrays as a data storage mechanism—use VI-scoped memory instead. For more information on using memory items on an FPGA target, see the LabVIEW Help document Storing Data on an FPGA Target.

FIFOs

If you are communicating messages or updates, or streaming data between two or more loops, consider using FIFOs for data transfer. A FIFO is a data structure that holds elements in the order they are received and provides access to those elements using a first-in-first-out access policy.

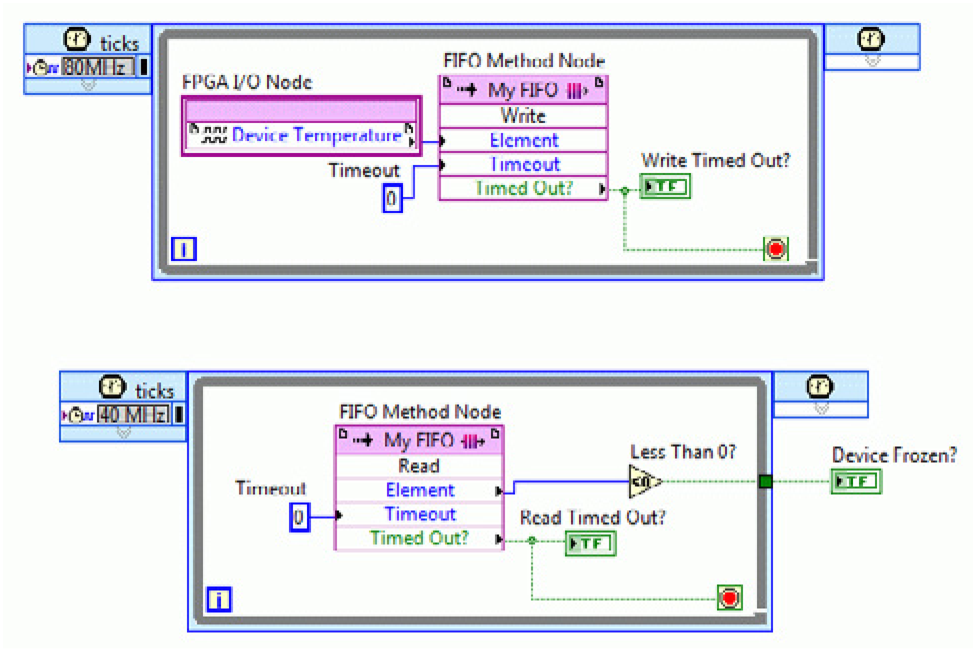


Figure 5.18. Use FIFOs to share buffered data between two parallel loops.

Similar to memory items, you have two types of FIFOs to choose from: target scoped or VI defined. You can access target-scoped FIFOs through all VIs under the FPGA target, and VI-defined FIFOs are scoped to the VI that defines them.

When you configure a FIFO, you must specify the implementation that determines which FPGA resources hold and transfer data in the FIFO. The following recommendations can help you choose one implementation over another.

- **Flip-flops**—Flip-flops use gates on the FPGA to provide the fastest performance. They are recommended only for very small FIFOs (fewer than 100 bytes).
- **Lookup table**—You can store data to two lookup tables per slice on the FPGA. Lookup tables are recommended only for small FIFOs (fewer than 300 bytes).
- **Block memory**—If you are trying to preserve FPGA gates and lookup tables for other parts of your application, you can store data in block memory.

For more information on using FIFOs on an FPGA target, see the LabVIEW Help document Transferring Data Between Devices or Structures Using FIFOs.

Intertarget Communication

You can choose from two methods for communicating data between an FPGA VI and a VI running on the real-time processor: front panel controls and indicators or DMA FIFOs. You can use front panel controls and indicators to transfer the latest values or tags and DMA FIFOs to stream data or send messages and commands. Both methods require using the FPGA Interface functions on the host VI to interact with the FPGA.

Front Panel Controls and Indicators

If you only need to transfer the latest values of data to or from a host VI, you can store the data to controls or indicators on the FPGA VI and access this data using the Read/Write Control function in a host VI. Figure 5.19 shows a simple host VI that is reading from a digital I/O indicator on the FPGA and writing to a digital I/O control on the FPGA.

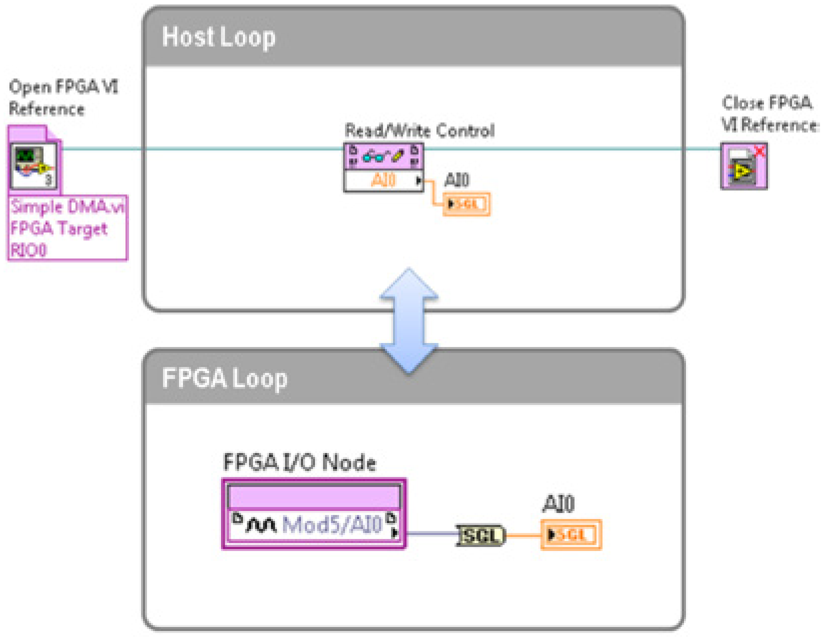


Figure 5.19. Read and write to front panel controls and indicators to share the latest values between targets.

Programmatic front panel communication has low overhead relative to other methods for transferring data between the FPGA and host processor. Read/Write nodes are good candidates for transferring multiple pieces of information between the FPGA and host processor given their relatively low overhead. DMA FIFOs, shown in Figure 5.20, can provide better throughput when streaming large amounts of data, but they are not as efficient for smaller and infrequent data transfers.

Transferring data between the FPGA and host processor using front panel controls and indicators requires involving the host processor more than the DMA FIFOs. As a result, the speed of data transfer is highly dependent on the speed and availability of the host processor. A slower processor or the lack of processor availability results in slower data transfer from the FPGA target to the host. A disadvantage of programmatic front panel communication is that this method transfers only the most current data stored on the control or indicator of the FPGA VI. For example, data could be lost if the FPGA VI writes data to an indicator faster than the host VI can read the data. Also, each control or indicator on the FPGA VI uses resources on the FPGA. Best practices in FPGA programming recommend limiting the number of front panel objects in FPGA VIs.

Additional LabVIEW Help Reference

- Read/Write Control Function

DMA FIFOs

If you need to stream large amounts of data between the FPGA and real-time processor, or if you require a small buffer for command- or message-based communication, consider using DMA FIFOs for data transfer. DMA does not involve the host processor when reading data off the FPGA; therefore, it is the fastest method for transferring large amounts of data between the FPGA target and the host.

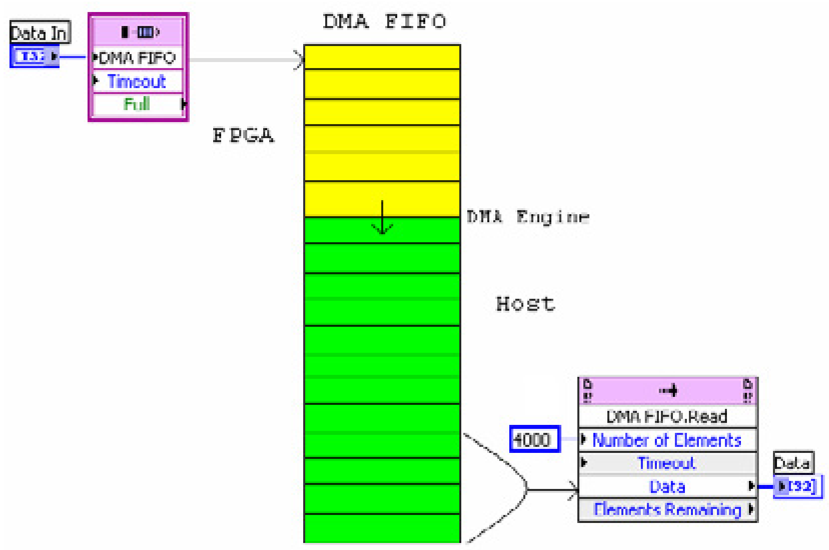


Figure 5.20. DMA uses FPGA memory to store data and then transfer it at a high speed to host processor memory with very little processor involvement.

The following list highlights the benefits of using DMA communication to transfer data between an FPGA target and a host computer:

- Frees the host processor to perform other calculations during data transfer
- Reduces the use of front panel controls and indicators, which helps save FPGA resources, especially when transferring arrays of data
- Automatically synchronizes data transfers between the host and the FPGA target

Though DMA FIFOs are a great mechanism for streaming data, they quickly become complicated when streaming data from more than one channel, optimizing your throughput, or scaling your data on the real-time host VI. If your application requires streaming analog data from one or more analog channels, you should use the NI CompactRIO Waveform Acquisition Reference Library discussed at the end of this section as a starting point. This library features an FPGA template and an NI-DAQmx-like API for streaming data and offers many benefits including optimized performance and built-in scaling.

Using DMA FIFOs in LabVIEW FPGA

To create a DMA buffer for streaming data, right-click on the FPGA target and select **New...»FIFO**. Give the FIFO structure a descriptive name and choose "target to host" as the type. This means that data should flow through this DMA FIFO from the FPGA target to the real-time host. You can also set data type and FPGA FIFO depths. Clicking OK puts this new FIFO into your project, which you can drag and drop to the FPGA block diagram.

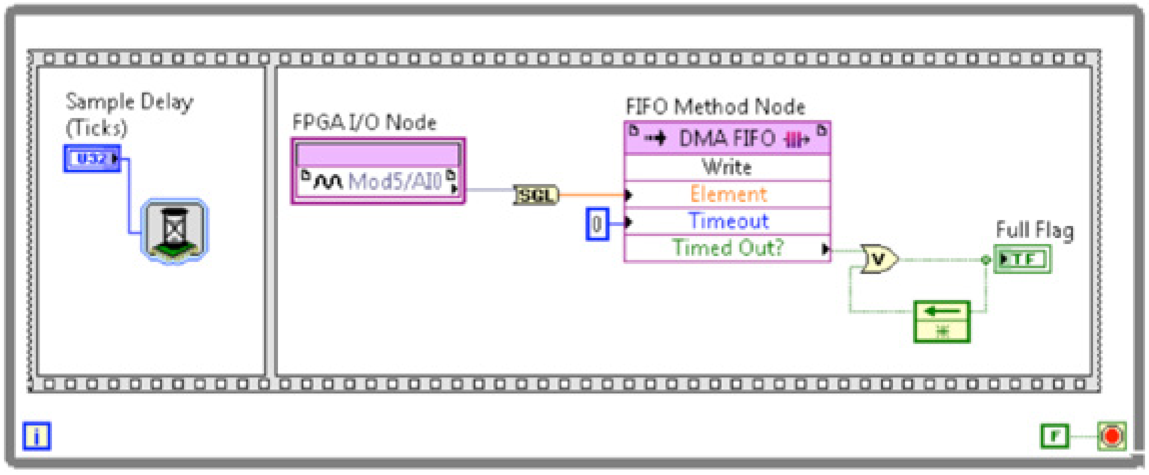


Figure 5.21. Simple DMA Transfer on One Channel (Simple DMA.vi)

Tip: In LabVIEW 2012 FPGA and later, you can convert fixed-point data to single precision floating-point data. Offloading this conversion to the FPGA can save significant processor resources. For earlier LabVIEW versions, you can obtain a similar conversion function through the downloadable example or the NI Developer Zone document Fixed-Point (FXP) to Single (SGL) Conversion on LabVIEW FPGA.

You can use the same DMA channel to transfer multiple streams of data, such as that collected from I/O channels. Most CompactRIO systems have three DMA channels. In Hybrid Mode, CompactRIO systems have only one DMA channel. To pack multiple data streams or I/O channels into one DMA FIFO, use the interleaving technique shown in Figure 5.22, and unpack using decimation on the host.

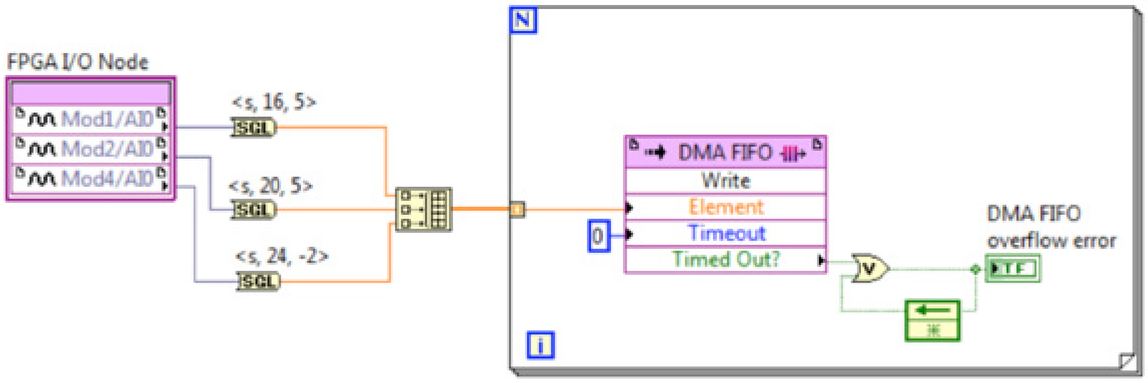


Figure 5.22. You can use build array and indexing on a For Loop to implement an interleaved multichannel data stream.

When passing multiple analog input channels into one DMA FIFO, the channels are stored in an arrangement similar to that shown in Table 5.1. This table assumes that four analog input channels are being interleaved into one DMA

FIFO. The unpacking algorithm on the host VI is expecting the elements to arrive in this specific order. If the FIFO overflows and elements are lost, then the unpacking algorithm on the host VI fails to assign data points to their correct analog input channels. Therefore, it is extremely important to ensure lossless data transfer when reading from multiple analog input channels.

Array Index	Element
0	AI 0
1	AI 1
2	AI 2
3	AI 3
4	AI 0
5	AI 1
6	AI 2

Table 5.1. When writing multiple channels to one DMA FIFO, the host VI expects the elements to arrive in a specific order.

Using DMA FIFOs on the Host

Typically, you dedicate a separate loop on the host VI to retrieve the data from the DMA buffer using the host interface nodes. When reading data from a DMA FIFO on a CompactRIO system, follow these three steps to achieve optimal performance:

1. Set the DMA Read timeout to zero.
2. Read a fixed-size number of elements (a multiple of the number of data channels).
3. Wait until the buffer is full before reading elements.

Figure 5.23 provides a good example of using the DMA FIFO Read functions efficiently to acquire a continuous stream of I/O data. The first DMA FIFO Read function computes the number of elements remaining in the buffer and checks it against the fixed size of 3000 elements. If you are passing an array of data, the Number of Elements input should always be an integer multiple of the array size. For example, if you are passing an array of eight elements (such as values from eight I/O channels), the Number of Elements should be an integer multiple of eight (such as 80, which gives 10 samples for each I/O channel).

