NI cRIO-9951

CompactRIO™ Module Development Kit User Manual

Software User Manual

November 2017
375951B-01

NATIONAL INSTRUMENTS™

Worldwide Technical Support and Product Information
ni.com

Worldwide Offices
Visit ni.com/niglobal to access the branch office websites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

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

For further support information, refer to the *NI Services* appendix. To comment on NI documentation, refer to the NI website at ni.com/info and enter the Info Code feedback.

Legal Information

Limited Warranty

This document is provided ‘as is’ and is subject to being changed, without notice, in future editions. For the latest version, refer to `ni.com/manuals`. NI reviews this document carefully for technical accuracy; however, NI MAKES NO EXPRESS OR IMPLIED WARRANTIES AS TO THE ACCURACY OF THE INFORMATION CONTAINED HEREIN AND SHALL NOT BE LIABLE FOR ANY ERRORS.

NI warrants that its hardware products will be free of defects in materials and workmanship that cause the product to fail to substantially conform to the applicable NI published specifications for one (1) year from the date of invoice.

For a period of ninety (90) days from the date of invoice, NI warrants that (i) its software products will perform substantially in accordance with the applicable documentation provided with the software and (ii) the software media will be free from defects in materials and workmanship.

If NI receives notice of a defect or non-conformance during the applicable warranty period, NI will, in its discretion: (i) repair or replace the affected product, or (ii) refund the fees paid for the affected product. Repaired or replaced Hardware will be warranted for the remainder of the original warranty period or ninety (90) days, whichever is longer. If NI elects to repair or replace the product, NI may use new or refurbished parts or products that are equivalent to new in performance and reliability and are at least functionally equivalent to the original part or product.

You must obtain an RMA number from NI before returning any product to NI. NI reserves the right to charge a fee for examining and testing Hardware not covered by the Limited Warranty.

This Limited Warranty does not apply if the defect of the product resulted from improper or inadequate maintenance, installation, repair, or calibration (performed by a party other than NI); unauthorized modification; improper environment; use of an improper hardware or software key; improper use or operation outside of the specification for the product; improper voltages; accident, abuse, or neglect; or a hazard such as lightning, flood, or other act of nature.

THE REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND THE CUSTOMER’S SOLE REMEDIES, AND SHALL APPLY EVEN IF SUCH REMEDIES FAIL OF THEIR ESSENTIAL PURPOSE.

EXCEPT AS EXPRESSLY SET FORTH HEREIN, PRODUCTS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND AND NI DISCLAIMS ALL WARRANTIES, EXPRESSED OR IMPLIED, WITH RESPECT TO THE PRODUCTS, INCLUDING ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE OR NON-INFRINGEMENT, AND ANY WARRANTIES THAT MAY ARISE FROM USAGE OF TRADE OR COURSE OF DEALING. NI DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF OR THE RESULTS OF THE USE OF THE PRODUCTS IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NI DOES NOT WARRANT THAT THE OPERATION OF THE PRODUCTS WILL BE UNINTERRUPTED OR ERROR FREE.

In the event that you and NI have a separate signed written agreement with warranty terms covering the products, then the warranty terms in the separate agreement shall control.

Copyright

End-User License Agreements and Third-Party Legal Notices

You can find end-user license agreements (EULAs) and third-party legal notices in the following locations:

- Notices are located in the `<National Instruments>\_Legal Information` and `<National Instruments>` directories.
- EULAs are located in the `<National Instruments>\Shared\MDF\Legal\license` directory.
- Review `<National Instruments>\_Legal Information.txt` for information on including legal information in installers built with NI products.

U.S. Government Restricted Rights

If you are an agency, department, or other entity of the United States Government (“Government”), the use, duplication, reproduction, release, modification, disclosure or transfer of the technical data included in this manual is governed by the Restricted Rights provisions under Federal Acquisition Regulation 52.227-14 for civilian agencies and Defense Federal Acquisition Regulation Supplement Section 252.227-7014 and 252.227-7015 for military agencies.

Trademarks

Refer to the *NI Trademarks and Logo Guidelines* at `ni.com/trademarks` for more information on NI trademarks.

ARM, Keil, and µVision are trademarks or registered of ARM Ltd or its subsidiaries.

LEGO, the LEGO logo, WEDO, and MINDSTORMS are trademarks of the LEGO Group.

TETRIX by Pitsco is a trademark of Pitsco, Inc.

FIELDBUS FOUNDATION™ and FOUNDATION™ are trademarks of the Fieldbus Foundation.

EtherCAT® is a registered trademark of and licensed by Beckhoff Automation GmbH.

Patents

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

Export Compliance Information

Refer to the *Export Compliance Information* at `ni.com/legal/export-compliance` for the NI global trade compliance policy and how to obtain relevant HTS codes, ECCNs, and other import/export data.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

YOU ARE ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY AND RELIABILITY OF THE PRODUCTS WHENEVER THE PRODUCTS ARE INCORPORATED IN YOUR SYSTEM OR APPLICATION, INCLUDING THE APPROPRIATE DESIGN, PROCESS, AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

PRODUCTS ARE NOT DESIGNED, MANUFACTURED, OR TESTED FOR USE IN LIFE OR SAFETY CRITICAL SYSTEMS, HAZARDOUS ENVIRONMENTS OR ANY OTHER ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, INCLUDING IN THE OPERATION OF NUCLEAR FACILITIES; AIRCRAFT NAVIGATION; AIR TRAFFIC CONTROL SYSTEMS; LIFE SAVING OR LIFE SUSTAINING SYSTEMS OR SUCH OTHER MEDICAL DEVICES; OR ANY OTHER APPLICATION IN WHICH THE FAILURE OF THE PRODUCT OR SERVICE COULD LEAD TO DEATH, PERSONAL INJURY, SEVERE PROPERTY DAMAGE OR ENVIRONMENTAL HARM (COLLECTIVELY, “HIGH-RISK USES”). FURTHER, PRUDENT STEPS MUST BE TAKEN TO PROTECT AGAINST FAILURES, INCLUDING PROVIDING BACK-UP AND SHUT-DOWN MECHANISMS. NI EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS OF THE PRODUCTS OR SERVICES FOR HIGH-RISK USES.

Contents

Chapter 4
Internal Channels

Chapter 5
Development and Export Process

Chapter 6
Modules Support VIs

About This Manual

This manual contains information about using the CompactRIO Module Development Kit 2 software. Many of the concepts discussed in this manual are described in further detail in the *CompactRIO Module Development Kit Hardware User Manual*. Refer to the *CompactRIO Module Development Kit Hardware User Manual* for more information about the CompactRIO Module Development kit.

Required Software

The following software is required to use the CompactRIO Module Development Kit:

☐ LabVIEW 2017 SP1

☐ LabVIEW FPGA Module 2017

☐ LabVIEW Real-Time Module 2017

☐ CompactRIO Device Drivers 17.6

☐ CompactRIO Module Support 4.0.1

☐ CompactRIO Module Development Kit 2.1

Related Documentation

The following documents contain information that you may find helpful as you read this manual:

- *Getting Started with CompactRIO and LabVIEW*—Use this tutorial to learn how to develop a CompactRIO application in LabVIEW. While developing the application, you can learn concepts and techniques that you can apply when you develop your own CompactRIO application. You can download the latest version of this document from the NI Web site at `ni.com/manuals`.
- *CompactRIO Module Development Kit Hardware User Manual*—Use this manual to learn the mechanical and electrical requirements for developing a custom CompactRIO module.
- *LabVIEW Help*—Use the *LabVIEW Help* to access information about LabVIEW programming concepts, step-by-step instructions for using LabVIEW, and reference information about LabVIEW VIs, functions, palettes, menus, tools, properties, methods, events, dialog boxes, and so on. Access the *LabVIEW Help* by selecting **Help»Search the LabVIEW Help**. You also can navigate on the **Contents** tab to the FPGA Module help and the *C Series Reference and Procedures* help.

