LabVIEW™ Real-Time 2 Course Manual

Course Software Version 2010
September 2010 Edition
Part Number 373247A-01

Worldwide Technical Support and Product Information
ni.com

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

Worldwide Offices

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

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

Contents

Lesson 6
Optimization

Appendix A
Additional Information about LabVIEW Real-Time

Appendix B
Instructor's Notes

Appendix C
Additional Information and Resources

2

Advanced Real-Time Intertask Communication Methods

This lesson describes different methods of sharing data between tasks on the RT target. You will learn the appropriate use case for each method.

Topics

A. Review – Single-Process Shared Variables with the RT FIFO Enabled

B. RT FIFO Functions

C. Functional Global Variables

A. Review – Single-Process Shared Variables with the RT FIFO Enabled

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

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

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

Note If you enable the Real-Time FIFO on a shared variable of waveform datatype, the variant element of the waveform does not transfer because variants are variable-sized and therefore incompatible with the Real-Time FIFO.

B. RT FIFO Functions

Use the Real-Time FIFO functions to share data between VIs running on an RT target. An RT FIFO acts like a fixed-sized queue, where the first value you write to the FIFO is the first value that you can read from the FIFO. RT FIFOs and LabVIEW Queues both transfer data from one VI to another. However, unlike a Queue function, an RT FIFO ensures deterministic behavior by imposing a size restriction on the data you share and preallocating memory for the data. You must define the number and size of the RT FIFO elements and ensure that you do not attempt to read and write data of different sizes. Both a reader and writer can access the data in an RT FIFO at the same time, allowing RT FIFOs to work safely from within a deterministic loop.

Because of the fixed-size restriction, an RT FIFO can be a lossy communication method. Writing data to an RT FIFO when the FIFO is full overwrites the oldest element. You must read data stored in an RT FIFO before the FIFO is full to ensure the transfer of every element without losing data. Check the **overwrite** output of the RT FIFO Write function to ensure that you did not overwrite data. If the RT FIFO overwrites data, the **overwrite** output returns a TRUE value.

A larger FIFO gives the normal priority loop more time to catch up if it falls behind, which can help avoid FIFO overwrites. However, setting a FIFO too large wastes memory.

Use the RT FIFO Create function to create a new FIFO or to create a reference to a FIFO that you previously created. Use the RT FIFO Read and RT FIFO Write functions to read and write data to the FIFO. Use the RT FIFO Delete function to delete a reference to an RT FIFO and release the memory allocated to the FIFO on the RT target.

Note If you use a Real-Time FIFO to transfer waveform data, the variant element of the waveform does not transfer because variants are variable-sized, and therefore incompatible with the Real-Time FIFO.

Using RT FIFO Functions

Use the RT FIFO Create function to create a new FIFO or open a reference to a FIFO that you previously created. Use the RT FIFO Read and RT FIFO Write functions to read and write data to the FIFO. Use the RT FIFO Delete function to delete a reference to an RT FIFO and release the memory allocated to the FIFO on the RT target. Refer to the *RT FIFO Functions* topic in the *LabVIEW Help* for more information about the RT FIFO functions and the data types supported by the RT FIFO functions.

Defining Read and Write Modes for Real-Time FIFOs

An RT FIFO function can wait until an empty slot becomes available for a write operation or wait until a value is available for a read operation. You can specify a read and write mode for an RT FIFO that defines the way you read a value from an empty FIFO or write a value to a FIFO that does not have an empty slot. You can specify one of the following modes for reads and writes on the **r/w modes** input of the RT FIFO Create function:

- **Polling**—Use this mode to optimize the throughput performance of read and write operations. The polling mode continually polls the FIFO for new data or an open slot. The polling mode responds quicker than the blocking mode to new data or new empty slots, but requires more CPU overhead. Use the **timeout in ms** input of the RT FIFO Read or RT FIFO Write function to specify the amount of time that a write operation should poll for an empty slot or the amount of time a read operation should poll for new data. You also can use the **overwrite on timeout** input of the RT FIFO Write function to specify whether to overwrite the oldest value in the RT FIFO when the value of the **timeout in ms** input expires.
- **Blocking**—Use this mode to optimize the utilization of the CPU during read and write operations. The blocking mode allows the thread of the VI to sleep while it waits, allowing other tasks in the system to execute. Use the **timeout in ms** input of the RT FIFO Read or RT FIFO Write

function to specify an amount of time a read operation can wait for a new value or an amount of time a write operation can wait for an empty slot. You also can use the **overwrite on timeout** input of the RT FIFO Write function to specify whether to overwrite the oldest value in the RT FIFO when the value of the **timeout in ms** input expires.

Note If you use the RT FIFO Create function to return a reference to an existing RT FIFO, the reference uses the read and write mode of the existing FIFO and ignores the modes specified with read/write modes.

Static versus Dynamic Configuration

Shared variables are configured statically. This means that the properties of a shared variable are defined through interactive dialog boxes as you write your application. For dynamically configured entities such as a Real-Time FIFO, properties are defined at run-time.

Static configuration simplifies programming and conserves space on the block diagram because you do not need to create controls and constants for each configuration option. Shared variables do not require reference wires, thus keeping the block diagram cleaner. In general, you must know the value of all variable properties before running the VI. This means that user input or acquired data cannot determine the values of properties. You can configure some shared variable properties dynamically through VI Server calls, but this requires significant programming.

Use the dynamic configuration capability of the RT FIFO functions to specify configuration settings as the program runs. For example, you can set the size of the RT FIFO based on user input or by calculating loop frequencies. You also can destroy and recreate an RT FIFO with new properties as a program runs. For example, if overflow errors occur consistently you can destroy the RT FIFO and create a new, larger RT FIFO. The ability to create and destroy the RT FIFO helps manage memory in systems where memory is limited. Dynamic configuration also makes it easier to determine the properties of the RT FIFO by inspecting the block diagram.

Choosing Your RT FIFO Method

Consider the following characteristics of shared variables and RT FIFO functions when choosing which RT FIFO method to use.

- Shared variables store the timestamp of each piece of data they receive, making write operations slightly slower than the low-level FIFO functions.
- You can change shared variable FIFOs to other shared-variable types. For example, without any significant changes to the block diagram

code, you can change a single-process shared variable FIFO to a network-published shared variable FIFO to communicate with the host.

- For small applications, single-process shared variables with the RT FIFO enabled are a good choice for inter-task communication. However, to create a scalable architecture for large applications, use the RT FIFO functions instead.
- When you need a scalable inter-task communication architecture for a large application, use the RT FIFO Create function to create RT FIFOs programmatically. For example, you can use the RT FIFO Create function inside a For Loop to create as many RT FIFOs as you need. You then can use the RT FIFO Read and RT FIFO Write functions inside For Loops to read from and write to all your RT FIFOs consecutively. Using this technique, you can scale your application up to use as many RT FIFOs as you need while keeping the size of your block diagram manageable.

C. Functional Global Variables

Functional global variables (FGV) are VIs that use loops with uninitialized shift registers to hold global data. A functional global variable usually has an action input parameter that specifies which task the VI performs. The VI uses an uninitialized shift register in a While Loop to hold the result of the operation. Functional global variables are discussed in detail in the *LabVIEW Core 1* course.

Normally, functional global variables act as a shared resource because they are implemented as non-reentrant subVIs. However, you can implement functional global variables such that they are not a shared resource. First, you must set the priority of the functional global variable VI to **subroutine**. Then, you can right-click any functional global variable subVI in the calling VI and select **Skip Subroutine Call If Busy** to force the execution system to skip the subVI if the functional global variable subVI is currently running in another thread. Skipping a functional global variable subVI helps in deterministic loops because the deterministic loop does not wait for the functional global variable subVI resource if it is already currently in use.