As soon as the buffer reaches 3000 elements, the second DMA FIFO Read function reads the data or sleeps in the False case and checks again on the next iteration. Each DMA transaction has overhead, so reading larger blocks of data is typically better.

Finally, use a Decimate 1D Array function to organize the data by channel.

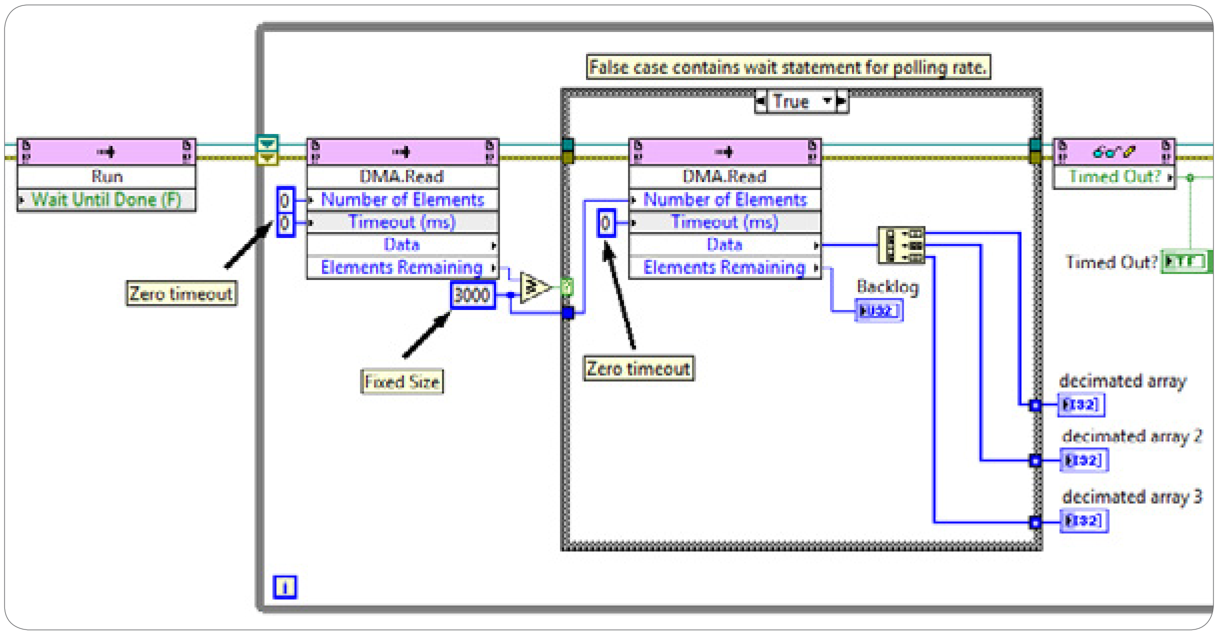


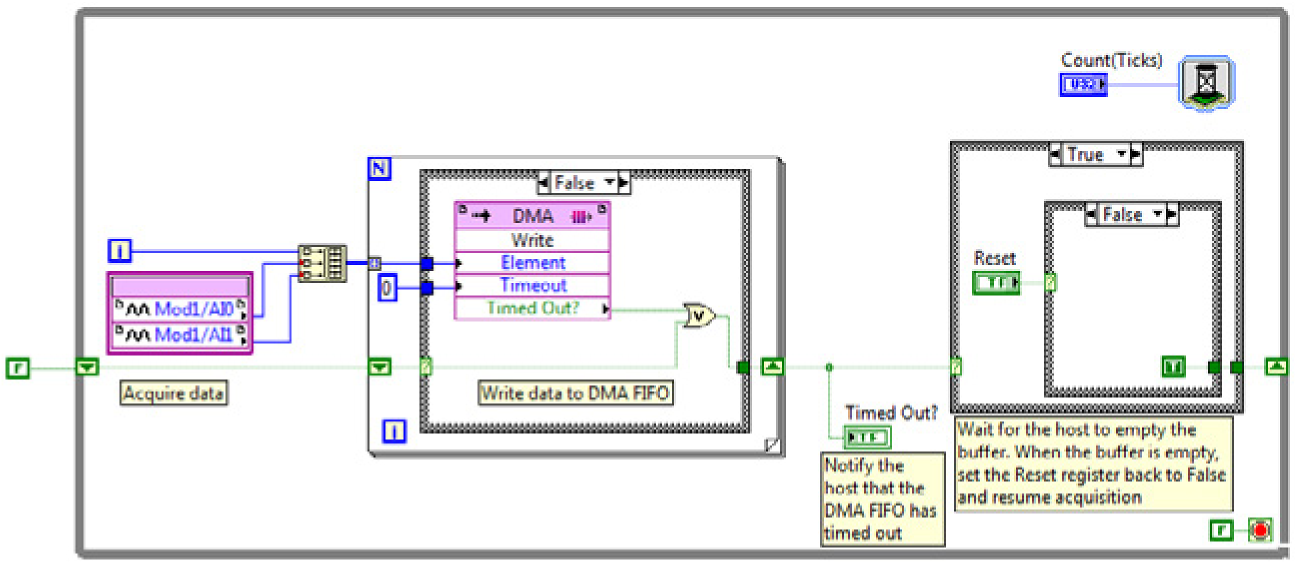
Figure 5.23. Reading Data Off a DMA FIFO (DMA RT.vi)

Whenever you are reading data from a DMA FIFO, you must be able to detect and recover from buffer overflows, or timeout cases, to maintain data correctness. The next section provides a recommended architecture for ensuring lossless data transfer when using DMA FIFOs.

Ensuring Lossless Data Transfer

If lossless data transfer is important, or if you are transferring data from multiple I/O channels into the same DMA FIFO, you must be able to monitor the state of the DMA mechanism and react to any faults that occur. On the FPGA DMA Write Node, a timeout usually indicates that the DMA buffer is full. You should monitor and latch it when it becomes true. You must do this from the FPGA side because simply sampling this register from the host is not sufficient to catch quick transitions. Once you detect a timeout event, you also need to recover from it. Figure 5.24 provides an example of detecting and handling a DMA FIFO timeout case.

FPGA VI


Count(Ticks)
U32
N
False
DMA Write
Element
Timeout
Timed Out?
Mod1/AI0
Mod1/AI1
0
Acquire data
Write data to DMA FIFO
True
False
Reset
Timed Out?
Notify the host that the DMA FIFO has timed out
Wait for the host to empty the buffer. When the buffer is empty, set the Reset register back to False and resume acquisition

RT Host VI

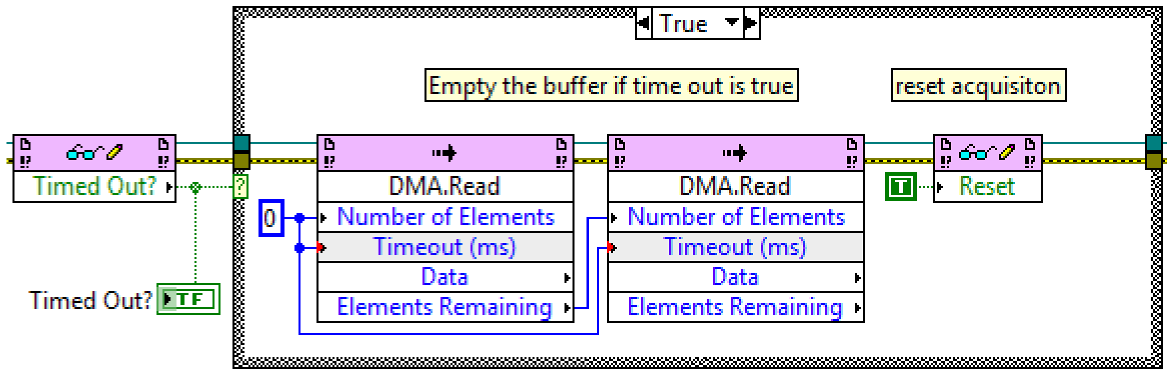


Figure 5.24. Example of Detecting and Recovering From Overflow (DMA Overflow Protection.vi and DMA RT.vi)

In this figure, the timeout event of the DMA FIFO is being monitored on the FPGA VI. When a timeout event occurs, the Timed Out? register is set to True. Once the Timed Out? register is set to True, it is latched until the host VI clears the buffer and sets the Reset register to True. The timeout case on the RT Host VI flushes the buffer by reading the remaining elements. Once the buffer is cleared, the Reset register is set to true and the acquisition on the FPGA VI resumes. By using this method of dealing with the DMA FIFO timeout event, you can detect and recover from buffer overflow so that you can avoid bugs created from missing data points.

Avoiding Buffer Overflows

If you are receiving buffer overflows, you need to either increase the DMA FIFO buffer size on the host, read larger amounts on the host, or read more often on the host. Keep in mind that many control and monitoring applications need only the most up-to-date data, so losing data may not be an issue for a system as long as it returns the most recent data when called.

If you are experiencing buffer overflows, you can increase the host FIFO buffer by using a DMA Configure call as shown in Figure 5.25. For FPGA to RT transfers, an overflow often occurs because the host FIFO buffer, not the FPGA FIFO buffer, is not large enough. The size of the DMA FIFO set through the FIFO properties page determines only the size of the FPGA FIFO buffer that uses up FPGA memory. By default, the host FIFO buffer size is 10,000 elements or twice the size of the FPGA FIFO buffer, whichever is greater.

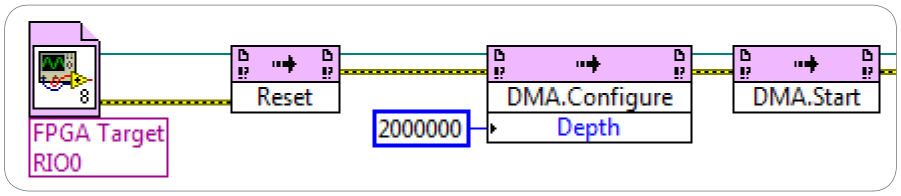


Figure 5.25. Increasing the DMA FIFO size on the host is one way of eliminating buffer overflow issues (DMA RT.vi).

Increasing the size of the buffers can help reduce buffer overflow conditions caused by sporadic events such as contention for buses and the host processor. But if the average transfer rate is higher than what the system can sustain, the data eventually overflows the buffer regardless of the buffer's size. One way to reduce buffer overflow conditions with appropriately sized buffers is to try to read larger amounts of data at a time, which leads to lower overhead per data unit and therefore increases overall throughput.

The CompactRIO Waveform Reference Library

If your application requires the use of DMA FIFOs for streaming analog data from the FPGA, you can jump-start your application development by taking advantage of the CompactRIO Waveform Reference Library developed by NI Systems Engineering. This reference application presents CompactRIO waveform data acquisition VIs and example source code, and supports both delta-sigma and SAR (scanned) modules.

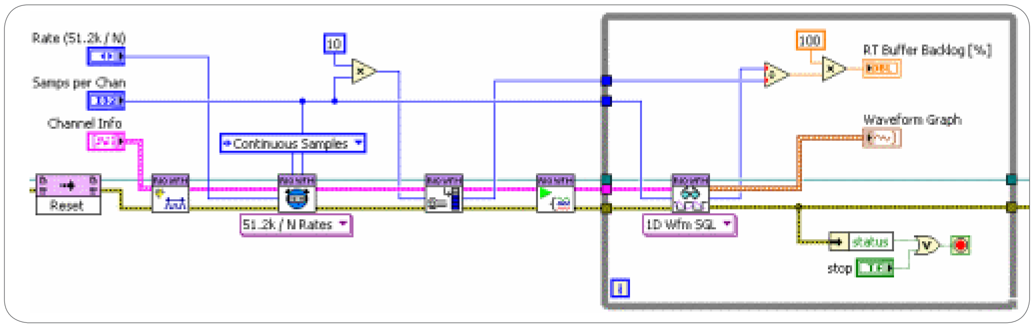
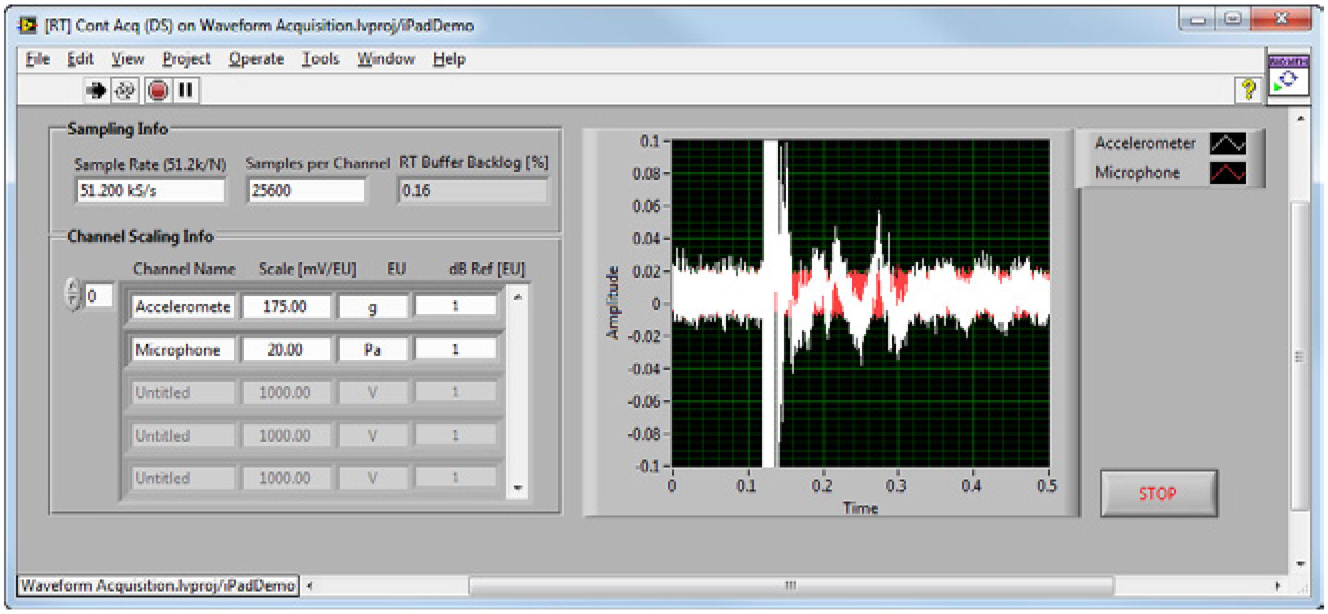


Figure 5.26. Continuous Acquisition Example Using the CompactRIO Waveform Reference Library API

The CompactRIO Waveform Reference Library includes an NI-DAQmx-like API that executes in LabVIEW Real-Time and converts raw analog data into the waveform data type. The waveform data type makes it easy to display multiple channels of I/O to a UI as well as to interface with common data-logging APIs such as Technical Data Management Streaming (TDMS). The API also converts your data to the appropriate engineering units depending on the scale that you configure.


[RT] Cont Acq (DS) on Waveform Acquisition.lvproj/iPadDemo
File Edit View Project Operate Tools Window Help
STOP
Amplitude
Time
Accelerometer
Microphone
Waveform Acquisition.lvproj/iPadDemo

Sampling Info

Sample Rate (51.2k/N)	Samples per Channel	RT Buffer Backlog [%]
51.200 kS/s	25600	0.16

Channel Scaling Info

Channel Name	Scale [mV/EU]	EU	dB Ref [EU]
Acceleromete	175.00	g	1
Microphone	20.00	Pa	1
Untitled	1000.00	V	1
Untitled	1000.00	V	1
Untitled	1000.00	V	1

Figure 5.27. When using the CompactRIO Waveform Reference API, you can quickly stream data and display multiple channels of I/O to a user interface.