1

CompactRIO Module Development Kit 2 Software Overview

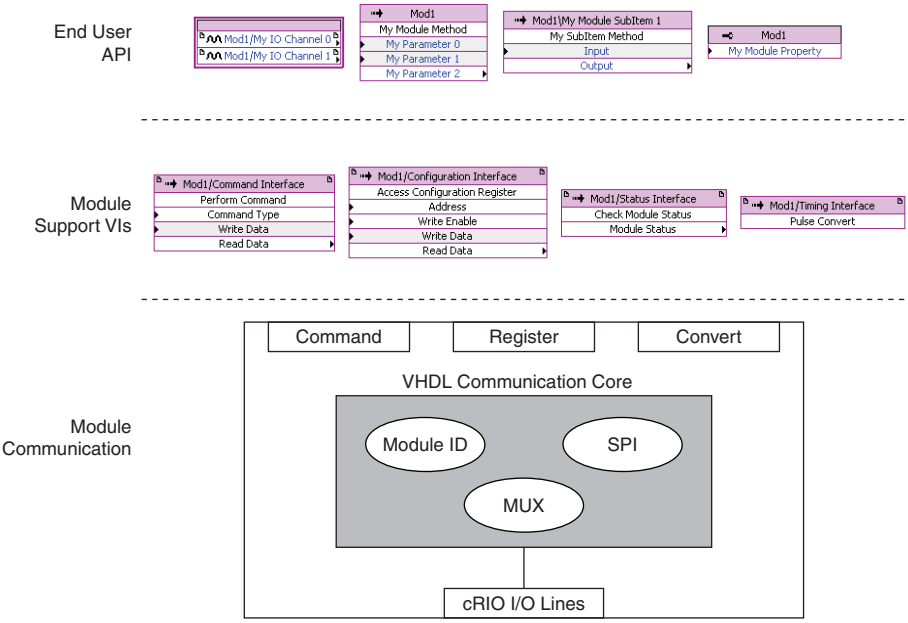
The CompactRIO Module Development Kit 2 (MDK 2) consists of two main software components.

The first component is the C Series Communication Core that communicates with the C Series module hardware. The C Series Communication Core is an IP block provided by National Instruments that exposes software interfaces that conform to the C Series architecture specification. Using the interfaces of the C Series Communication Core, you can write VIs that will communicate with your module.

The second component gives third party module developers the ability to script their own Module Support VIs beneath I/O, method, and property nodes. You will write VIs to communicate with your module hardware using the C Series Communication Core. The end user of your module will have an API composed of I/O, method, and property nodes just like an NI module. Because the Module Support VIs that you provide are scripted beneath the end user API nodes, your modules will look and behave like NI modules.

The end user API along with the Module Support VIs and the C Series Communication Core make up the software layers that are shown in Figure 1-1.

Figure 1-1. MDK 2 Software Layers



The end user API consists of I/O, method, and property nodes that provide the end user with an easy-to-use interface that is consistent with NI modules.

The Module Support VIs layer contains all the complex code that handles the operation of the C Series module. You will write the VIs that make up this layer.

The Module Communication layer contains the C Series Communication Core. The C Series Communication Core exposes the following interfaces through method nodes on the LabVIEW FPGA block diagram.

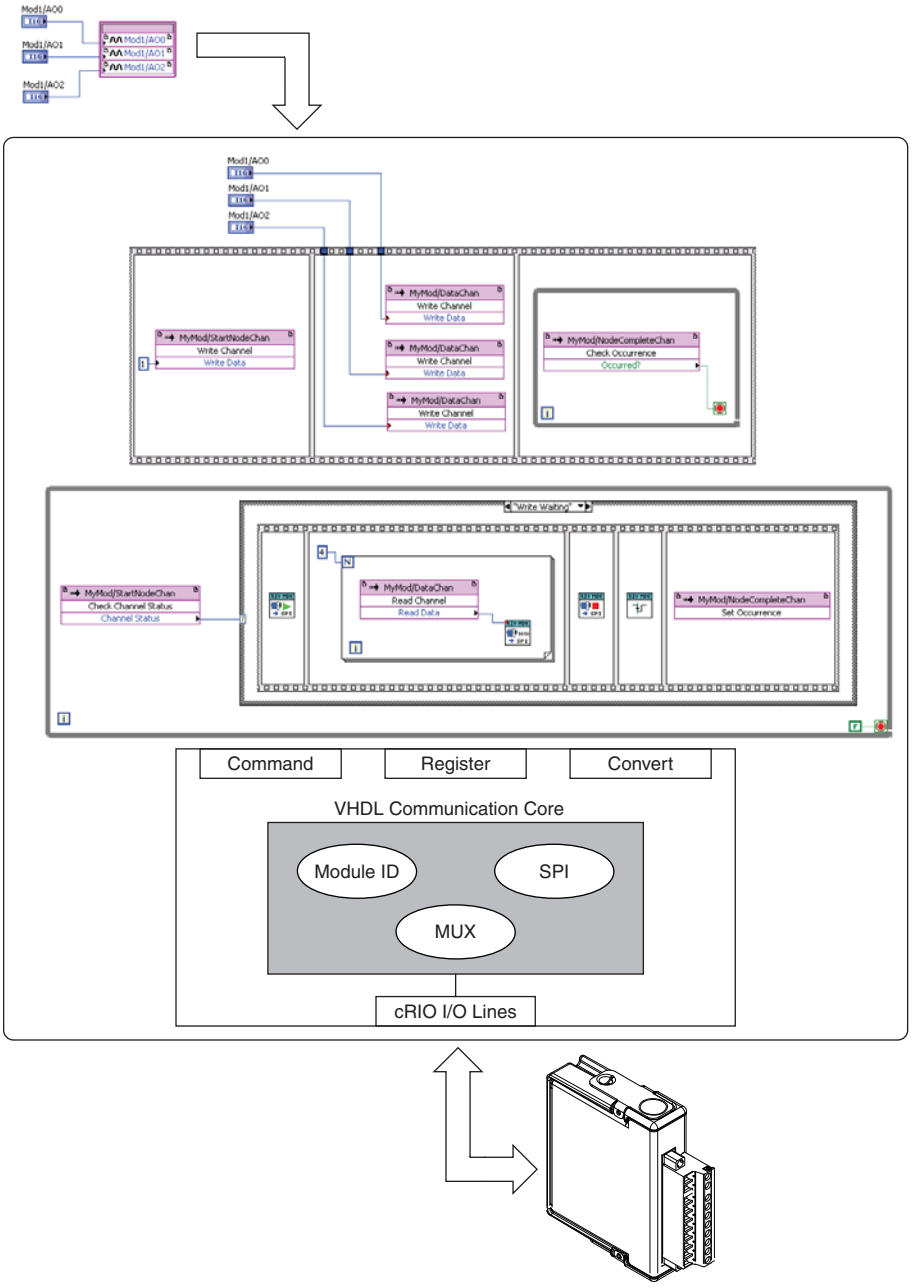
- SPI
- EEPROM
- Mode Change
- Module Identification
- Pulsing Convert
- Digital I/O

Additionally, there are two XML files you will write that allow your module to be identified by NI-RIO and added to an FPGA target. The XML files also specify how you want the Module Support VIs to be scripted beneath the end customer nodes.