If you skip the execution of a subVI, the subVI returns the default indicator value. If you want to detect the execution of a functional global variable, wire a TRUE constant to a Boolean output on the functional global variable block diagram, and ensure that the default indicator value is set to FALSE. If the Boolean output returns TRUE, the functional global variable executed. If the Boolean output returns the default value of FALSE, the functional global variable did not execute.

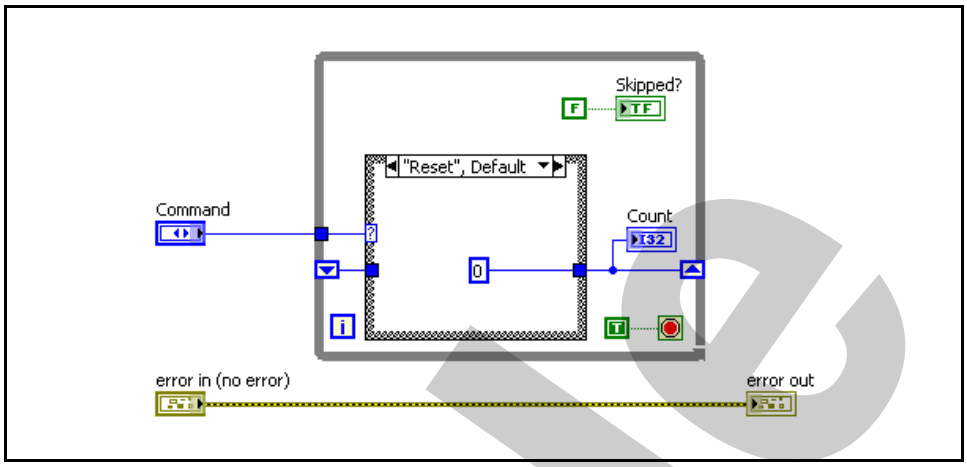

Figure 2-1. Example Functional Global Variable VI Block Diagram with Boolean Indicator

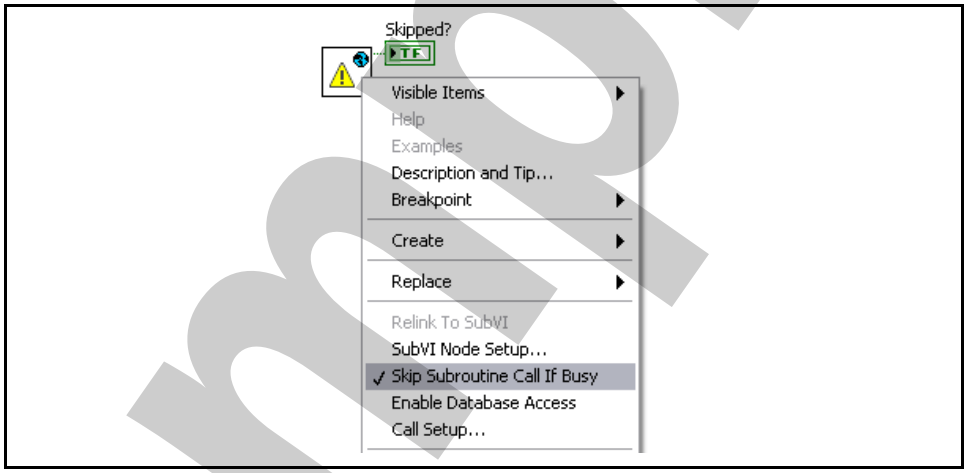

Figure 2-2. Skip Subroutine Call If Busy Selection

Skip functional global variables in deterministic loops but not in non-deterministic loops. In non-deterministic loops, you can wait to receive non-default values.

Functional global variables can be a lossy communication method if a VI overwrites the shift register data before another VI reads the data.

Using Functional Global Variables for Encapsulation

A critical section of code is code that must behave consistently in all circumstances. When you use multi-tasking programs, one task may interrupt another task as it is running. In nearly all modern operating systems, this happens constantly. Normally, this does not have any effect upon running code, however, when the interrupting task alters a shared resource that the interrupted task assumes is constant, then a race condition

occurs. Race conditions and critical sections of code are discussed in the *LabVIEW Core 1* course.