This library includes an FPGA VI that you need to slightly modify for the VI to reference the I/O modules used in your system. For most applications, you only need to modify the FPGA VI. This VI, designed to offer optimal streaming performance when using DMA FIFOs, includes mechanisms for monitoring and handling buffer overflow conditions. Use this FPGA VI template as a starting point whenever you are streaming data from one or more analog channels using DMA FIFOs.

You can find more information including installation files in the NI Developer Zone white paper NI CompactRIO Waveform Reference Library.

Delta Sigma Acquisition.lvpj and SAR Acquisition.lvpj are available at **\Program Files\National Instruments\LabVIEW[Version]\user.lib\cRIO Wfm_exampleProjects** after installing the library.

Additional References

- How DMA Transfers Work
- Best Practices for DMA Applications

For more information on using DMA FIFOs on an FPGA target, see the LabVIEW Help document Transferring Data Using Direct Memory Access.

Synchronizing FPGA and Host VIs Through Interrupts

LabVIEW FPGA features interrupts, which synchronize the FPGA and host VIs by allowing the host VI to wait until a specified interrupt is raised by the FPGA. The FPGA VI can block until the interrupt is acknowledged by the host. This eliminates the need to continually check the status of an FPGA register to see if a given flag has been set, and it lowers CPU use while the interrupt is not raised. On single-core hosts, you can achieve relatively good latency and decreased CPU load when compared to the poll-and-sleep approach. The example in Figure 5.28 shows how you might implement an interrupt on the FPGA. This example is called the Interrupt Method for Synchronization and can be found in the NI Example Finder.

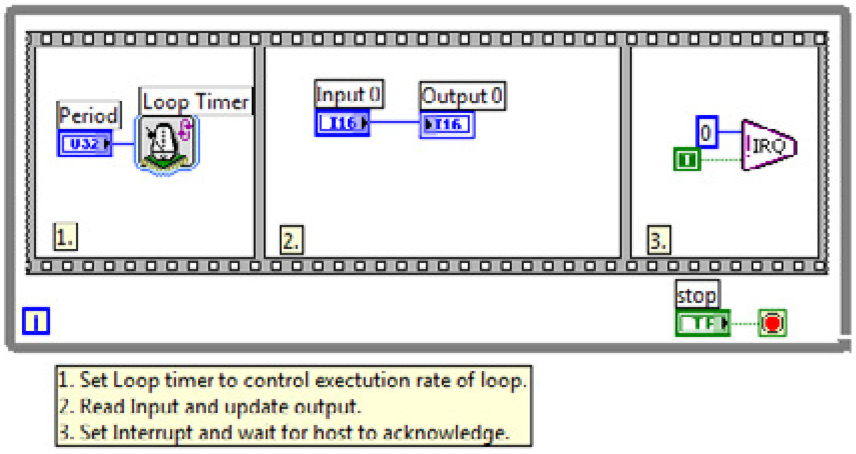

Period
U32
Loop Timer
Input 0
I16
Output 0
I16
0
IRQ
1.
2.
3.
stop
TF
1. Set Loop timer to control exectution rate of loop.
2. Read Input and update output.
3. Set Interrupt and wait for host to acknowledge.

Figure 5.28. Basic LabVIEW FPGA Programming Structure for Sending an Interrupt

Interrupts offer simple code with a straightforward API, but you should be aware of the following caveats:

- Handling the interrupt on the host side involves overhead usually on the order of 5 µs to 50 µs. This response latency may be acceptable depending on the application or the frequency of the event.
- If you do not want the FPGA VI to block until the interrupt is acknowledged, you should add a separate FPGA loop for sending the interrupt.
- Interrupts are not the lowest latency option on multicore targets. The fastest response time is obtained by dedicating a CPU core to continuously monitor an FPGA register (no sleep).

Additional References

- Using Interrupts on the FPGA to Synchronize the FPGA and Host
- Synchronizing FPGA VIs and Host VIs Using Interrupts

Test and Debug LabVIEW FPGA Code

As described in the Best Practices for Implementing LabVIEW FPGA Code section, you should develop your LabVIEW FPGA VIs in simulation mode to quickly iterate on your designs and avoid long compile times. When you need to test and debug your VIs, you can stay in simulation mode or take advantage of several other options. You should select an execution mode depending on your requirements for functional verification versus performance and on the type of code you are testing: unit, component, or system. Each type of code has different attributes and verification requirements as described below.

Unit

"Unit" is the most fundamental level of IP you can build in the sense that it maps to a specific processing function or algorithm. You would not split it and test it as a set of smaller functional units.

You want to keep these simple, so you would call it a unit if:

- It can be encapsulated as a subVI you may want to reuse in other parts of your design
- It does not include I/O, data communication, or any target resources
- It does not have multiple loops running in parallel or at different rates
- It is functional in nature—you can provide some known inputs and test for expected outputs
- It may hold state, in which case you may need to call it multiple times to verify it, but its behavior should not rely on the explicit specific passing or control of time

Component

Components are more complex pieces of logic that include elements that can exhibit side effects or focus more on the timing in the system. They are composable by definition, and usually have a clear task or objective to accomplish. An FPGA application can often be broken down into multiple components, and verification at this level ensures that the components interact as expected when integrated into a larger component. You may also want to make sure that a subcomponent is interacting properly with the I/O or host (through the host interface) without waiting until the whole system is assembled.

System

You can think of the system level as the top-most component. It is represented by your top-level FPGA VI plus any additional HDL IP imported through the Component-Level IP (CLIP) Node. It is somewhat different from other components in that its interface is exposed to the host application, so verification tests are either similar to running your host app or they are your host app. Verification therefore requires using the host interface API as well as connecting real I/O signals to your system. A system usually contains multiple While Loops or SCTLs.

Table 5.2 provides guidance on which execution mode you should use for verification and debugging. Keep in mind that if you can perform extensive debugging and verification at the unit and component levels, then you decrease your verification work at the system level.

Execution Mode	Verify Functional Performance	Verify Timing	Verify Integration of HDL IP	Good for Unit Testing	Good for Component Testing	Good for System Testing
Windows PC	X			X		
FPGA Simulation Mode	X	X (only code contained in SCTL)			X	
FPGA Hardware	X	X	X		X	X
Advanced: Third-Party Simulator	X	X	X		X	

Table 5.2. This table provides a comparison of different execution modes for verifying and debugging your LabVIEW FPGA code.

Execute LabVIEW FPGA Code on Your Windows PC

You can execute your FPGA VI on your Windows PC by dragging the FPGA VI to the “My Computer” target in the LabVIEW project. This is the quickest and easiest approach for debugging and/or testing unit-level code. All functions contained in the LabVIEW FPGA palette, excluding target resources, can be executed in the desktop context.

There are several benefits to this approach. For debugging purposes, you have access to standard LabVIEW debugging features and visualization options (graphs, charts, and so on). For testing purposes, you have access to the hundreds of libraries in the LabVIEW for the Desktop programming palette.

Do **not** make changes to your LabVIEW FPGA code while in this context since you are no longer confined to the limited LabVIEW FPGA palette and you might introduce constructs that are supported only on the desktop. The My Computer execution context is ideal for developing tests that involve writing code around your LabVIEW FPGA VI.

Execute in Simulation Mode

Another option is to use LabVIEW FPGA simulation. Simulation executes your FPGA code on the host computer using a built-in high-fidelity, bit-accurate simulator. The simulator is cycle-accurate when simulating code contained within an SCTL, assuming that the design can compile at the desired rate. The simulator supports FPGA target resources such as I/O, memory items, and DMA FIFOs, so it can be used for code at the unit and component levels. You can configure your LabVIEW FPGA VI to run in the simulation mode by right-clicking FPGA Target in the LabVIEW project and selecting Execute VI on»Development Computer with Simulated I/O.

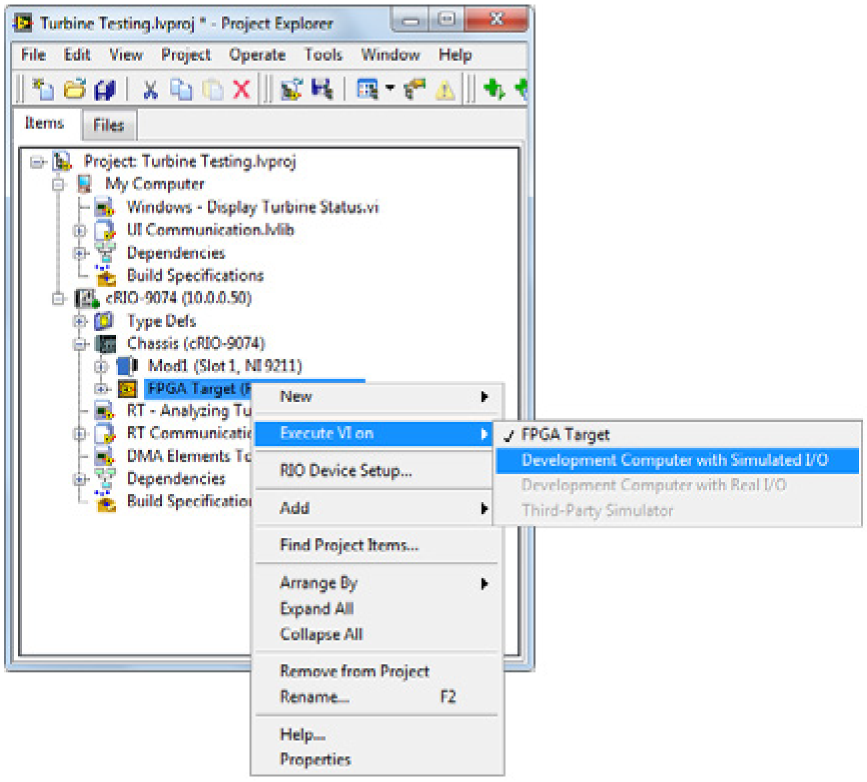


Figure 5.29. You can change the execution mode of your LabVIEW FPGA VI by right-clicking the FPGA target in the LabVIEW project.

Debugging in Simulation Mode

When executing your LabVIEW FPGA VI in simulation mode, you have access to standard LabVIEW debugging features including highlight execution, probes, and breakpoints. LabVIEW 2013 and later includes an additional debugging tool called the Sampling Probe. When inserted into your FPGA design while running in simulation, these probes provide a unique probe window of the different digital signals in relationship to one another with respect to time (or tick/clock iteration count in this case).

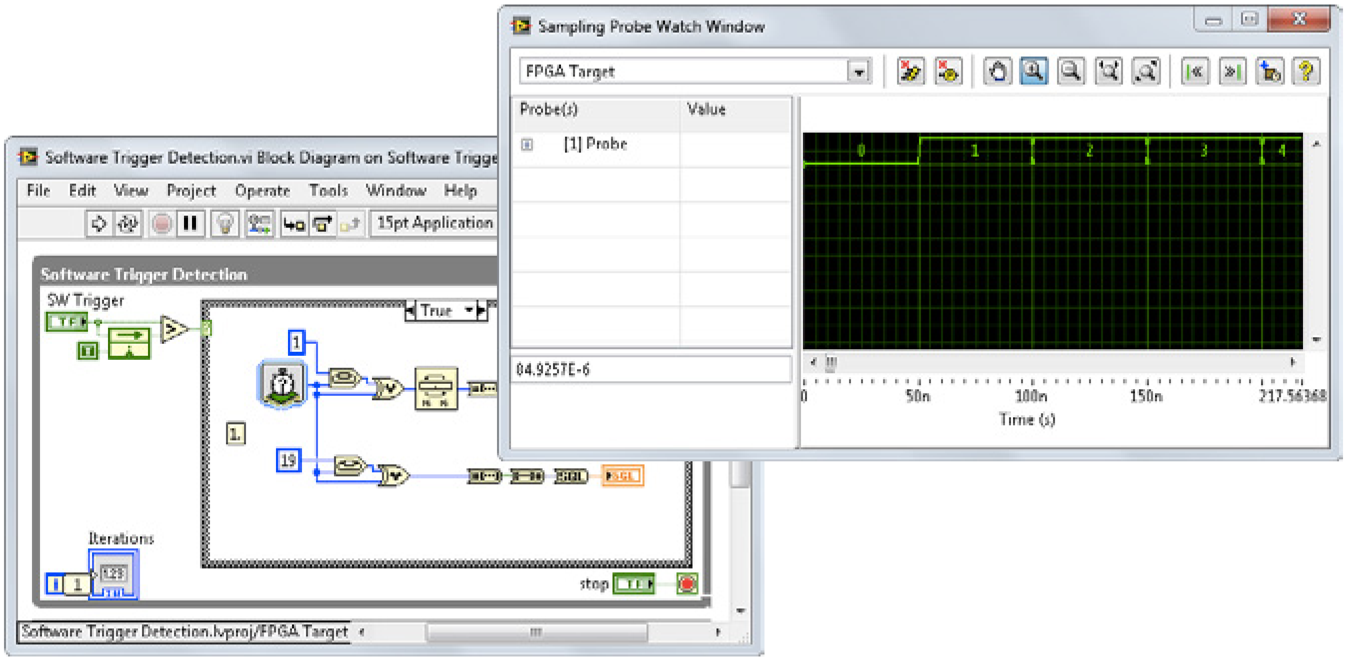

Sampling Probe Watch Window
FPGA Target
Probe(s)
Value
[1] Probe
04.9257E-6
0
50n
100n
150n
217.56368
Time (s)
Software Trigger Detection.vi Block Diagram on Software Trigge
File Edit View Project Operate Tools Window Help
15pt Application
Software Trigger Detection
SW Trigger
True
Iterations
stop
Software Trigger Detection.lvproj/FPGA Target

Figure 5.30. View signals in relation to each other with respect to time using the new LabVIEW FPGA sampling probe.

The window also includes functions for locating the next rising edge or previous rising edge on a signal, and zoom capabilities to navigate through the data. For information on how to use the Sampling Probe, see the LabVIEW FPGA Help Document: Debugging Using Sampling Probes.

Testing Your LabVIEW FPGA Code in Simulation

You can create tests in LabVIEW that exercise your LabVIEW FPGA components. The recommended approach is to have a VI that performs testing (or testbench VI) running in the host (My Computer or Real-Time) context, and the LabVIEW FPGA VI running in the FPGA context in simulation mode or in actual hardware. Since the testbench VI runs in the host context, you have access to hundreds of functions that you can use to create test vectors, cases, and stimuli in addition to the checking, analysis, display, and reporting of test results.

You can also use the same library of functions to build a reference implementation of your component. You can import implementations built in C/C++ or any other language that can generate a DLL or shared library as well as take advantage of support for other languages such as .m script or The MathWorks, Inc. Simulink® software. Figure 5.31 provides a diagram of a basic test, all of which can be implemented within the LabVIEW graphical development environment.

Simulink® is a registered trademark of The MathWorks, Inc.