Block Diagram

Figure 1-2 shows what happens when LabVIEW FPGA code is compiled. The third party module provides the end user with an API of I/O nodes (AO0, AO1, AO2). When the FPGA is compiled, the I/O nodes are replaced with Module Support VIs that the third party module developer provides. The Module Support VIs directly communicate with the module using the C Series Communication Core.

Figure 1-2. MDK 2 Implementation



Designing the Module API

The end user API of your module is defined in the Module Support XML. Like NI modules, third-party modules have I/O channels and module sub-items visible in the LabVIEW project. On the block diagram, your module will have I/O nodes, method nodes, and property nodes. Each of these API elements have a different purpose. NI recommends that you follow these guidelines when designing your API.

LabVIEW FPGA API Elements

LabVIEW FPGA provides several different types of API elements. Each of these API elements have a different purpose.

I/O Channel

Use I/O channels to represent the following physical channels on the module connector:

- Analog input
- Analog output
- Digital input
- Digital output

An I/O channel supports I/O nodes, property nodes, and method nodes.

Module Sub-Item

Use module sub-items to expose capabilities of the module that are not traditional analog or digital physical channels, such as serial ports. Module sub-resources support method nodes or property nodes.

I/O Node

Use I/O nodes as the primary mechanism for acquiring data from your module. I/O nodes are available for each of the I/O channels on the module and can be configured as read, write, or bi-directional.

Property Node

Use property nodes to read the module EEPROM and to write and read various runtime settings that the module or I/O channels may have. Running a property node can induce communication with the module or set a register in the FPGA and not perform any module communication. Property nodes can only have a single input or output and can be configured to read, write, or bi-directional.

Method Node

Use method nodes for module operations that I/O nodes and property nodes do not handle well. Any operation that has multiple inputs and/or outputs, or operations that gather data from the module should use a method node.

Recommended API Elements

You can define API elements that are useful for your module. NI recommends that you always support the following three property nodes:

- Vendor ID
- Module ID (Product ID in the EEPROM, not the module model code)
- Serial number

This allows end user applications to detect and identify any module that is in a slot configured for your module.

Creating Parallel DIO API Elements

You will write Module Support VIs that execute underneath your end user API elements. However, you may also specify parallel DIO API elements in your Module Support XML. These API elements do not use Module Support VIs beneath them. Instead, parallel DIO I/O nodes provide a direct connection between the end user block diagram, the C Series Communication Core and ultimately the FPGA pins.

Parallel DIO I/O nodes may be used both inside and outside of the Single Cycle Timed Loop (SCTL). DIO lines 0-7 of the cRIO bus may be used for parallel DIO I/O nodes.

Refer to the *Module Support XML* section of Appendix A, *Module XML*, for more information on the parallel DIO I/O nodes.

Development Mode versus Release Mode

You can use MDK 2 software in Development mode and Release mode.

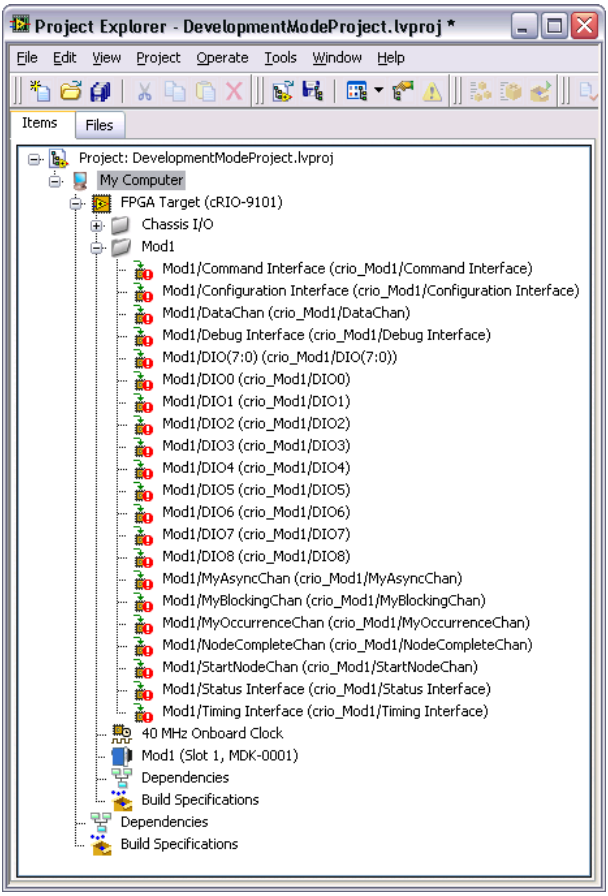
In Development mode, the interfaces to the C Series Communication Core are visible in the LabVIEW project. You can add these method nodes on the block diagram to communicate with your module. When creating or editing your Module Support VIs, you will be using the module in Development mode.

In Release mode, the interfaces to the C Series Communication Core are hidden in the LabVIEW project. Instead, the end user APIs that are defined in the XML are visible. Release mode is what end users use. The C Series Communication Core interfaces and your Module Support VIs are hidden from the end user; only the end customer API is visible in the LabVIEW project.

Refer to Chapter 5, *Development and Export Process*, for more information about creating module support in Development and Release modes.

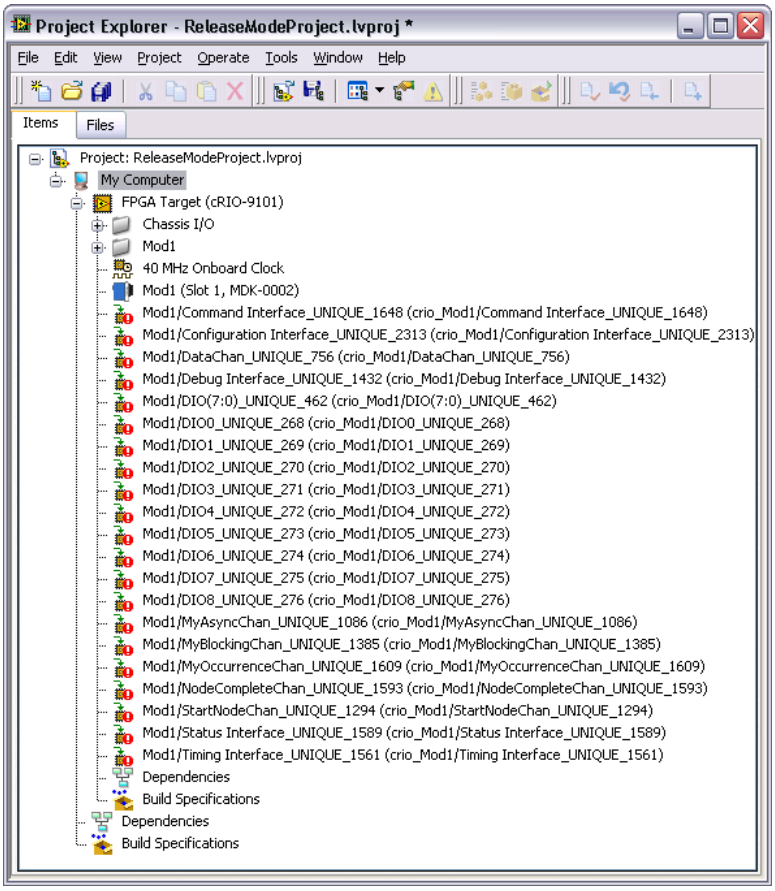
You will frequently switch between Development mode and Release mode if you are creating module support to deploy to end users. If you create a LabVIEW project that uses your module while the module is in Development mode, you can only use that project when the module is in Development mode. If you open that project when the module is in Release mode, there will be broken I/O items in the project as shown in Figure 1-3.