One way to protect critical sections is to place them in non-reentrant subVIs. You can only call a non-reentrant subVI from one location at a time. Therefore, placing critical code in a non-reentrant subVI keeps the code from being interrupted by other processes calling the subVI. Using the functional global variable architecture to protect critical sections is particularly effective because shift registers can replace less protected storage methods like global or single-process shared variables. Functional global variables also encourage the creation of multi-functional subVIs that handle all tasks associated with a particular resource.

After you identify each section of critical code in your VI, group the sections by the resources they access, and create one functional global variable for each resource. Critical sections performing different operations each can become a command for the functional global variable, and you can group critical sections that perform the same operation into one command, thereby re-using code.

You can use functional global variables to protect critical sections of code.

Using Functional Global Variables for a Circular Buffer

A circular buffer is a data structure of a fixed size which operates as if its ends were connected together to form a ring. The circular buffer is a useful way to buffer data between two operations such as data acquisition and analysis. It allows you to decouple and parallelize different operations which would normally be used in a sequential manner. It is also useful in applications where operations using the same data set execute at different intervals. Queues and RT FIFOs are examples of a circular buffers.

If you need functionality beyond an RT FIFO, you can implement a custom circular buffer using a functional global variable. Examples of custom functionality include the following:

- Preview element without removing it from the buffer.
- Support two different read modes: Read (continuous) and Read (most recent). The Read (continuous) function always reads data out in the same order that it was written. It reads out like a FIFO. The Read (most recent) only returns the most recent data in the circular buffer, and it could return the same data from the circular buffer in successive reads.
- Obtain pretrigger data by filling buffer before receiving trigger.

Developing your own custom circular buffer requires extra development time. The custom circular buffer will most likely involve the following tasks:

- Storing buffer values as array data in an uninitialized shift register
- Storing a write pointer and a read pointer
- Using array functions to operate on buffer values
- Alerting overflow and underflow buffer status

For an example of using a functional global variable to implement a custom circular buffer, you can view the *Software Circular Buffer in LabVIEW* document on `ni.com` and download the example.

If you must use a custom circular buffer functional global variable in a deterministic loop, make sure that you select the **Skip Subroutine Call if Busy** option so the functional global variable can execute deterministically.

Using Functional Global Variables for a Current Value Table

A Current Value Table (CVT) is a central data repository containing only the current values of all its data. A CVT centralizes operations with data shared by many processes, allows many application components to share a common data repository, and allows direct access to latest values of data.

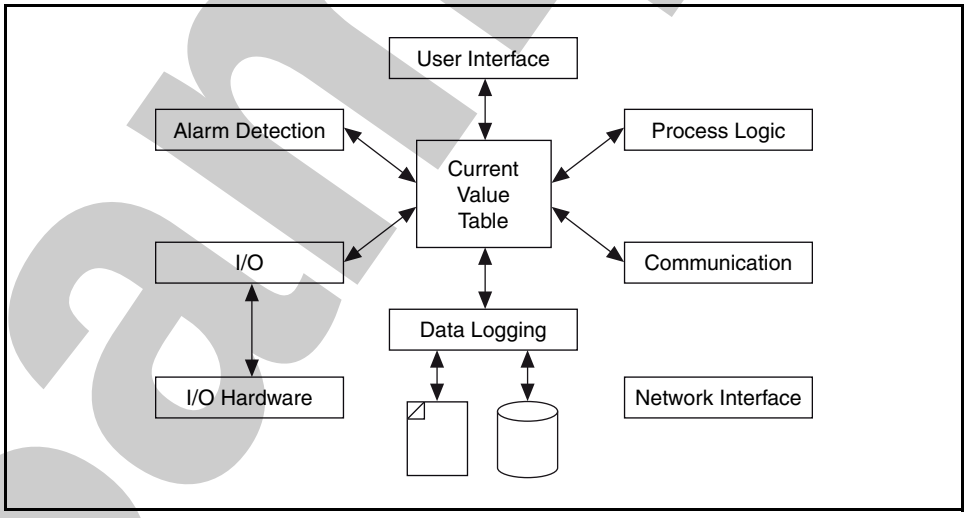

Figure 2-3. Example Application Using Current Value Table

A CVT can be implemented using a set of functional global variables that developers use to store and retrieve data asynchronously from different parts of an application. Implementing a CVT using functional global variables will allow you to:

- Dynamically create variables at run-time. For example, you could create and initialize variables based on a configuration file.

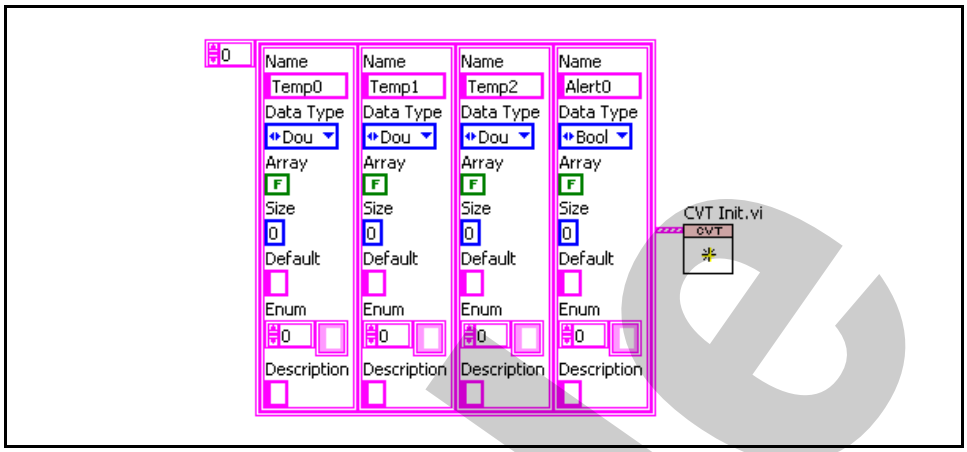

Figure 2-4. Dynamically Create Variables at Run-Time

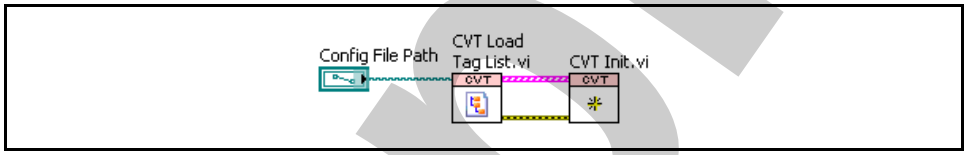

Figure 2-5. Dynamically Create Variables from a Config File

- Dynamically access large groups of variables. For example, you could initialize 300 PWM variables to the same value and perform the same logic on 500 thermocouple variables.