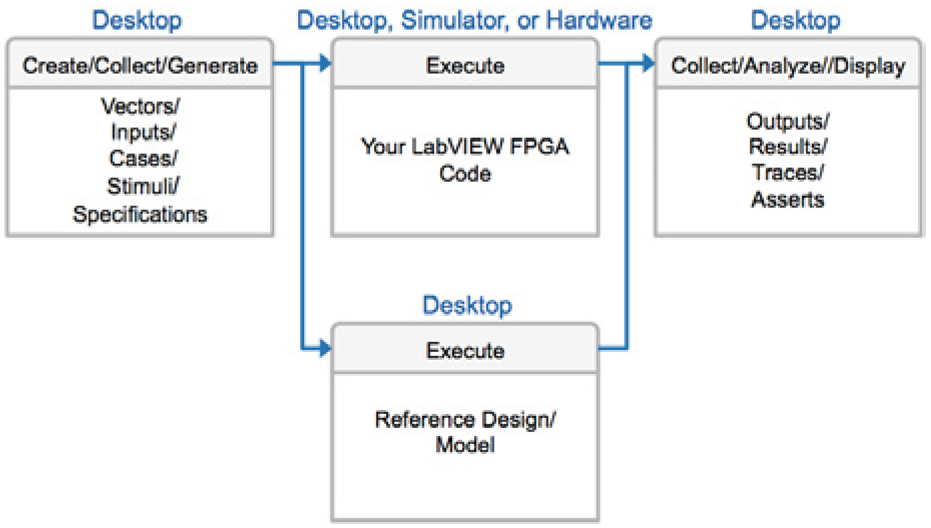


Figure 5.31. A typical test bench consists of several components, all of which can be implemented in the LabVIEW graphical development environment.

In the LabVIEW FPGA Module Version 2013 and later, you can test LabVIEW FPGA components using the Desktop Execution Node. With the Desktop Execution Node, you can perform verification on a LabVIEW FPGA VI without making changes to your LabVIEW FPGA code to accommodate the testing. The Desktop Execution Node uses simulated time to reflect timing in hardware. You can provide stimuli to controls, indicators, and I/O before time advances; force time to advance a set number of ticks; and then read the values of the controls, indicators, and I/O before providing additional stimuli to the VI.

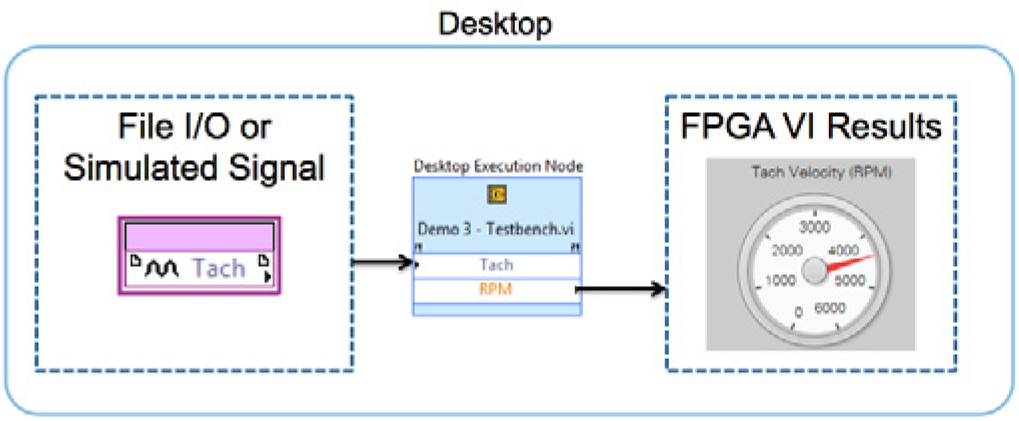


Figure 5.32. The Desktop Execution Node can be used to create a test bench for your LabVIEW FPGA VI

Understanding Simulated Time

To be successful with the Desktop Execution Node, you need to understand the concept of simulated time. Starting with LabVIEW 2013 FPGA, a developer may consider two timing paradigms when creating an FPGA design: wall clock time and simulated time. Wall clock time is the actual, real-world time a slice of logic may take to execute. An FPGA device that has been programmed with a bitfile runs in wall clock time. Simulated time is an event-driven model of wall clock time. During execution, some nodes may announce that a time step is required at a particular time. When LabVIEW detects that execution is idle, simulated time advances to the next earliest time step.

The following nodes tie into simulated time and announce time steps or return a simulated time value:

- While Loop
- Single-Cycle Timed Loop
- Wait Express VI
- Loop Timer Express VI
- Tick Count Express VI
- FIFOs except DMA FIFOs
- Wait on Occurrence and Wait on Occurrence with Timeout in Ticks
- Interrupt VI when Wait Until Cleared is TRUE

For specifics on individual node timing or execution behavior, read the NI white paper Using the LabVIEW FPGA Desktop Execution Node.

Using the Desktop Execution Node

The Desktop Execution Node is generally recommended for validating components. Since it executes the FPGA VI in simulation mode, you can develop tests for VIs that contain target resources such as I/O and memory items. This section describes the steps to set up the Desktop Execution Node for component testing.

Consider an example featuring a LabVIEW FPGA component that has been designed to read tachometer and accelerometer data from an analog input node (NI 9234) and then convert the tachometer reading into revolutions per minute (rpm) before sending it to the host. Since the NI 9234 uses delta-sigma modulation, you are specifying the sample rate of the module using a property node. The goal of your test is to verify the tachometer IP (Tach_ FPGA.vi) that you downloaded from ni.com/ipnet.

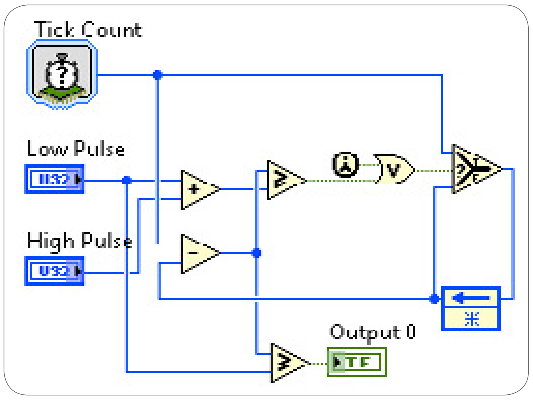


Figure 5.33. This section references an example FPGA VI that acquires tachometer data and converts it to rpm.

Step 1: Create a Test VI in the Windows Context

Right-click **My Computer** in the LabVIEW project to create a new VI. Select the FPGA Desktop Execution Node in the FPGA Interface palette.

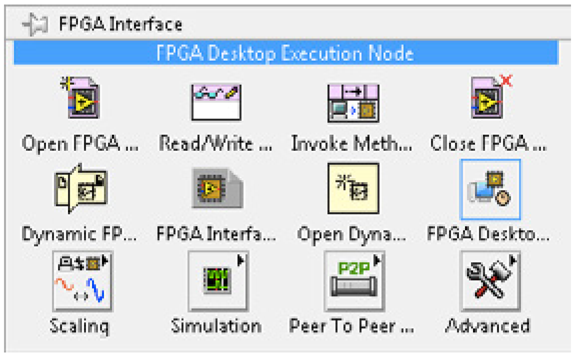


Figure 5.34. The Desktop Execution Node is on the FPGA Interface palette.

Step 2: Configure the Desktop Execution Node

Select Your VI

Each component you want to verify using the Desktop Execution Node must be saved as a VI prior to testing. You can then point to your FPGA VI from within the Desktop Execution Node configuration window. Once you select your FPGA VI, all of the controls, indicators, and FPGA resources that you have access to populate under Available Resources.

Select Terminals

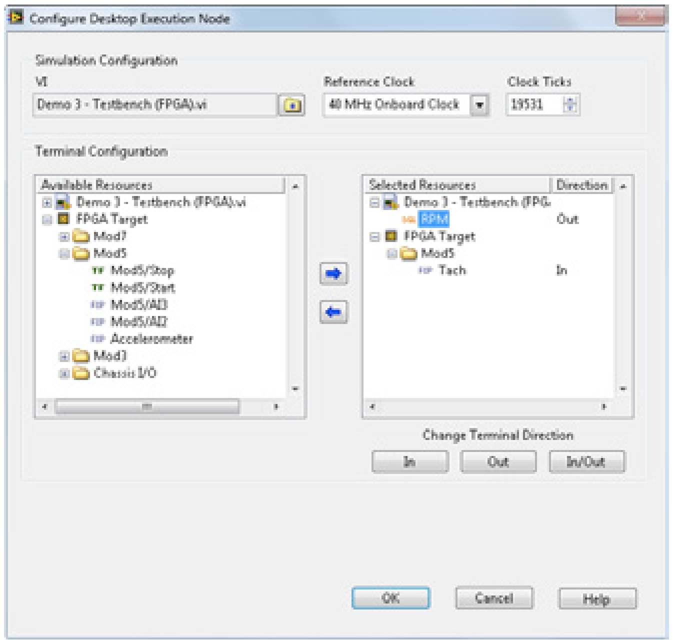
You can configure the Desktop Execution Node terminals by selecting your resources of interest and using the blue arrows to copy them over into the Selected Resources window. For this example, you want to write to the analog input node named **Tach** and read from the indicator named **RPM**.

Select a Reference Clock

To configure the Reference Clock, select the clock that the loop within your FPGA VI is referencing. The default clock on most RIO hardware targets is 40 MHz. If you are using an SCTL, the reference clock could be the top-level clock or a derived clock.

Notes

- Most components contain a single loop. If your component contains multiple loops, select the fastest clock referenced in your VI.
- If you are performing unit testing and your code is not contained within a loop, encompass your unit code within a While Loop for the purpose of testing. Otherwise, the Desktop Execution Node stops after completing the first iteration.


Configure Desktop Execution Node

Simulation Configuration

VI: Demo 3 - Testbench (FPGA).vi

Reference Clock: 40 MHz Onboard Clock

Clock Ticks: 19531

Terminal Configuration

Selected Resources	Direction
Demo 3 - Testbench (FPG...	
RPM	Out
FPGA Target	
Mod5	
Tach	In

Figure 5.35. The first step to configuring the Desktop Execution Node is to select your FPGA VI.

Determine the "Clock Ticks" or Simulated Time

The Desktop Execution Node can control simulated time so developers can alter stimuli at key points during execution on the development computer with simulated I/O. To successfully use this feature, you need to measure the time your FPGA VI takes to complete or you need to design your VI in such a way that the time it takes to complete is intuitively known (for example, using a Loop Timer to guarantee timing). Some tips on measuring the time that your FPGA VI takes to complete are below.

Single-Cycle Timed Loop

If you are using an SCTL, the code contained within that loop always takes one tick of the reference clock to execute, so you can set the Clock Ticks to 1 tick. If you have multiple SCTLs, select the fastest clock as the reference clock.

While Loop With a Loop Timer

In this case, you can specify the value of the Loop Timer in ticks. If your loop timer is configured in milliseconds or microseconds, perform the conversion. For example, if you have a Loop Timer set to 10 µs, perform the following calculation:

- Ticks = clocks (Hz) x Time (s)
- Ticks = 40,000,000 Hz x 0.00001 seconds
- Ticks = 400

Therefore, you would configure your Clock Ticks input in the Desktop Execution Node to 400 ticks.

While Loop With No Loop Timer

If you are using a While Loop that does not contain any analog I/O (it may contain digital I/O), the easiest way to measure the number of ticks is by using the Sampling Probe. To do this, create an indicator off the While Loop iteration terminal. Right-click the wire and select **Sampling Probe»FPGA**. Run the VI in simulation mode for a second or two, stop, and view the Sampling Probe window. In the example below, you can see that the While Loop takes two ticks of the 40 MHz clock to execute, according to the Sampling Probe window (each tick equals 25 ns). For this case, you would set the Clock Ticks input to two ticks.

Note: Because this is a While Loop, the number of ticks measured in simulation does **not** necessarily equal the number of ticks that result when executing the same code in hardware. Any code inside an SCTL, however, is guaranteed to be cycle accurate.

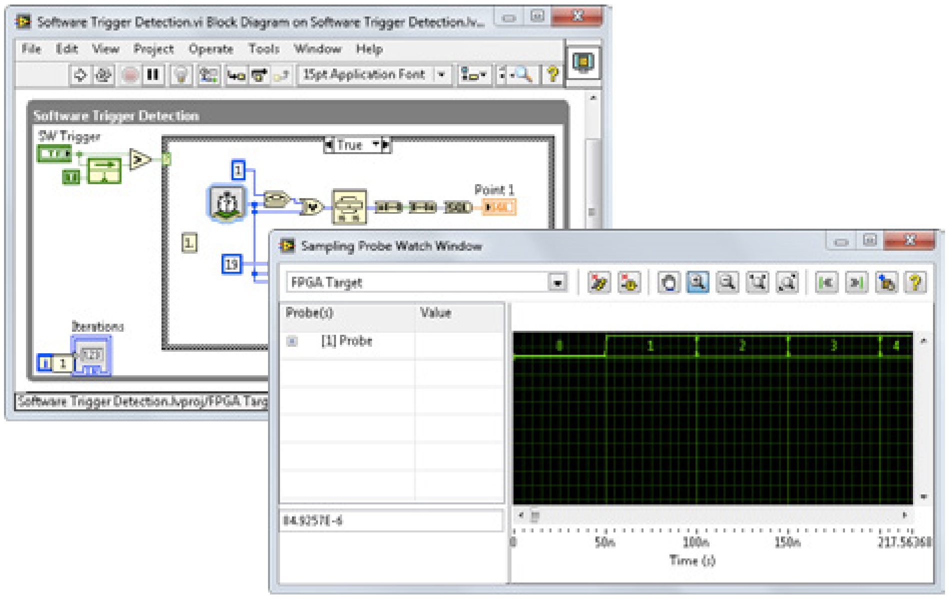

Software Trigger Detection.vi Block Diagram on Software Trigger Detection.lv...
File Edit View Project Operate Tools Window Help
15pt Application Font
Software Trigger Detection
SW Trigger
True
Point 1
Iterations
Software Trigger Detection.lvproj/FPGA Targ
Sampling Probe Watch Window
FPGA Target
Probe(s)
Value
[1] Probe
84.9257E-6
0
50n
100n
150n
217.56368
Time (s)

Figure 5.36. If your While Loop does not contain a timing source, you can measure the number of clock ticks per iteration by using a Sampling Probe.

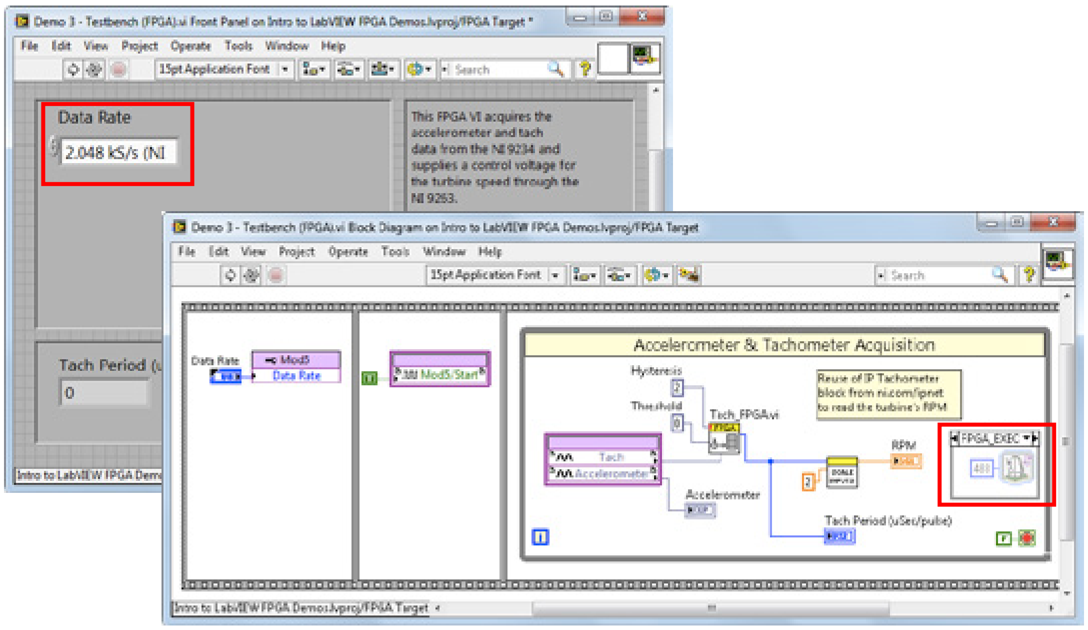
While Loop Executing at the Rate of a Delta-Sigma Module Clock

You may have an FPGA VI that is executing at a rate defined by a delta-sigma analog input module, such as the VI used in the example shown in Figure 5.37. In simulation mode, LabVIEW ignores the timing input from these property nodes. Therefore, these situations require you to perform two steps:

1. Control the simulation timing of the VI by adding a Loop Timer that is equivalent to the scan rate

2. Set the Clock Ticks input in the Desktop Execution Node to the same value that is input to the Loop Timer

In this specific example where you are reading from a tachometer, the analog input module is configured to run at a rate of 2.048 kS/s, or 488 µs. For the first step, you need to add a Loop Timer to the While Loop and configure it to a loop rate of 488 µs.