Figure 1-3. Opening a Development Mode Project in Release Mode



If you create a LabVIEW project that uses your module in Release mode, you can only use that project when the module is in Release mode. If you open that project when the module is in Development mode, there will be broken I/O items in the project as shown in Figure 1-4.

Figure 1-4. Opening a Release Mode Project in Development Mode



Use caution when creating and opening projects with third party modules on your development computer to avoid broken I/O items in the project.

MDK and NI-RIO Versions

MDK 2 installs documentation, LabVIEW examples, and API palettes. NI-RIO installs the files that allow you to create and run MDK 2 modules. Support for MDK 2 is included in CompactRIO Module Support 4.0.1, which requires NI-RIO 4.0. Versions of NI-RIO later than 4.0 will not require installing the separate CompactRIO Module Support patch.

Future versions of NI-RIO may include improvements to the MDK support and increase the MDK version number. If you update to a newer version of NI-RIO, check what version of MDK is installed with that version of NI-RIO using the `Mdk2Utility_GetInstalledMDKVersion.vi` utility located at `labview\vi.lib\LabVIEW Targets\FPGA\cRIO\shared\nicrio_Mdk2Utility`.

Note Go to `ni.com/info` and enter `criomdkupdate` for information about updates in the latest MDK support installed with NI-RIO.

Module support developed in a particular version of LabVIEW does not work in previous versions of LabVIEW, but generally works in later versions. However, updates to LabVIEW might prevent module support created in a previous version of LabVIEW from working. NI recommends that you test your module support in each new version of LabVIEW. The utility also displays the oldest compatible version of MDK.

2

Module XML Files

Note The XML files used to develop module support are case sensitive. Take care to ensure that all of your tags and values use the correct case.

The XML tags use a variety of data types. These data types are enforced by XML schemas and the rules checker. Refer to Table A-1, *XML Data Types*, in Appendix A, *Module XML*, for more information on these data types.

Some of the tags in the XML specify integer values. Depending on what is being specified, it may be useful to write the number in hexadecimal. To use a hexadecimal number, prefix `0x` to the number. For example, the decimal number `4243` can be specified as `0x1093`.

Module Type XML File

The Module Type XML file adds your module to the LabVIEW project and assists the module detection.

When a module is installed in a CompactRIO chassis, the C Series Communication Core automatically reads the module EEPROM and compares what is read against values that are pulled from the Module Type XML file.

Refer to the *Module Type XML* section of Appendix A, *Module XML*, for more information on the Module Type XML.

Module Support XML File

The Module Support XML file specifies the following:

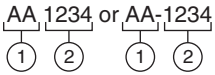
- the hardware functionality of your module
- the API of the module (I/O nodes, method nodes and property nodes)
- how the VI scripting tools connect the Module Support VIs during code generation

Refer to Appendix A, *Module XML*, and Appendix B, *Module Support XML Example*, for more information and an example of a Module Support XML file.

Module Name

The module name is specified in both the Module Type and Module Support XML files. The module name is used as an identifier when naming all of your module support files. Use the format shown in Figure 2-1 for your module name:

Figure 2-1. Model Name Format



1	Company Code	2	Module Model Code

The company code is a two or four letter character string to represent the name of your company. NI recommends that you use an acronym for your company code that it is consistent with NI modules, such as the NI 9263.

The module model code must match what is stored in the module EEPROM and Module Type XML.

All of your filenames must be unique. It is important that you select a module name that is unlikely to conflict with third party modules made by other vendors. All files, including XML and VIs, should be pre-pended with the module name. NI will advise you on a company code when you contact NI for a C Series vendor ID. If you already have a vendor ID, contact NI for assistance with your company code.

3

C Series Communication Core

The C Series Communication Core exposes the following interfaces of the C Series specification through the following I/O items.

- Command interface
- Timing interface
- Status interface
- DIO0 through DIO8 (digital lines)
- DIO (7:0) (digital port)
- Configuration interface
- Debug interface

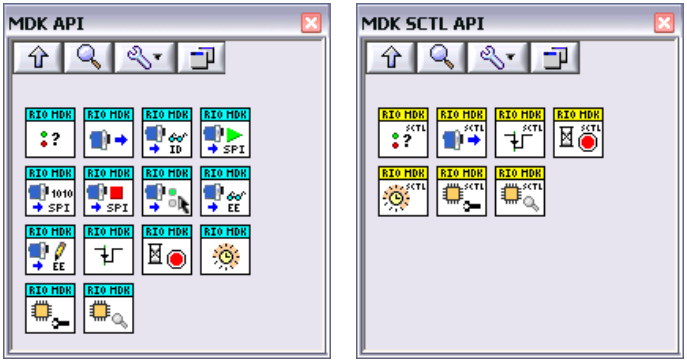
When in Development mode, each of these interfaces appear as I/O items in the LabVIEW project. The digital I/O (DIO) interfaces have I/O nodes defined for them and the rest of the interfaces use method nodes to perform operations.

Note These interfaces are exposed as I/O items on the module so that I/O references can be used on them. This is important when scripting the Module Support VIs beneath end user API nodes. Using I/O references allows you to create a single VI to use with your module in any chassis slot.

Using the C Series Communication Core

Use the MDK API and MDK SCTL API palettes, shown in Figure 3-1, to access the C Series Communication Core interfaces.

Figure 3-1. MDK API and MDK SCTL API Palettes



The MDK API palette contains VIs that execute outside of the Single-Cycle Timed Loop (SCTL). The MDK SCTL API palette contains VIs that execute within the SCTL. Use these VIs correctly inside and outside of the SCTL to prevent errors from occurring during code generation.

Adding the MDK API Palette

In order to use the MDK API, you must copy the palette files into your LabVIEW installation.