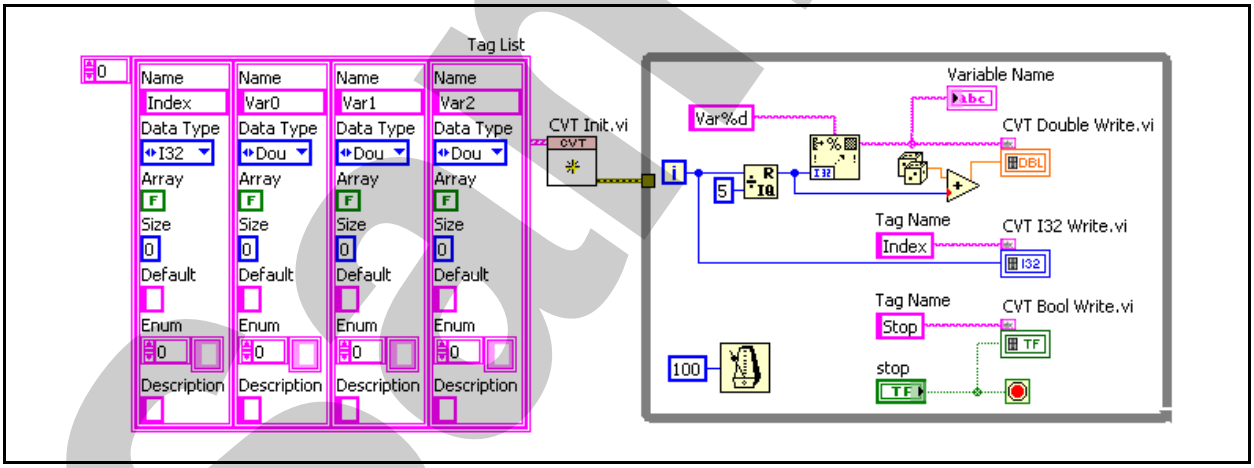

Figure 2-6. Dynamically Write to Large Groups of Variables

The main reason to use a CVT is for dynamic access to your variables.

Example CVT Implementation

Developing your own CVT will require extra development time. To download a completed CVT implementation using functional global variables, view the *Current Value Table (CVT) Reference Library* document

on ni.com and download the implementation. This implementation will install the CVT VIs to the User Library palette in LabVIEW.

This CVT implementation is implemented in a two layer hierarchy consisting of core VIs and API VIs. The core contains all of the functionality of the CVT, including the data storage mechanism and additional service functions. The API VIs provide access to the CVT functionality in a simple interface. There are three groups of API functions that provide slightly different access to the CVT and vary in flexibility and performance, as well as ease-of-use.

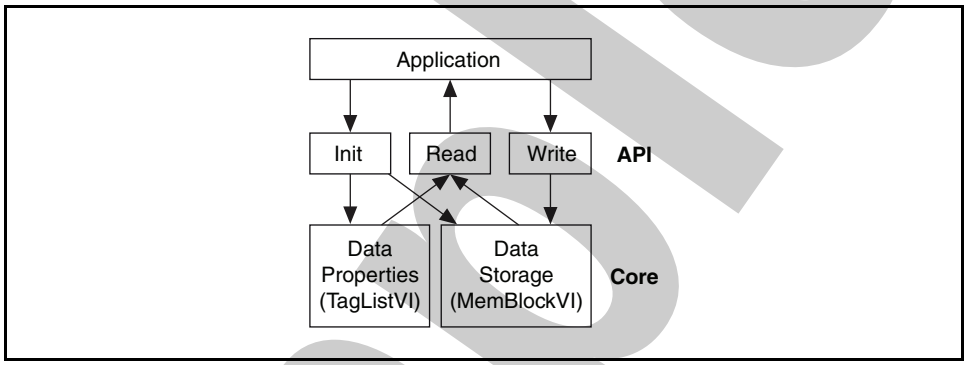

Figure 2-7. Example CVT Implementation VI Hierarchy

Across the core and API implementations, the CVT manages different data types in different sets of VIs. For all of the core and API VIs there are separate implementations for each CVT supported data type. Users can extend the supported data types in the CVT by using the current VIs as a template and adding VIs for additional data types. Booleans, 32-bit integers, double precision floating-point numbers, and strings are currently supported.

Summary – Quiz

1. Match the following items with their attributes.

Functional Global Variables	A. Configure FIFO size dynamically using block diagram code.
Shared Variables with RT FIFO enabled	B. Configure FIFO size statically using dialog boxes.
RT FIFO Functions	C. Can have several inputs and outputs. Can contain additional custom code.

Summary – Quiz Answers

1. Match the following items with their attributes.

Functional Global Variables	**C. Can have several inputs and outputs. Can contain additional custom code.**
Shared Variables with RT FIFO enabled	**B. Configure FIFO size statically using dialog boxes.**
RT FIFO Functions	**A. Configure FIFO size dynamically using block diagram code.**

Notes