Demo 3 - Testbench (FPGA).vi Front Panel on Intro to LabVIEW FPGA Demos.lvproj/FPGA Target *
Demo 3 - Testbench (FPGA).vi Block Diagram on Intro to LabVIEW FPGA Demos.lvproj/FPGA Target
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
Accelerometer & Tachometer Acquisition
Hysteresis
Threshold
Tach_FPGA.vi
Reuse of IP Tachometer block from ni.com/ipnet to read the turbine's RPM
RPM
Tach
Accelerometer
Tach Period (uSec/pulse)
FPGA_EXEC
488
Mod5 Data Rate
Mod5 Start
Intro to LabVIEW FPGA Demos.lvproj/FPGA Target

Data Rate: 2.048 kS/s (NI

Tach Period: 0

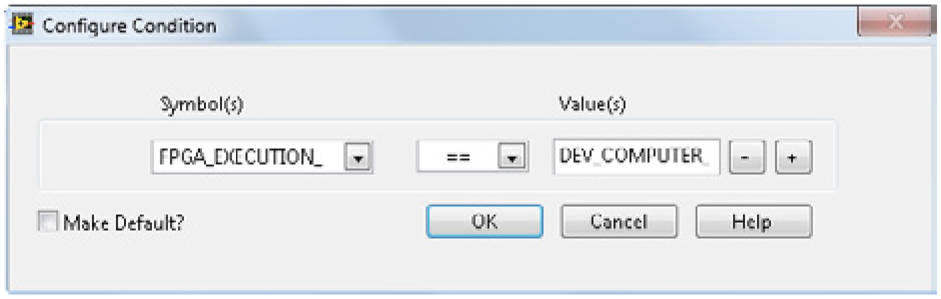
This FPGA VI acquires the accelerometer and tach data from the NI 9234 and supplies a control voltage for the turbine speed through the NI 9263.

Figure 5.37. When the timing source of your loop is the onboard clock of a delta-sigma analog input module, you need to specify the simulated time by adding a conditional Loop Timer.

You can put the Loop Timer inside a Conditional Disable structure so that it is called only when the VI is executing in simulation mode. Configure the Conditional Disable structure with the following values:

- FPGA_EXECUTION_MODE == DEV_COMPUTER_SIM_IO

The Loop Timer should be placed inside this conditional case, and the default case should be empty.



Symbol(s)		Value(s)
FPGA_EXECUTION_	==	DEV_COMPUTER_

☐ Make Default?

Figure 5.38. Configure a Conditional Disable structure to execute the Loop Timer only when in simulation mode.

Finally, the last step is to set the Clock Ticks input in the Desktop Configuration Node. In this example, 488 µs is equal to 19,531 ticks, so you set the Clock Ticks input to 19,531 ticks.

To learn more about using the Desktop Execution Node, see the NI white paper Using the LabVIEW FPGA Desktop Execution Node.

Step 3: Build Your Test Bench

Once you have successfully configured the Desktop Execution Node, you can begin building your test bench. In the example shown below, logged tachometer data is being read from a TDMS file and written to the analog input I/O node called Tach. The FPGA VI is executed in simulation mode, and the resulting rpm is displayed on a chart. To verify the tachometer IP, you can view the tachometer data read from the file and the rpm data plotted to the chart.

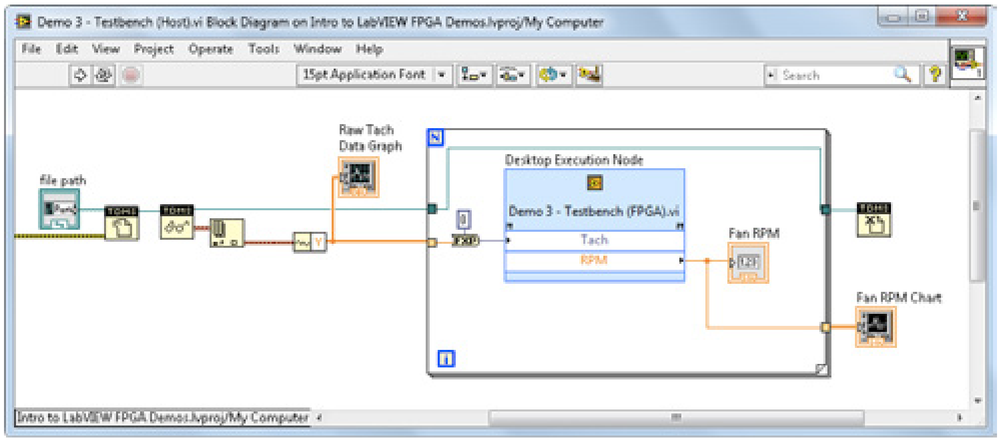

Demo 3 - Testbench (Host).vi Block Diagram on Intro to LabVIEW FPGA Demos.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
file path
Raw Tach Data Graph
Desktop Execution Node
Demo 3 - Testbench (FPGA).vi
Tach
RPM
Fan RPM
Fan RPM Chart
Intro to LabVIEW FPGA Demos.lvproj/My Computer

Figure 5.39. Once the Desktop Execution Node is configured, you can create a test VI and verify your FPGA code.

Execute Within a Third-Party Simulator

If you are required to verify timing in addition to functionality, before you compile your LabVIEW FPGA VI to hardware, you can interface with three third-party simulators: Mentor Graphics ModelSim (LabVIEW 2013 and earlier), Mentor Graphics Questa, and Xilinx ISim. You can use the design verification and debugging capabilities of these simulators to debug functional behavior and timing-based errors. To use ISim with the LabVIEW FPGA Module, you should be familiar with HDL simulators and VHDL, which is required for writing a test bench. You now have the option to write test benches for ModelSim using either VHDL or LabVIEW. The white paper Cycle-Accurate Co Simulation with Mentor Graphics ModelSim provides a tutorial for interfacing to ModelSim.

Execute in Hardware

Debugging in simulation mode or in the desktop context can save you time, but situations do arise that require the code to be compiled to the FPGA target and then run and debugged in real time. In these situations, you have to write additional code into the application to test and validate the application's core functionality. This code is usually removed or disabled when debugging is complete. The NI FPGA Debug Library component offers a set of simple functions to debug LabVIEW FPGA applications in real time. In addition to performing common simple tasks, these functions provide a modular programming interface to help you rapidly construct advanced debugging structures.

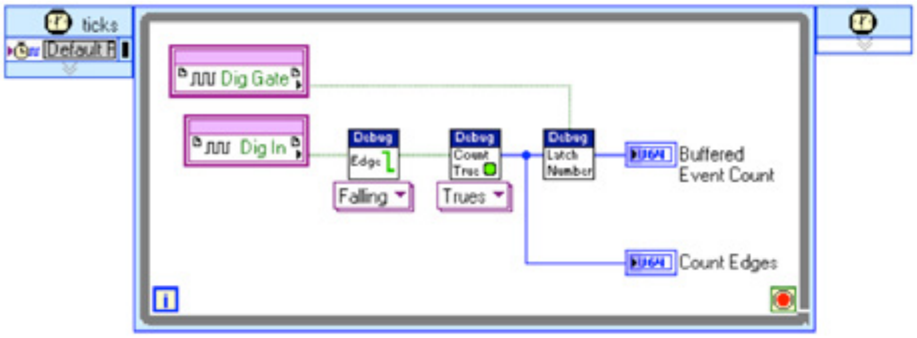


Figure 5.40. The NI FPGA Debug Library contains VIs that you can use for common simple tasks, or advanced debugging structures like the advanced I/O counters shown here.

Optimizing LabVIEW FPGA Code

When you use regular LabVIEW programming techniques in LabVIEW FPGA, you immediately reap the benefits of the FPGA-based approach. Sometimes you may need to push your system even further in one or more of these related dimensions: throughput, timing, resources, and numerical precision.

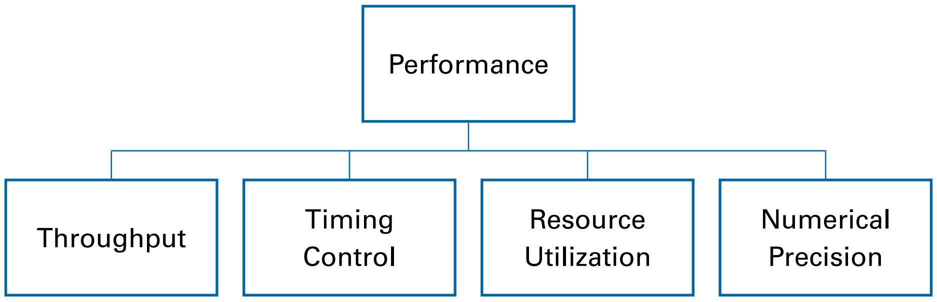


Figure 5.41. With LabVIEW FPGA programming techniques, you can push your system further for throughput, timing control, resource utilization, or numerical precision.

These dimensions are often interconnected, so improving your design with one of them affects the others sometimes positively but more often at their expense. It is important to understand these dimensions and how they relate to each other, so this guide lists some basic definitions below and explores related techniques in later chapters.

Throughput

Throughput is a key concern for DSP and data processing applications. It is measured as work per unit of time. In the majority of applications using NI RIO hardware, work refers to the processing or transfer of samples, so their throughput is usually measured in samples per second or some equivalent form such as bytes, pixels, images, frames, or operations per second. The fast Fourier transform (FFT) is an example of a processing function with throughput that can be measured in FFTs, frames, or sample per second.

The High-Performance RIO Developer’s Guide offers a more in-depth discussion of the factors that affect throughput as well as a set of techniques that can help you achieve higher throughput when creating LabVIEW FPGA applications.

Timing Control

Timing control refers to the ability to prescribe and measure the amount of time between events of interest in the system. When you use LabVIEW FPGA, your designs are translated to a hardware circuit so you can create designs that have fast timing responses with little jitter. Control applications usually require a guaranteed maximum response time between system sampling and control signal updating. This amount of time is referred to as latency. In digital protocol applications, timing specifications may refer to the target, minimum, or maximum time between events related to the data or signals being transmitted. Precise timing control is important to both the control and digital protocol application areas.

The High-Performance RIO Developer’s Guide for a more in-depth discussion of latency as well as a set of techniques that can help you achieve lower or more precise timing responses when using LabVIEW FPGA.

Resource Utilization

An FPGA has a finite number of resources and is usually much more constrained in storage and memory elements than a processor or microcontroller. Being able to fit your design into the FPGA is a hard constraint on the whole development process. FPGAs are also made up of different types of resources, so running out of one type of resource can keep you from progressing in your applications.

More importantly, resource utilization can have a dramatic impact on the other performance dimensions, especially throughput and meeting timing constraints. Refer to the High-Performance RIO Developer's Guide for descriptions of the different resources that compose the FPGA and how you can balance them to make your design fit and increase its performance.

Numerical Precision

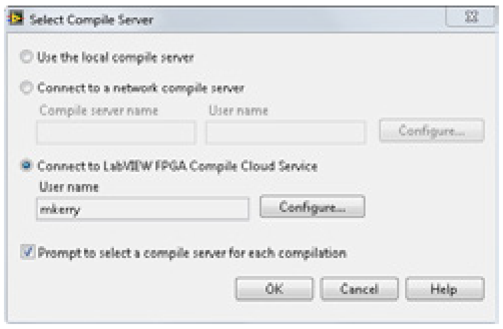
Numerical precision concerns revolve around having enough digits or bits so your application can work correctly. Inadequate numerical precision is considered a functional problem that must be avoided and tested against. The number of bits used to represent system variables, including the number of bits used for integers, the integer and fractional parts of fixed-point numbers, and the dynamic range of floating-point numbers, can have a substantial impact on the performance and resource utilization of your application, so you should consider this.

Compiling LabVIEW FPGA Code

The LabVIEW FPGA compilation process is described in the first section of this chapter, "FPGA Technology." This process can take up to several hours depending on how intricate your design is. This section provides some tips for reducing compile times and understanding the compile report.

Reduce Compile Times

NI provides several options for compiling your LabVIEW FPGA code. When you create your LabVIEW FPGA build specification in the LabVIEW project, you see the options in a dialog similar to the one shown in Figure 5.42.


Select Compile Server

( ) Use the local compile server

( ) Connect to a network compile server

Compile server name:
User name:

(●) Connect to LabVIEW FPGA Compile Cloud Service

User name: mkerry

☑ Prompt to select a compile server for each compilation

Figure 5.42. You have several options for selecting a compile server when compiling your LabVIEW FPGA VI.

Use the local compile server—By default you can compile on your local development PC. This is a viable solution for compiling small VIs, but if you are concerned about long compile times, you should consider the other options below.

Connect to a network compile server (single server)—You can choose to offload your compiles to a single Windows- or Linux-based machine on the network. Compilation times are proven to be reduced when compiling on a Linux machine versus a Windows machine.

Connect to a network compile server (farm)—If you are working on a team, you can use the LabVIEW FPGA Compile Farm Toolkit to set up a compile farm with multiple workers. This toolkit helps you create an on-site server to manage FPGA compilations. You can connect as many worker computers as you need, and the central server software manages the farming out of parallel compilations and queuing. This is an effective way to reduce compile times if you cannot use cloud technology for your project.

Connect to the LabVIEW FPGA Compile Cloud Service—This service helps you compile your FPGA VIs on Linux with the latest dedicated high-RAM high-end computers. Depending on the size of your LabVIEW FPGA VI, you may notice substantial reductions in your compile times compared to compiling on your Windows desktop. Compiling in the cloud also adds the capability to compile many VIs in parallel. To try a 30-day trial of the LabVIEW FPGA Compile Cloud Service, visit ni.com/trycompilecloud.

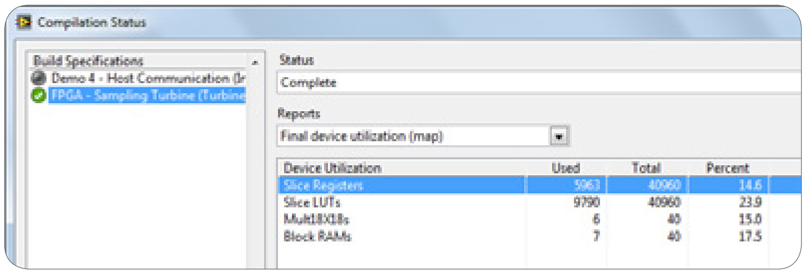
Reading the Compile Report

After compiling an FPGA VI, LabVIEW displays a Compile Report that contains information about the overall size and speed of the application. This information can help you decide how to optimize your code if necessary.