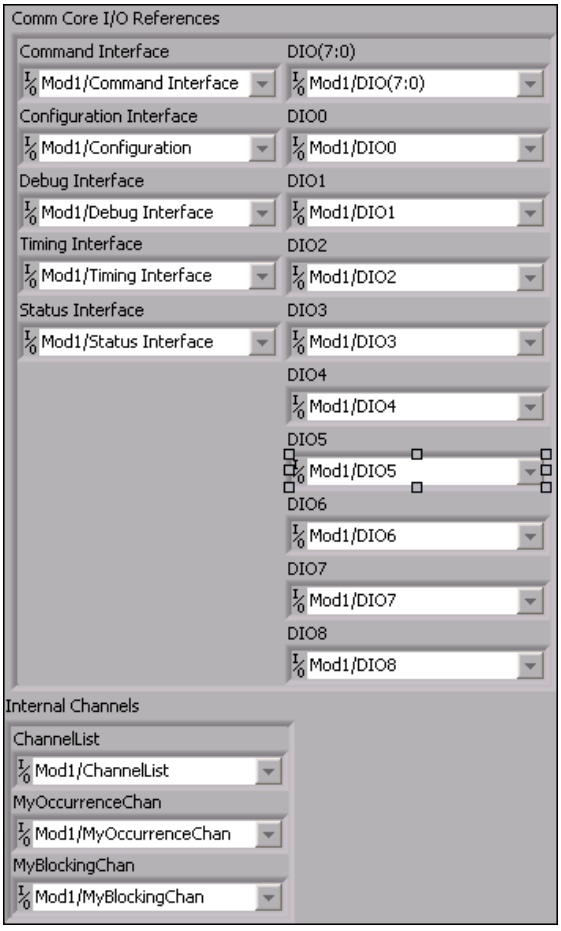
Copy the palette files from `C:\Program Files\National Instruments\CompactRIO\CompactRIO MDK 2\LabVIEW API Palette\_CrioMdk2Api` to `C:\Program Files\National Instruments\LabVIEW 2011\Targets\NI\FPGA\menus\FPGACategories\Programming\_CrioMdk2Api`.

Note You must restart LabVIEW after copying the palette files.

I/O References Cluster

Each of the API VIs contain an I/O References cluster input shown in Figure 3-2.

Figure 3-2. I/O References Cluster



When debugging or using the module in Development mode, use the I/O References cluster to set which module (chassis slot) the particular API VI executes on. When the module is used in Release mode, an I/O reference cluster configured for the correct slot is scripted onto the diagram at compile time.

C Series Communication Core MDK API

The following API VIs expose the interfaces of the C Series Communication Core.

Command Interface

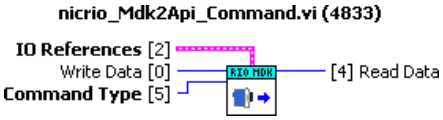
The Command interface exposes most of the C Series Communication Core functionality. Use the Command interface to perform the following operations:

- Module identification
- Mode change
- EEPROM read
- EEPROM write
- SPI

All of these operations go through the same interface because they share hardware resources. Only one of these operations may run at a time, which is enforced by exposing them through a single interface.

Command VI

The Command VI directly exposes the Perform Command method of the Command interface. It is a wrapper of the Perform Command Method Node.



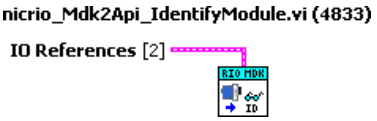
Perform Command Method Node

The Command interface exposes a single Perform Command Method Node.

The Perform Command Method Node exposes several types of operations. Use the Command Type terminal of the method node to select the operation.

Identify Module VI

The Identify Module VI is a wrapper around the Command interface that performs the Identify Module command.



Identify Module Command

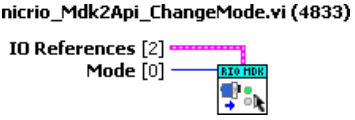
The Identify Module command reads start sentinel, vendor ID, and the module model code on the EEPROM of a module to determine if they are correct. Identify Module automatically runs when a chassis powers on or a new module is installed in the chassis.

The module must be in ID mode before running the Identify Module command. If the module is not in ID mode, this operation will fail.

The result of the Identify Module command may be read by the Module Status interface.

Change Mode VI

The Change Mode VI is a wrapper around the Command interface that performs the Change Mode command.



Change Mode Command

The Change Mode command changes the states of the ~ID_SELECT and FUNC lines that put the module in a different mode of operation. Refer to Table 3-1 for a list of Change Mode options. You must use the Change Mode command to place the module in a different mode of operation.

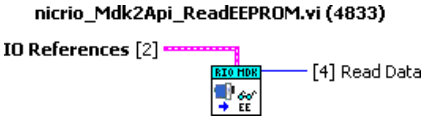
Table 3-1. Change Mode Options

Option	Mode of Operation
0	Idle (always enabled)
1	ID (always enabled)
2	Auxiliary Communication (enabled by Module Support XML)
3	Normal Operation (enabled by Module Support XML)

Note The Change Mode command cannot run at the same time as the Output Enable Method Node on the DIO lines. Running these two operations simultaneously results in a failure.

Read EEPROM VI

The Read EEPROM VI is a wrapper around the Command interface that performs the Read EEPROM command.



Read EEPROM Command

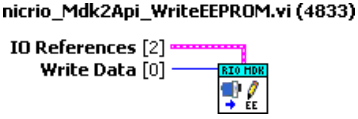
The Read EEPROM command performs a single byte EEPROM read on the module. The data returns through the Read Data terminal of the method node.

The module must be in ID mode before running the Read EEPROM command. If the module is not in ID mode, this operation will fail.

The EEPROM address that is being accessed is configured through the Configuration Register interface.

Write EEPROM VI

The Write EEPROM VI is a wrapper around the Command interface that performs the Write EEPROM command.



Write EEPROM Command

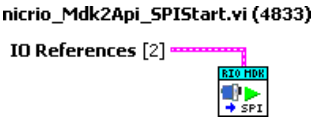
The Write EEPROM command performs a single byte EEPROM write on the module. Wire the data to be written to the Write Data terminal of the method node.

The module must be in ID mode before running the Write EEPROM command. If the module is not in ID mode, this operation will fail.

The EEPROM address that is being accessed is configured through the Configuration Register interface.

SPI Start VI

The SPI Start VI is a wrapper around the Command interface that performs the SPI Start command.



SPI Start Command

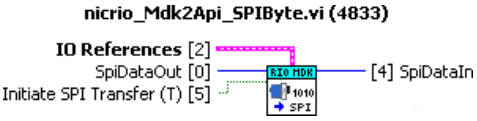
The SPI Start command begins an SPI transfer with the module. The module must be in a mode that supports SPI. ID mode always supports SPI. Normal Operation and Auxiliary Communication modes may support SPI if it is configured in the Module Support XML.

The SPI Start command asserts ~SPI_CS (drive it low).

You must run the SPI Start command before any other SPI commands are attempted.

SPI Byte VI

The SPI Byte VI is a wrapper around the Command interface that performs the SPI Byte command.



Use the Initiate SPI Transfer input to execute a new SPI byte transfer or wait for the last byte transfer to complete. Set the Initiate SPI Transfer input to TRUE to run the SPI Byte command. Set the Initiate SPI Transfer input to FALSE to run the SPI Wait command.

SPI Byte Command

The SPI Byte command initiates a single byte SPI transfer with the module. SPI Byte commands are pipelined. This means that the first time this command is called, the Write Data (*n*) goes out on the SPI bus and zeros are returned on the Read Data terminal. The second time this operation is called, the Write Data (*n*) will go out on the SPI bus and the previous SPI Byte Read Data (*n*-1) is returned.

You can only run the SPI Byte command once the SPI transfer has been started with the SPI Start command.

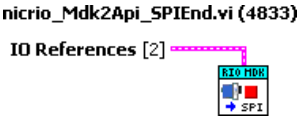
SPI Wait Command

The SPI Wait command waits until the previously run SPI byte has completed. The Write Data is ignored and the previously run SPI Byte read data is returned.

You can only run the SPI Wait command after starting the SPI transfer with the SPI Start command.

SPI End VI

The SPI End VI is a wrapper around the Command interface that performs the SPI End command.



SPI End Command

The SPI End command ends an SPI transfer with the module. The SPI End command de-asserts ~SPI_CS (drive it high).

You can only run the SPI End command after starting the SPI transfer with the SPI Start command.

Advanced Commands

The following commands combine the functionality of two commands.

SPI Byte & Start

The SPI Byte & Start command combines the functionality of the SPI Start and SPI Byte commands. As soon as ~SPI_CS is asserted, the SPI engine will begin transferring the first byte of the SPI transfer.

SPI Byte & End

The SPI Byte & End command combines the functionality of the SPI End and SPI Byte commands. When the SPI byte initiated by this command completes, the ~SPI_CS signal will de-assert and the SPI transfer will complete.

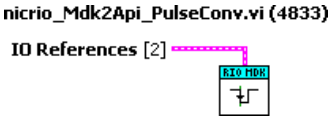
Because the SPI Byte commands are pipelined, the data returned from the SPI Byte & End command will be the second-to-last SPI Byte read data. You can run an SPI Wait command after this to retrieve the final SPI Byte read data.

Timing Interface

The Timing interface exposes portions of the C Series Communication Core that relate to timing the operation of the module.

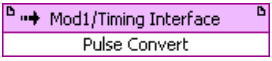
Pulse Convert VI

The Pulse Convert VI pulses the ~CONVERT line. It is a wrapper of the Pulse Convert Method Node.



Pulse Convert Method Node

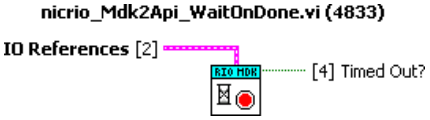
The Pulse Convert Method Node performs a single pulse on the ~CONVERT line of the CompactRIO bus. The pulse will be the width specified in the Module Support XML.



A fatal error occurs if the Pulse Convert Method Node is run with the module in a mode that does not support the Convert Pulse operation as defined in the Module Support XML. Only Normal Operation mode supports the Pulse Convert Method Node.

Wait on Done VI

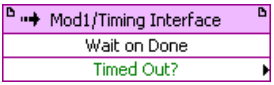
The Wait on Done VI waits until the ~DONE line is low. It is a wrapper of the Wait on Done Method Node.



Wait on Done Method Node

The Wait on Done Method Node waits until the ~DONE line is low. If the ~DONE line is already low when the Wait on Done Method Node is executed, it returns immediately. If the ~DONE line is not low before the timeout expires, the method node completes with a timeout.

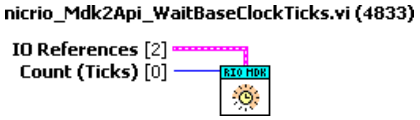
When executing, the Wait on Done Method Node will wait a few more clock ticks than specified in the XML before returning with a timeout. This is due to LabVIEW FPGA overhead when executing the method node.



A fatal error occurs if the Wait on Done Method Node is run with the module in a mode that does not support the Wait on Done operation as defined in the Module Support XML. Only Normal Operation mode supports the Wait on Done Method Node.

Wait Base Clock Ticks VI

The Wait Base Clock Ticks VI waits for the specified number of 25 ns base clock ticks. It is a wrapper of the Wait Base Clock Ticks Method Node.



Wait Base Clock Ticks Method Node

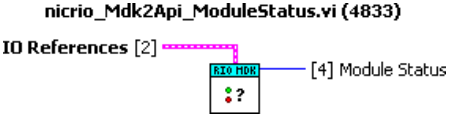
The Wait Base Clock Ticks Method Node waits for the specified number of 25 ns base clock ticks, which is useful because it always waits the same amount of time no matter what the top level clock is. This is necessary to get accurate timing of module operations since you do not have control over the top level clock that the end user uses.

Status Interface

The Status interface exposes portions of the C Series Communication Core that relate to the status of the module.

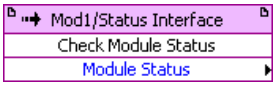
Module Status VI

The Module Status VI returns the module status of the C Series Communication Core. It is a wrapper of the Check Module Status Method Node.



Check Module Status Method

The Check Module Status Method Node returns the status of the module. This status is based on the presence of the module in the chassis and the result of the previously run Identify Module command.



Refer to Table 3-2 for a list of module statuses.

Table 3-2. Module Statuses

Status	Description
Unknown (0)	The chassis is powering up and the presence of the module has not yet been determined. The module status also briefly transitions to unknown during an Identify Module command.
Correct (1)	The module has been detected as present in the chassis and the EEPROM contents match the expected vendor ID and module model code.
Incorrect (2)	The module has been detected as present in the chassis and the EEPROM contents do not match the expected vendor ID and module model code.
No Module (3)	The module has not been detected as present in the chassis.
Invalid (4)	The module has been detected as present in the chassis and the EEPROM start sentinel does not match the expected value.
Incorrect Program Mode (5)	The slot is not configured for LabVIEW FPGA.

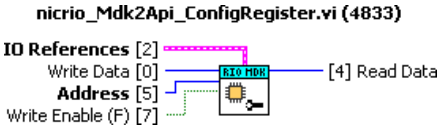
Note The Incorrect Program Mode status is only available in NI-RIO 17.6 and later.

Configuration Interface

The Configuration interface is used to set parameters on how the different C Series Communication Core interfaces operate.

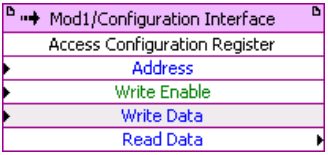
Configuration Register VI

The Configuration Register VI provides access to the Configuration register. It is a wrapper of the Access Configuration Register Method Node.



Access Configuration Register Method Node

The Access Configuration Register Method Node exposes the Configuration Register interface.



A fatal error occurs if an invalid address was written or read. Refer to Table 3-3 for the Configuration register map.

Table 3-3. Configuration Register Map

Address	Read/Write	Name	Function
0	R/W	EEPROM	This is the address of the EEPROM that will be accessed through the Command interface.

Debug Interface

Use the Debug interface for prototyping, debugging and manufacturing C Series modules. Do not use the Debug interface in Module Support VIs that ship to end users.

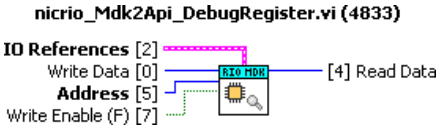
All of the Debug interface registers are available in Development mode. Some of the Debug interface registers are available in Release mode. Using a register in Release mode that is not available results in a fatal error.

Debug Register VI

The Debug Register VI provides access to the Debug register. It is a wrapper of the Access Debug Register Method Node.

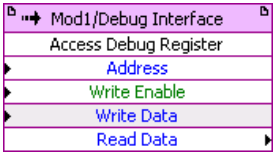
Note When writing the register, write 0 to reserved bits. When reading the register, reserved bits should be ignored.

The Debug register returns a U16, however data read from the internal error code register should be converted to an I16 before using it.



Access Debug Register Method Node

The Access Debug Register Method Node exposes the Debug Register interface.



A fatal error will occur if an invalid address was written or read. Refer to Table 3-4 for the Debug register map.