Note that the compiler algorithm is not deterministic, and you might get different results from one compile to the next even if you do not change the VI or the compiler settings. You can change the compiler settings if the compile fails by right-clicking the build specification in the LabVIEW project and selecting **Properties** from the context menu. In the Properties page, select the category **Xilinx Options**.

VI Size

The Final device utilization (map) report shown in the dialog of a successful compile offers information about the number of slice registers, slice lookup tables, and multipliers used as well as the amount of block RAM used. A recommended best practice is to keep your overall FPGA usage below 90 percent in your final application. If you upgrade your software in the future and need to recompile your VI, you may find that a different version of the Xilinx compilation tool chain uses more or less fabric. In this case, you need some extra room to work with.


Compilation Status
Build Specifications
Demo 4 - Host Communication (Ir
FPGA - Sampling Turbine (Turbine

Status: Complete

Reports: Final device utilization (map)

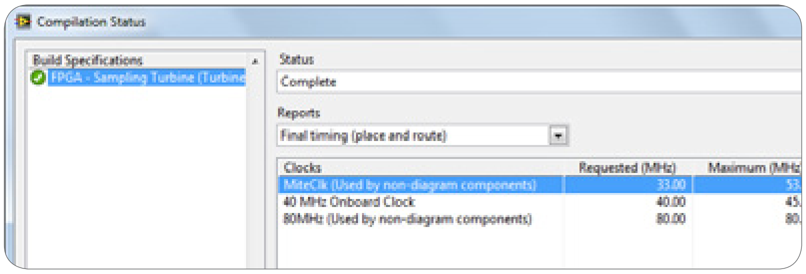
Device Utilization	Used	Total	Percent
Slice Registers	5963	40960	14.6
Slice LUTs	9790	40960	23.9
Mult18X18s	6	40	15.0
Block RAMs	7	40	17.5

Figure 5.43. Final Device Utilization (Map) Report

VI Speed

The Successful Compile Report dialog box also contains information about the clock rates of the application.

- **Requested**—Displays the clock rate at which the compiled FPGA VI runs. The default setting is 40 MHz.
- **Maximum**—Displays the theoretical maximum compile rate for the FPGA VI.



Status: Complete

Reports: Final timing (place and route)

Clocks	Requested (MHz)	Maximum (MHz
MiteClk (Used by non-diagram components)	33.00	53.
40 MHz Onboard Clock	40.00	45.
80MHz (Used by non-diagram components)	80.00	80.

Figure 5.44. Final Timing (Place and Route) Report

If the maximum rate is slower than the requested rate, an error results and the compile process stops. You must modify the application to the point where the maximum rate is equal to or greater than the requested rate. A common problem when you use SCTLs is that the Requested Rate exceeds the Theoretical Maximum. Refer to the "Single-Cycle Timed Loop" section of this chapter for more information about optimizing FPGA applications using SCTLs.

CHAPTER 6
Timing and Synchronization of I/O

Understanding how data travels between module hardware components and the LabVIEW FPGA block diagram can help you develop better programs and debug faster. This section introduces you to the different hardware architectures of analog and digital C Series I/O modules and how to communicate with each one. These modules are typically used for measurement or control signals and feature model numbers that follow this style: NI 92xx, NI 93xx, or NI 94xx.

Some basic terminology used in this section is listed below.

- ADC—Analog-to-digital converter. Discrete component that converts an input analog signal (usually voltage) into a digital representation. Front-end circuitry, also known as signal conditioning, is used to convert real-world analog signals into voltage levels within the set range of the ADC.
- DAC—Digital-to-analog converter. Discrete component that converts a digital value into an analog value. Analog output is usually a voltage, but, if you add circuitry, you can convert it into a current value.
- Arbitration—The process of providing one request priority while causing all other requests to wait.
- Jitter—Inconsistent periods between multiple iterations of a looping program structure. Measured as the difference between the longest period experienced and the nominal period requested.

LabVIEW FPGA Communication Nodes

You can use three graphical function blocks to communicate with modules from a LabVIEW block diagram. At a lower level, these programmatic interfaces vary due to hardware architecture differences. For example, the graphical function blocks that retrieve data from AI Channel 0 all look the same even though at a lower level, the raw FPGA communication differs between modules. This abstraction reduces development time and provides the open environment that supports several chassis and module combinations.

The three main function blocks that communicate with C Series modules are the I/O node, method node, and property node.

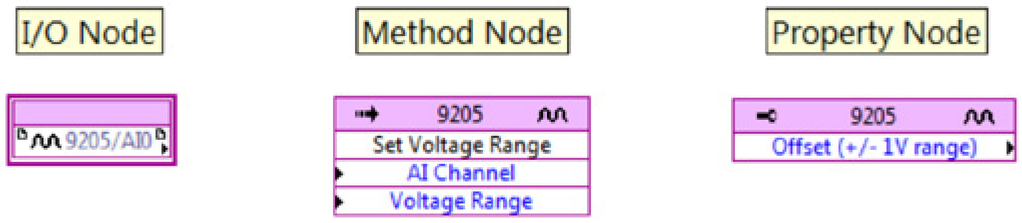


Figure 6.1. The I/O node, method node, and property node for LabVIEW FPGA visually have subtle differences.

I/O Nodes

- Pull data from hardware channels
- Read calibration information
- Are designed to be "thin" interfaces to the module (in other words, minimal data or timing manipulation)
- Block loops until data is available
    - Cannot be used in SCTLs (except I/O nodes for parallel digital lines)

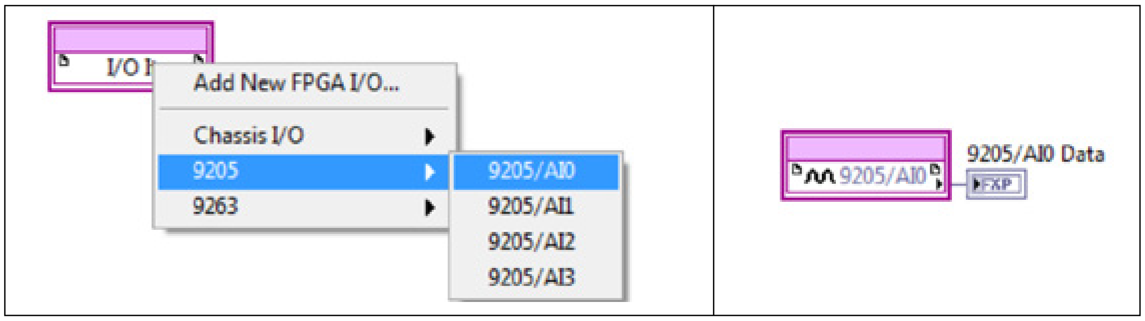


Figure 6.2. I/O Node Selection Menu (left) and an I/O Node Set to Channel AI0 (right)

Method Nodes

- Invoke features that are special to a particular set of modules
- Used when the method involves multiple parameters
- Examples include
    - Wait for change on a digital line
    - Triggering on the NI 9205 C Series analog input module

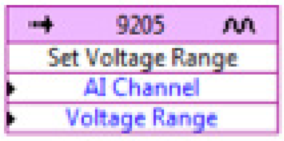


Figure 6.3. Method Node to Set Voltage Range on the NI 9205

Property Nodes

- Access module properties
- Feature some properties that are available during run time
- Examples include
    - Data rate setting for delta-sigma modules
    - Calibration constants
    - Serial number
    - Module ID
    - Vendor ID

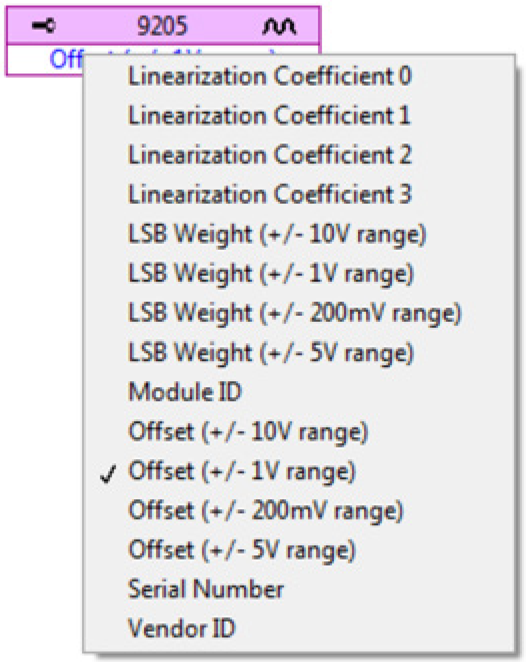

9205
Offset (+/- 1V range)
Linearization Coefficient 0
Linearization Coefficient 1
Linearization Coefficient 2
Linearization Coefficient 3
LSB Weight (+/- 10V range)
LSB Weight (+/- 1V range)
LSB Weight (+/- 200mV range)
LSB Weight (+/- 5V range)
Module ID
Offset (+/- 10V range)
✓ Offset (+/- 1V range)
Offset (+/- 200mV range)
Offset (+/- 5V range)
Serial Number
Vendor ID

Figure 6.4. Property nodes for some modules feature several options.

Specialty C Series Modules

Some specialty C Series modules such as motor drivers, CAN, serial, and SD card modules do not have a generalized API for the FPGA like the analog and digital modules do. These NI modules have examples in the NI Example Finder that include LabVIEW FPGA and LabVIEW Real-Time host code, and most third-party modules are shipped with examples. Use these examples as a starting point if you are using a specialty module in your application.

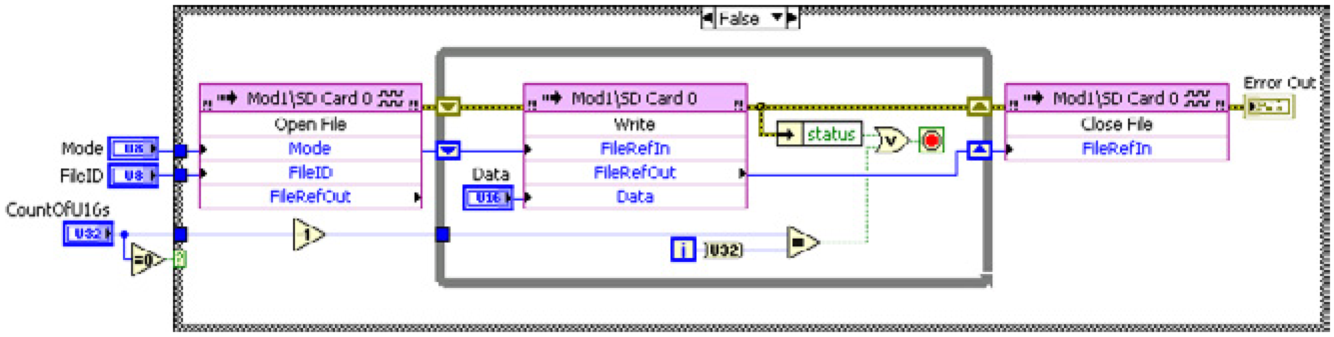


Figure 6.5. Write SD Card Example for the NI 9802

Module Classifications

This section describes the different types of C Series I/O modules. You need to understand how these modules are designed to properly implement timing and synchronization. The basic types of module classifications are presented in Figure 6.6.

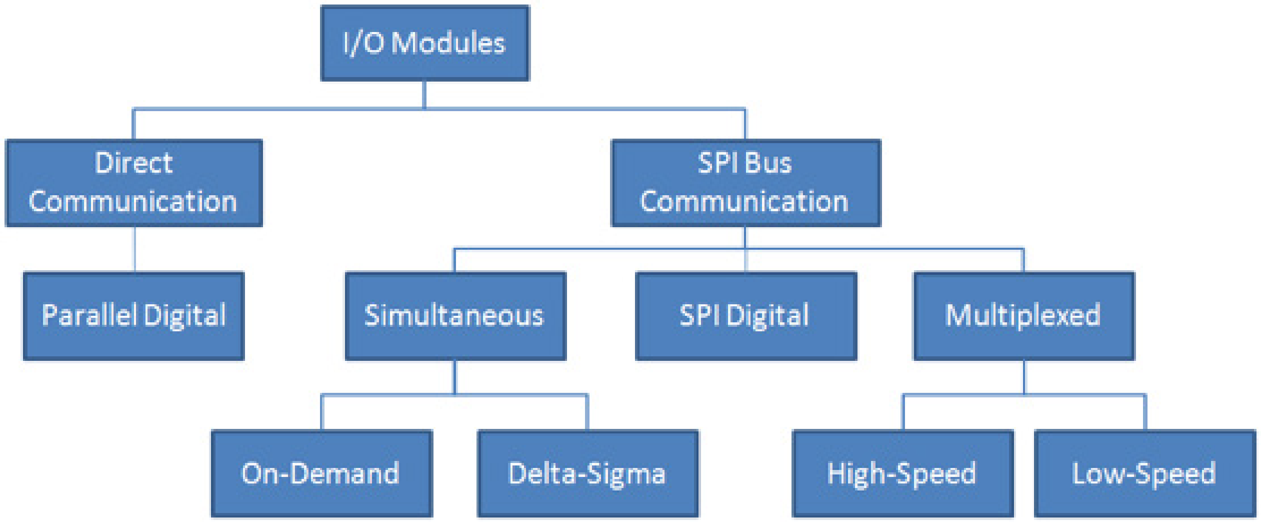


Figure 6.6. C Series Module Classification Organizational Tree

Direct FPGA Communication

Modules that feature direct communication with the FPGA in a CompactRIO chassis route signals from the I/O connector on the front of the module through the 15-pin D-SUB on the back of the module to the FPGA. This architecture makes programming with direct FPGA communication modules easier because you can think of each line as a unique programming entity. The only modules that use direct FPGA communication are digital I/O (DIO) modules with eight or fewer lines.

Parallel Digital

Because each of the digital lines on a direct FPGA communication module is programmed as a separate entity, digital modules with eight or fewer channels are also referred to as parallel digital modules. The parallel design of these modules makes them the easiest to program in LabVIEW. I/O nodes for parallel digital modules can be called from SCTLs, which make programming more efficient, and individual channel lines can be called simultaneously from separate loops, as seen in Figure 6.7, with no concern for arbitration.

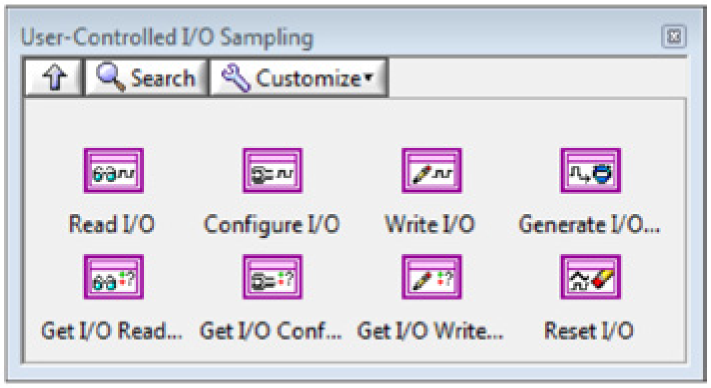


Figure 6.7. You can call lines on the same parallel digital module from separate loops.