Table 3-4. Debug Register Map

Address	Read / Write	Available in Release Mode	Name	Function
0	Read Only	Yes	Internal Error Code	Returns the internal error code produced from a failed operation. Refer to Table 3-5 for a list of internal error codes.
1	R/W	No	SPI Rate Override Value	Write a 16-bit value to override the HalfTauTicks that were specified in the Module Support XML. The valid range for SPI HalfTau is 2 to 65535. The HalfTauTicks value is specified in 25 ns base clock ticks and the SPI clock frequency is automatically adjusted for different top level clock frequencies.
2	R/W	No	SPI Rate Override Enable	Write a 1 to bit 0 of this register to enable the SPI Rate Override value. Write a 0 to bit 0 of this register to revert to the HalfTauTicks specified in the Module Support XML.

Table 3-4. Debug Register Map (Continued)

Address	Read / Write	Available in Release Mode	Name	Function
3	R/W	No	Module Status Override Value	Write bits to set the desired module status.
4	R/W	No	Module Status Override Enable	Write a 1 to bit 0 of this register to override the module status of the C Series Communication Core. Write a 0 to bit 0 of this register to revert back to the actual module status of the C Series Communication Core.
5	Read Only	No	~ID_SELECT Status	Returns 1 if the ~ID_SELECT Line is high. Returns 0 if it is low.

SPI Rate Override Behavior

When you override the SPI rate value, it will only override the SPI rate as long as you remain in the same mode. When you change modes, that mode change will update the SPI rate register within the C Series Communication Core to the appropriate value for that mode. If you wish to re-enable overriding the SPI rate, you must write both the SPI rate override and SPI rate override enable registers.

Internal Error Code Behavior

A negative value returned from the Internal Error Code register indicates a fatal error. Fatal errors are unrecoverable and you must restart the FPGA VI to continue using the C Series Communication Core.

A positive value returned from the Internal Error Code register indicates a warning. Warnings are cleared when they are read from the Internal Error Code register and do not affect the functionality of the C Series Communication Core.

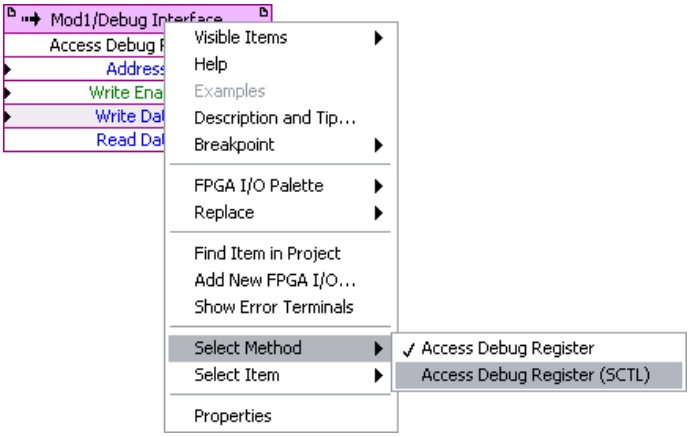
C Series Communication Core MDK SCTL API

The following API VIs expose the interfaces of the C Series Communication Core.

C Series Communication Interfaces (SCTL)

All of the C Series Communication Core interfaces can be accessed from both inside and outside of an SCTL. Most of these interfaces take multiple clock cycles to complete. These interfaces have two method nodes available on them. One may only be used outside of the SCTL and the other may only be used inside of the SCTL. Figure 3-3 shows the Access Debug Register Method Nodes.

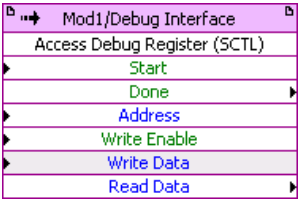
Figure 3-3. Selecting Method Nodes



The interfaces that work inside of the SCTL have the same functionality as the non-SCTL interfaces. However, since these API VIs execute within the SCTL, there are two additional terminals that are used for execution control. These additional control terminals are necessary because these operations take multiple clock cycles to complete.

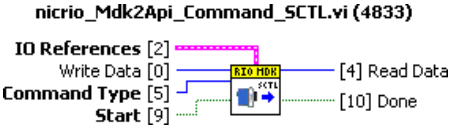
Use the Start terminal to start the operation. When the interface sees the Start terminal set to TRUE, the operation begins. Once started, the operation will complete regardless of whether or not the Start terminal is de-asserted. You may put a single cycle pulse on the Start terminal. On the clock cycle that the operation starts, the Done terminal will change to FALSE if it was previously true. Once the operation has completed, the Done terminal will change to true. The Done terminal remains true until the next operation is started. Figure 3-4 shows an SCTL Method Node.

Figure 3-4. SCTL Method Node



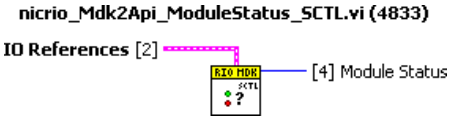
Command (SCTL) VI

The Command (SCTL) VI directly exposes the Perform Command method of the Command interface. It is a wrapper of the Perform Command (SCTL) Method Node.



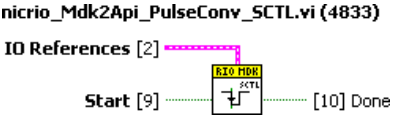
Module Status (SCTL) VI

The Module Status (SCTL) VI returns the module status of the Communication Core.



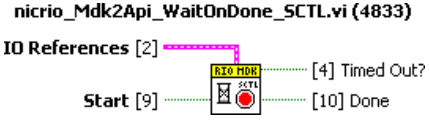
Pulse Convert (SCTL) VI

The Pulse Convert (SCTL) VI pulses the ~CONVERT line. It is a wrapper of the Pulse Convert (SCTL) Method Node.



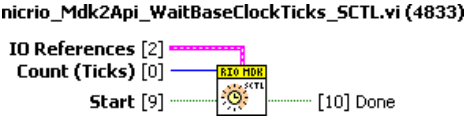
Wait on Done (SCTL) VI

The Wait on Done (SCTL) VI waits until the ~DONE line is low. It is a wrapper of the Wait on Done (SCTL) Method Node.



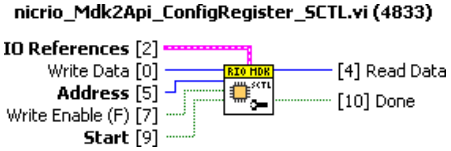
Wait Base Clock Ticks (SCTL) VI

The Wait Base Clock Ticks (SCTL) VI waits for the specified number of 25 ns base clock ticks. It is a wrapper of the Wait on Base Clock Ticks (SCTL) Method Node.



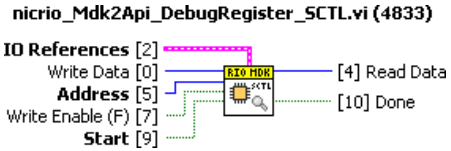
Configuration Register (SCTL) VI

The Configuration Register (SCTL) VI provides access to the Configuration register. It is a wrapper of the Access Configuration Register (SCTL) Method Node.



Debug Register (SCTL) VI

The Debug Register (SCTL) VI provides access to the Debug register. It is a wrapper of the Access Debug Register (SCTL) Method Node.



Using VIs Outside and Inside of the SCTL

The API VIs that are instantiated outside of the SCTL use normal LabVIEW arbitration between multiple instantiations of the same method node. If two method nodes try to access the same interface at the same time, the LabVIEW arbiter makes one method node wait until the other one completes.