Be aware that a call for direction change, at any time, on the DIO lines of an 8-channel parallel module causes an interrupt that you must handle appropriately on the LabVIEW block diagram to prevent a glitch or jitter on other lines in use at the time of direction change. The NI 9402 does not have this problem. Four lines are used for DIO and the other four lines are used to control the direction. Direction changes can happen independently of data transfer. The NI 9401, one of the most popular DIO modules, uses all eight DIO lines and must switch modes to communicate line states. This mode change affects all DIO lines regardless of which ones are changing state.

SPI Bus Communication

The Serial Peripheral Interface bus, or SPI, is a standard 4-wire communication protocol set up for master/slave, full-duplex (two-way simultaneous) communication. The bus clock rates typically range from 1 MHz to 70 MHz. The SPI clock rate that CompactRIO hardware uses to communicate with C Series modules varies but usually operates around 10 MHz.

The common architecture of SPI bus modules contains a complex programmable logic device (CPLD) that, on the module side, controls the timing to and data collection from the ADC/DAC chips. On the chassis side, the CPLD communicates via SPI bus back to the FPGA. Based on the specific module used and slot location, the LabVIEW API knows how and where to communicate to the individual CPLD on the module. This is why new module support is added in new versions of LabVIEW.

SPI Bus Challenges

Regardless of front-end circuitry, each SPI module communicates to the FPGA in a CompactRIO chassis over a single dedicated SPI bus. This means that even though commands to the module (from multiple loops on a LabVIEW block diagram) or data retrieval (from modules with multiple ADCs) can happen in parallel, those commands must be interlaced through a single SPI bus. LabVIEW and the NI-RIO driver can properly prioritize multiple calls that come in at the same time, but if a call comes in while the previous call is still being handled, you may introduce unwanted jitter into your control or data acquisition loop. Another way to convey this concept is to say that I/O node calls to a single module from different parts of a LabVIEW FPGA program create a "hardware race condition." This in no appreciable way limits the abilities of the module or a CompactRIO system, but it does require that you arbitrate I/O node calls within your program. The easiest way to do this is to keep all of the I/O node calls in the same loop, but you can also use sequence structures or semaphores. You can find a reference design for semaphore implementation in LabVIEW FPGA in the NI Developer Zone document Semaphore Reference Design for LabVIEW FPGA.

As you continue to read about the different classifications of modules, note that all modules that use SPI communication must adhere to the same caveat of hardware arbitration. Simultaneous Modules

Modules that are simultaneous have one ADC per channel and acquire data with no appreciable skew between channels. The two subcategories of simultaneous modules, on-demand and delta-sigma, transfer data via the SPI bus and are subject to all of the specifications and challenges of other SPI bus modules.

On-Demand Conversion

NI C Series On-Demand Simultaneous Modules	
Model Number	**Description**
NI 9215	Analog Input, ±10 V, 100 kS/s/ch
NI 9263	Analog Output, ±10 V, 100 kS/s/ch
NI 9265	Analog Output, 0 to 20 mA, 100 kS/s/ch
NI 9219	Analog Input, Universal, 100 S/s/ch
NI 9222[1]	Analog Input, ±10 V, 500 kS/s/ch
NI 9223[1]	Analog Input, ±10 V, 1 MS/s/ch

[1]These modules are simultaneous modules and can be programmed with FPGA I/O nodes, similar to other modules listed in this table. However, to run at higher speeds, you need to program them with the user-controlled I/O sampling API as outlined in the Data on Demand section below. The maximum sampling rates achievable when using both FPGA I/O nodes and the user-controlled I/O sampling API can be found in the "Maximum Sampling Rate" section of the user manual.

Table 6.1. Examples of Simultaneous Modules With On-Demand Conversion

On-demand modules, like the ones listed in Table 6.1, have few specific challenges when it comes to programming with LabVIEW FPGA. This makes them some of the easiest modules to program. The biggest caveat involves arbitration, which is shared by all modules that use SPI communication.

Data on Demand

Data on demand is less of a caveat and more of a feature. On-demand simultaneous C Series modules have the ability to return data when the I/O node is called at any interval down to the minimum conversion time as listed in the manual. This means that the acquisition can be clocked by external, irregular clocks. The Δt does not need to be a constant for an acquisition with an on-demand simultaneous module.

Pipelined Simultaneous Data or User-Controlled I/O Sampling

Example

The NI 9223 User-Controlled IO Sampling.lvpj is in the NI Example Finder.

Some modules designed for high-speed measurements exceed the data throughput capabilities of the FPGA I/O node. In these situations, you can apply user-controlled I/O sampling functions to communicate with a module. These add complexity to the program but dramatically increase the bandwidth from the module.

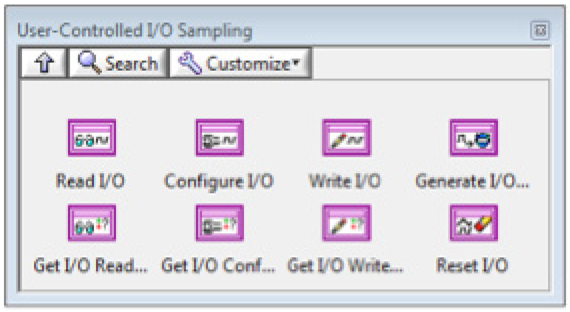


Figure 6.8. You can use the User-Controlled I/O Sampling palette for higher bandwidth communication to some modules.

When programming with the user-controlled sample method, you may find it easier to start with an existing example program that is shipped with LabVIEW. Figure 6.9 is the block diagram from the NI 9223 User-Controlled IO Sampling.lvproj, which you can find in the NI Example Finder.

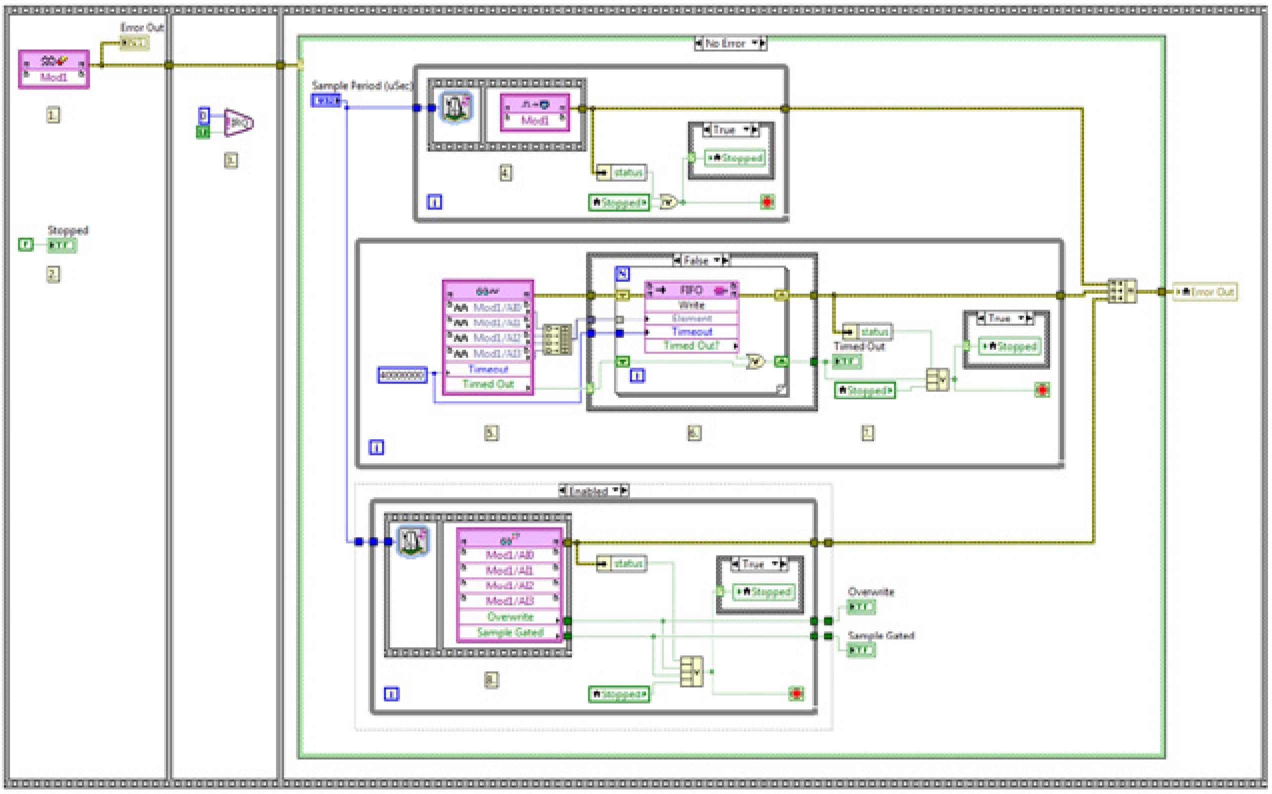


Figure 6.9. NI 9223 User-Controlled IO Sampling.lvproj Example Program

To program with the user-controlled I/O sample method, follow these steps that reference the example program in Figure 6.9.

Initialize the Process

1. Call the Reset I/O function. When this call completes, the module is ready to perform an acquisition using the user-controlled I/O sampling functions. You must call the Reset I/O function first to prepare the NI 9223 to use the other user-controlled I/O sampling functions.

2. Set the Stopped Boolean to false. This Boolean provides synchronization between the While Loops in the last sequence frame. If any loop stops, it causes the others to stop as well.

3. Use an interrupt to signal the host that the FPGA is ready to begin acquiring data and wait to start the acquisition until the host acknowledges it. This is necessary to ensure that the DMA FIFO has been started prior to acquiring data.

Loop 1

4. Call the Generate I/O Sample Pulse function to begin acquiring data. The rate at which this function is called determines the sample rate for the acquisition, so a loop timer is used to enforce the desired sample period.

Loop 2

5. Call the Read I/O function to read data acquired from the module. This function is configured to read a single sample from each channel on the module. Because this function waits for data to become available, a generous but noninfinite timeout is provided. In the case of the default 40 MHz top-level clock, the timeout is 1 second.
6. Write the acquired data into the DMA FIFO.
7. If a timeout occurred either while waiting for data from the module or waiting to write the data to the DMA FIFO, then report the timeout to the host and stop the VI.

Loop 3

8. Call the Get Read I/O Status function at the same rate you call the Generate I/O Sample Pulse function. This checks the status of every sample you acquire. If an "overwrite" or "sample gated" occur (output = true), then report the status to the host and stop the VI. In the example, this loop is encased in a diagram disable structure to make it easily removable from the application. Calling the Get Read I/O Status function is useful for development and debugging but not strictly necessary for deployment if the application does not feature variable timing in the application. In the Figure 6.9 example, you cannot obtain a sample gated status unless the top loop's sample period is less than the minimum sample period supported by the module. However, this application could produce an overwrite status if the host VI cannot read from the DMA FIFO fast enough.

Delta-Sigma Modulation

Example

The NI 9234 Getting Started.lvpj
is in the NI Example Finder.

NI C Series Simultaneous Modules With Delta-Sigma Modulation	
Model Number	**Description**
NI 9229	Analog Input, ±60 V, 50 kS/s/ch
NI 9239	Analog Input, ±10 V, 50 kS/s/ch
NI 9233	Analog Input, IEPE, 50 kS/s/ch
NI 9234	Analog Input, IEPE, 51.2 kS/s/ch
NI 9235	Analog Input, 120 Ω ¼ bridge, 50 kS/s/ch
NI 9236	Analog Input, 350 Ω ¼ bridge, 50 kS/s/ch
NI 9237	Analog Input, ¼, ½, full bridge, 50 kS/s/ch
NI 9225	Analog Input, 300 V_{rms}, 50 kS/s/ch
NI 9227	Analog Input, 5 A_{rms}, 50 kS/s/ch

Table 6.2. Examples of Simultaneous Modules With Delta-Sigma Modulation

Many C Series modules designed for high-speed, dynamic measurements use delta-sigma (ΔΣ) converters. To better understand how these modules work, you must first know the fundamentals of delta-sigma modulation. The Greek letters delta and sigma are mathematical symbols for difference and sum, respectively. The modulation circuit of a delta-sigma converter compares the running sum of differences between the desired voltage input, V_{in}, and a known reference voltage, V_{ref}. The output from the comparator becomes a bitstream that is sent into a digital filter and a 1-bit DAC. Because of this negative feedback, the differences oscillate around 0 V, or ground. The digital filter effectively keeps track of how many times the difference is above 0 V, and, based on this count along with the reference voltage, you can determine the input voltage. This modulation loop runs at a much higher frequency than the actual output frequency of the converter.

C Series modules with delta-sigma converters feature an oversample clock that runs the modulation circuitry. Oversample clocks, which run at 12 MHz or faster, affect timing, synchronization, and programming paradigms. The following list provides insight into the specific challenges of C Series modules that use delta-sigma modulation.

- Need a Sync Pulse to Reset—Oversample clocks need to be "reset" before they are used. This is why there is a LabVIEW FPGA I/O node to send a "start" event to the module.

- Time to Data Ready Is Non-Zero—The time between the "start" event and data availability is specified as "time to first data." This time may vary slightly between different delta-sigma-based modules and greatly between modules of other types. On-demand modules have a "time to first data" of zero. You can find documentation on how to align the data sets in KnowledgeBase 4DAEUNNQ: How Do I Compensate for Different Group Delays With C Series Modules in LabVIEW FPGA? and KnowledgeBase 53CHLD6C: What Is the Best Method to Synchronize Two Different DSA Modules in LabVIEW FPGA?

- Sample Rates Are Discrete and Specific—Because of the oversample clock and the digital filter, delta-sigma modules can run only at discrete sample rates. These sample rates are a function of a divisor and the oversample clock. This is why the "rate" input for a delta-sigma module is an enumerated data type of predetermined sample rates. If you try to input a sample rate that is not supported, it is rounded to the next highest available sample rate.

- Minimum Sample Rates Are Greater Than 1 kHz—The minimum sample rate for delta-sigma modules is often over 1 kHz. Use averaging, filtering, or some form of decimation to further reduce your data set beyond the rate the digital filter on the module outputs.

- No Irregular or External Clocks—Delta-sigma modules cannot report data "on demand" and thus do not work with irregularly timed I/O node calls because the iterative process must complete a large number of loops before returning accurate data. The I/O node for a delta-sigma module always blocks for the exact (Δt) for which the module has been set to acquire. An I/O node call at an interval less than Δt must wait until the full sample period is complete. This gives the oversample circuitry enough time to calculate an accurate value. To compensate for this, implement a resampling algorithm on the FPGA before processing your data or using it in a control loop.

- Physically Share Oversample Clock to Synchronize ΔΣ Modules—Two delta-sigma modules that need to be synchronized must share the same oversample clock. To synchronize modules, the oversample clock from one module must be exported from the right-click properties menu as the "source" module in the LabVIEW project. It must be imported from the same menu of the other "client" modules. Any module can be a "source" or a "client" module—it is up to the programmer's discretion. Remember, changes made to a module property window from the project view cause a recompile and cannot be changed at run time.

You can see many of these caveats in the block diagram of the example program NI 9234 Getting Started.lvprj shown in Figure 6.10.

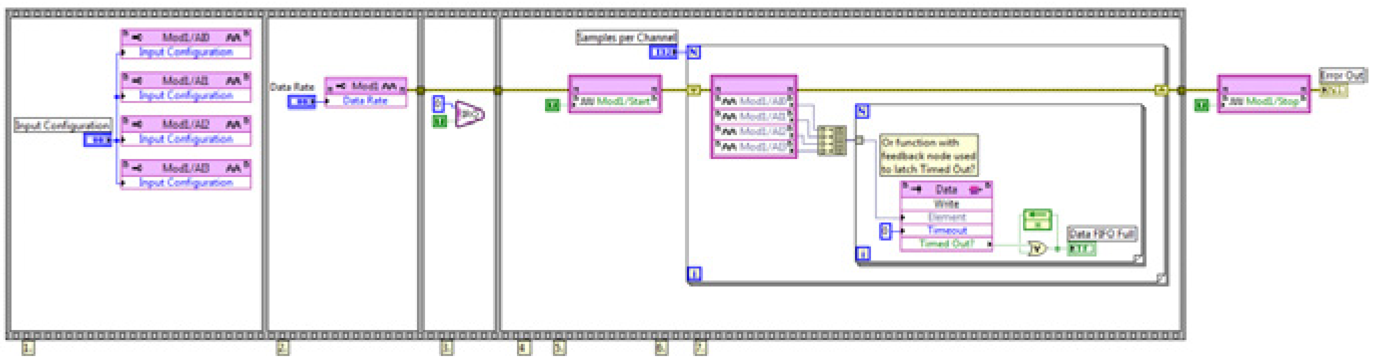


Figure 6.10. Block Diagram From NI 9234 Getting Started.lvprj

You can find example programs for all of the delta-sigma modules in the NI Example Finder.

Multiplexed

One of the more expensive components of a module is the ADC. By using a multiplexer, also known as a mux, to route multiple channels through a single ADC, multiplexed modules offer higher channel counts at lower per channel prices than simultaneous modules.

Before learning how to program these modules, you need to know some specification-level details. First, the sample rates are often listed as the total rate of all channels put together, also known as an aggregate rate. From the module hardware standpoint, all channels selected must run at the same rate (aggregate rate divided by number of channels), but from the program standpoint, you can remove samples from select channels with FPGA processing. Second, it is important to note that there is an interchannel delay, or skew, between all channels in a multiplexed module. You can implement processing in the FPGA to compensate for this skew via shifting or data resampling, but most systems that incorporate multiplexed modules are not impacted by this small offset. If hardware-based phase alignment is important to your system, you should select a module with multiple ADCs.

You can choose from two main subsets of multiplexed modules: high speed and low speed.

High Speed

High-speed multiplexed modules implement a double pipeline to increase the throughput of data to the chassis. With a double pipeline, the first valid data cannot be returned until the process has run two complete iterations. Once the first two iterations have run to "prime" the pipeline, the subsequent iterations begin to yield valid data. When using an I/O node to sample channels, the pipeline is automatically managed by the FPGA I/O node, and the channels within the FPGA I/O node are sampled in numerical order regardless of the order they appear in the node. If the first two channel requests in the FPGA I/O node do not match the two channel requests stored in the module pipeline, there is a delay before the first channel sample occurs. This delay is caused by the FPGA I/O node automatically updating the module channel sample pipeline, which takes two channel sample cycles. This situation could happen if you have I/O nodes addressing the same module from different parts of the block diagram. For example, if in an "init" case of a Case structure, you read channels 0, 1, and 2 from an I/O node and then in the "acquire" case, you read from channels 5, 6, and 8, you incur the pipeline updating penalty because the module is already primed for channels 0, 1, and 2 and needs to flush the pipeline. If this delay causes problems, the workaround is to acquire all of the channels in the same step and place data from channels 5, 6, and 8 into a FIFO to be called upon later.

NI C Series High-Speed Multiplexed Modules	
Model Number	**Description**
NI 9205	Analog Input, ±10 V, 250 kS/s (aggregate rate)
NI 9206	Analog Input, ±10 V, 250 kS/s (aggregate rate)
NI 9201	Analog Input, ±10 V, 500 kS/s (aggregate rate)
NI 9221	Analog Input, ±60 V, 800 kS/s (aggregate rate)
NI 9203	Analog Input, ±20 mA, 200 kS/s (aggregate rate)

Table 6.3. Examples of High-Speed Multiplexed Modules

I/O Sample Method for High-Speed Multiplexed Modules

The NI 9205 Basic I/O Sample Mode.lvpj
is in the NI Example Finder.

Some high-speed multiplexed modules, such as the NI 9205 and NI 9206, have an alternate method for programming. This method, referred to as the "I/O sample method," is more difficult to implement but takes up less FPGA space and provides an easier input into DMA nodes on a LabVIEW block diagram. This lower-level access to the module does not account for the double-pipeline architecture, so you must explicitly discard the first two data points returned from the I/O sample method because they are invalid. The step that adds the most difficulty is building the channel array to feed into the sample method. You can find an example program for this method, NI 9205 Basic I/O Sample Mode, in the NI Example Finder.

Low Speed

Low-speed modules do not require the same amount of bandwidth as high-speed modules and do not implement a pipeline. This makes the LabVIEW implementation straightforward. You face SPI bus module caveats when you work with low-speed multiplexed modules.

NI C Series Low-Speed Multiplexed Modules	
Model Number	**Description**
NI 9211	Thermocouple, 14 S/s (aggregate rate)
NI 9213	Thermocouple, 1200 S/s (aggregate rate)
NI 9214	Thermocouple, 1088 S/s (aggregate rate)
NI 9217	RTD, 400 S/s (aggregate rate)
NI 9207	Analog Input Combo V/I, 500 S/s (aggregate rate)
NI 9208	Analog Input, ±20 mA, 500 S/s (aggregate rate)

Table 6.4. Examples of Low-Speed Multiplexed Modules

SPI Digital

Digital modules that feature more than eight lines exceed the number of pins allowed for direct FPGA communication and, thus, communicate over the SPI bus. These modules operate with a convert pulse, and they latch inputs and update outputs simultaneously with each convert. As with any other SPI bus module, all of the lines on a 32-channel digital I/O module are routed through the same communication lines to the backplane. You should control the I/O node calls to SPI bus modules with dataflow to prevent simultaneous calls resulting in jitter. These calls include normal I/O calls as well as commands to change line direction from input to output.

Synchronizing Modules

Some applications, such as vibration or sound measurement, require a high level (sub 100 nS) of synchronization between channels. This section discusses timing and synchronization of both delta-sigma-based modules and scanned (SAR) modules. Any NI C Series I/O module that is not classified as delta sigma is classified as SAR.

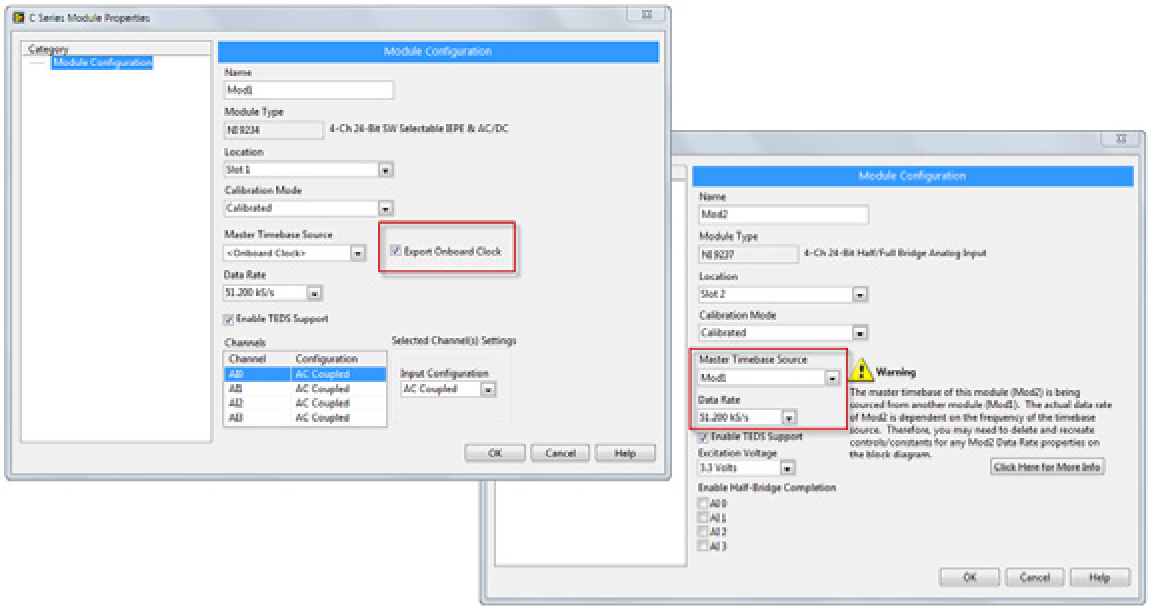
Synchronizing Delta-Sigma Modules

Example

The NI 9205 Basic I/O Sample Mode.lvpj
is in the NI Example Finder.

To synchronize delta-sigma modules in CompactRIO hardware, you need to physically share the oversample clock and start triggers between all of the modules. You can find the phase match specification between channels and between modules in the specifications section of the manual for many modules. Delta-sigma-based modules may be synchronized even if they are not the same model number.

1. Select one of the modules, "the master," to export the clock to the backplane. The other modules are set to import this clock from the backplane. Whichever module you select as the master overrides both the timebase and available rates. For example, if you want 51.2k rates in the system, select the NI 9234 or NI 9232. If you want 50k rates, select the NI 9237. You modify import/export properties from the properties window accessed from the right-click menu of the module in the LabVIEW Project Explorer. Import/export settings cannot be set programmatically because this information is compiled.



Module Configuration
Name: Mod1
Module Type: NI 9234 4-Ch 24-Bit SW Selectable IEPE & AC/DC
Location: Slot 1
Calibration Mode: Calibrated
Master Timebase Source: <Onboard Clock>
☑ Export Onboard Clock
Data Rate: 51.200 kS/s
☑ Enable TEDS Support
Channels

Channel	Configuration
AI0	AC Coupled
AI1	AC Coupled
AI2	AC Coupled
AI3	AC Coupled

Selected Channel(s) Settings
Input Configuration: AC Coupled

Module Configuration
Name: Mod2
Module Type: NI 9237 4-Ch 24-Bit Half/Full Bridge Analog Input
Location: Slot 2
Calibration Mode: Calibrated
Master Timebase Source: Mod1
Data Rate: 51.200 kS/s
☑ Enable TEDS Support
Excitation Voltage: 3.3 Volts
Enable Half-Bridge Completion
☐ AI 0
☐ AI 1
☐ AI 2
☐ AI 3

Warning
The master timebase of this module (Mod2) is being sourced from another module (Mod1). The actual data rate of Mod2 is dependent on the frequency of the timebase source. Therefore, you may need to delete and recreate controls/constants for any Mod2 Data Rate properties on the block diagram.

Figure 6.11. Export the clock from the chosen master module (left) and then select it from all subsequent modules to be synchronized (right).

2. On the block diagram, create a property node for each I/O module and use the Data Rate enum to specify the rate, as shown in Figure 6.12. Note that even though the I/O modules share the same sampling rate, you must create a unique Data Rate enum for each property node (right-click on the property node for each module and select "create constant"). This ensures that the enum integer is properly matched to the expected rate for the specific I/O module.

3. Create a Start Trigger for each I/O module and place them into the same I/O node. This ensures the start triggers are properly routed.

4. Place all channel reads from all synchronized modules in the same I/O node as seen in Figure 6.12. Using this process, you can mix and match any of the existing simultaneous delta-sigma modules.

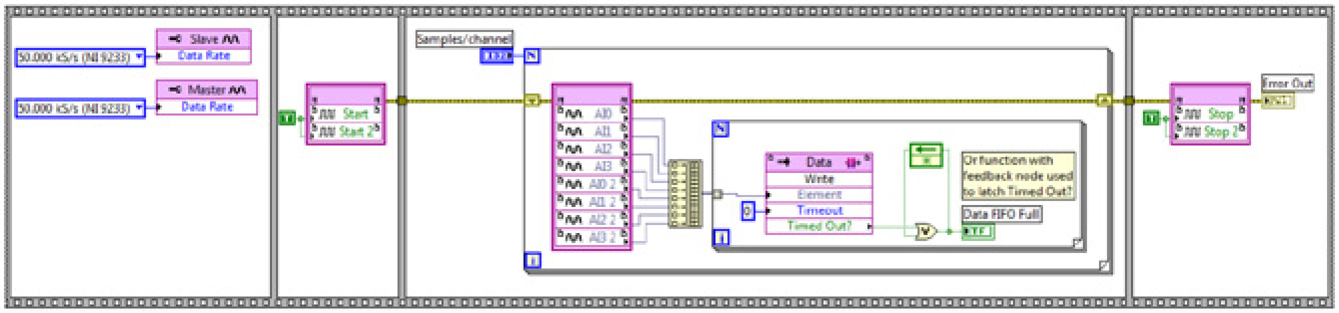


Figure 6.12. Block Diagram From Synchronizing NI 9233 Modules.lvpj

The best method for synchronizing different delta-sigma modules in LabVIEW FPGA is to have the I/O nodes for each module in the same While Loop. If you place the I/O nodes for the different modules in parallel While Loops, you must address additional startup delays. You also need to take into account the group delay for each module because the modules acquire data at the same time when in the same loop. View KnowledgeBase 4DAEUNNQ: How to Compensate for Different Group Delays With C Series Modules in LabVIEW FPGA for tips on this topic.

Synchronizing Simultaneous On-Demand Modules

This process is easier than delta-sigma modules because there is no oversample clock to share. These modules are clocked by a convert pulse that originates from the programmed I/O node on the FPGA. To synchronize the convert pulse, place all channel reads or updates in the same I/O node call. You can mix analog input, analog output, and digital channels in the same I/O node with minimal skew.

Synchronizing Multiplexed Modules

Multiplexed modules that share the same model number operate in "lock step" as they move through the channels. Channel 0 on each module is synchronized as are channels 1 through n. This is more informational than instructional because multiplexed applications rarely are affected by interchannel delay.

Synchronizing Delta-Sigma and Scanned (SAR) Modules

Synchronizing delta-sigma modules with SAR (non-delta-sigma) modules is slightly more complex. This is because delta-sigma modules have their own timebases, while SAR modules are a slave to the FPGA clock. From a programming perspective, loop timing with delta-sigma modules is determined by the data rate node; While Loop timing with SAR modules is determined by using the FPGA loop timer. The best way to synchronize delta-sigma with SAR modules is to design your application for delta-sigma timing (similar to Figure 6.13) and then add your I/O blocks for the SAR modules as a separate I/O node as shown in Figure 6.13. This requires synching your SAR modules to a delta-sigma module clock.

Note: Delta-sigma-based modules and SAR modules should not share the same FPGA I/O node because they execute in series and limit the maximum data rate of the system.

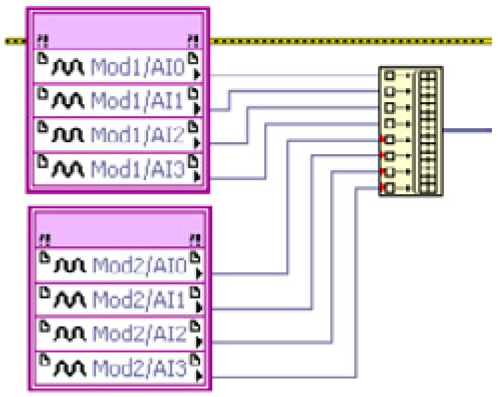


Figure 6.13. FPGA I/O Nodes for Hybrid Applications (Delta-Sigma and SAR Modules)

An application becomes more complicated when it requires multirate synchronization between delta-sigma and SAR modules. Since these modules have different timebases, they cannot share a clock when they are separated into two or more loops executing at multiple rates. The best option is to separate them into multiple loops and then expect some drift between the timebases over time, in addition to a phase offset caused by the varying startup times.

Example

LabVIEW example code
is provided for this section.