The API VIs that are instantiated inside of the SCTL do not use arbitration. When the SCTL method nodes are placed on the LabVIEW block diagram, there is never any arbitration between the method node and the C Series Communication Core. This means that only *one* of each of these SCTL interfaces may be placed on the LabVIEW block diagram. This is why the MDK SCTL API palette does not have the same Command interface wrapper VIs that the MDK API palette has.

Digital I/O

The C Series Communication Core also provides eight DIO channels. These DIO channels do not have API VIs. You can directly instantiate the I/O and method nodes to access the DIO lines.

Digital Input and Output Interfaces

The C Series Communication Core exposes nine DIO lines. Lines 7:0 are also available in an 8-bit port.

Digital Input I/O Node

You can read the state of the DIO lines with the Digital Input I/O Node.

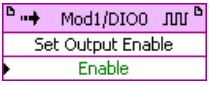
Right-click on the **I/O Node** and select **Properties** to configure the number of input synchronization registers for the I/O Node.

Digital Output I/O Node

You can write to the DIO lines using the Digital Output I/O Node.

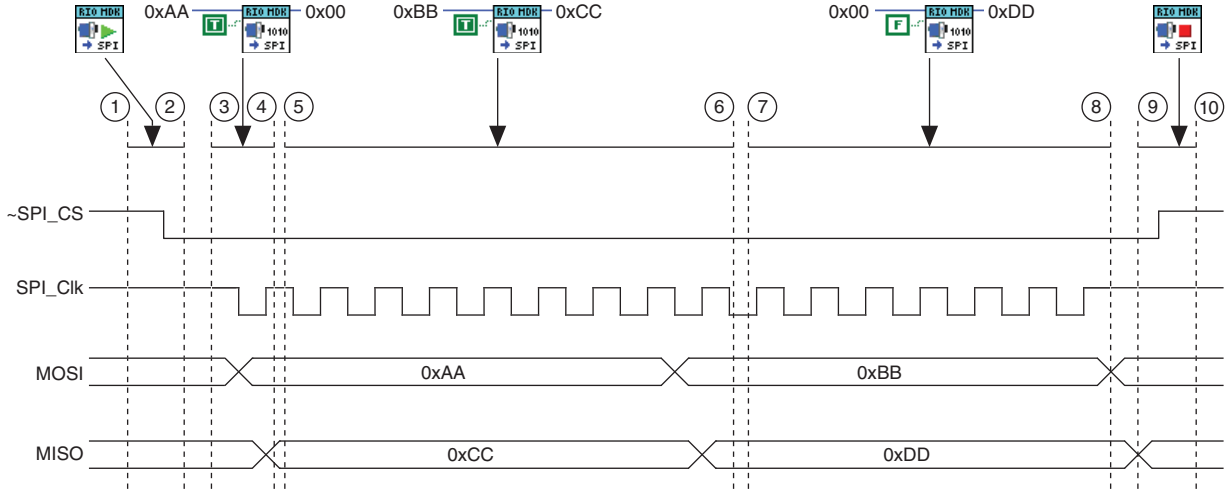
Set Output Enable Method Node

You can change the direction of a DIO line with the Set Output Enable Method Node.



Note The Change Mode command cannot run at the same time as the Output Enable Method Node on the DIO lines. Running these two operations simultaneously results in a failure.

Figure 3-5. SPI Clock Timing



1 The SPI Start VI begins its execution. This causes ~SPI_CS to assert.

2 The SPI Start VI completes.

3 The first execution of the SPI Byte VI begins. The SPI Write Data is 0xAA. The Initiate SPI Transfer is set to TRUE, which causes the SPI engine to start toggling SPI_CLK. The data 0xAA then goes out to the module on the MOSI line.

4 The SPI Byte VI completes the first execution almost immediately after it starts, because it was priming the pipeline of the SPI engine. The SPI Read data returned is 0x00 which should be ignored. The data read from the module on the first SPI byte will be returned by the next execution of the SPI Byte VI. At this point the SPI engine is SPIing the first byte.

5 SPI Byte VI begins its second execution. Its SPI Write Data is 0xBB. It has Initiate SPI Transfer set to TRUE which will cause the SPI engine to continue toggling SPI_CLK. The data 0xBB will go out to the module on the MOSI line.

6 SPI Byte VI completes its second execution after the first byte is SPIed to the module. The SPI Read data returned is 0xCC which was read from the module on MISO. At this point the SPI engine is SPIing the second byte.

7	SPI Byte VI begins its third execution. Its SPI Write Data is 0x00. It has Initiate SPI Transfer set to FALSE which will cause the SPI engine to not start another SPI byte. Because of this, the SPI Write Data is ignored in this execution. This VI will only wait until the last byte of SPI Read data is read by the FPGA before completing.
8	SPI Byte VI completes its third execution. The SPI Read data returned is 0xDD which was read from the module on MISO. At this point the SPI engine is not toggling SPI_CLK.
9	SPI Stop VI begins its execution. This causes ~SPI_CS to de-assert.
10	SPI End VI completes.

Using SPI_CLK as a Digital Line (DIO 8)

The SPI engine uses SPI_CLK (DIO 8) to perform module communication. To ensure correct mode transitions, SPI_CLK (DIO 8) must idle high when not in use, which the C Series Communication Core SPI engine handles.

You can use SPI_CLK (DIO 8) as a digital line. Observe the following requirements if you use the SPI_CLK (DIO 8) as a digital line.

- SPI_CLK must not be exposed to the end user as general purpose DIO
- SPI_CLK must idle high
- SPI_CLK must remain high during all mode transitions

NI does not recommend that you use SPI_CLK (DIO8) as a digital line in your module design. Refer to *CompactRIO Module Development Kit Hardware User Manual* for more information on the SPI_CLK (DIO 8).

Reserved Digital Lines

Most of the CompactRIO bus lines may be configured for DIO. However, depending on the mode and what subsystems of the C Series Communication Core are enabled, some CompactRIO bus lines may be reserved and not available for DIO.

~ID_SELECT

The ~ID_SELECT line is always reserved by the C Series Communication Core. DIO operations are not available on the ~ID_SELECT line. You may query the state of the ~ID_SELECT line by reading the ~ID_SELECT Status Debug register.

~CONVERT

The ~CONVERT line is reserved by the C Series Communication Core in Normal Operation mode when the `<ConvertPulseConfiguration>` section is specified in the Module Support XML. When the convert pulse functionality of the C Series Communication Core is enabled, ~CONVERT cannot be used as a DIO line in Normal Operation mode.

~DONE

The ~DONE line is reserved by the C Series Communication Core in Normal Operation mode when the `<DoneWaitConfiguration>` section is specified in the Module Support XML. When the done wait functionality of the C Series Communication Core is enabled, ~DONE cannot be used as a DIO line in Normal Operation mode.

~SPI_CS, SPI_CLK, MOSI, MISO

The four SPI lines are reserved by the C Series Communication Core in Normal Operation or Auxiliary Communication mode when the `<SPIConfiguration>` section is specified in the

Module Support XML for that mode. When the SPI functionality of the C Series Communication Core is enabled for a particular mode, the four SPI lines may not be used as DIO lines when in that mode.

SPI_FUNC

The SPI_FUNC line is reserved by the C Series Communication Core in Auxiliary Communication mode. The state of SPI_FUNC is always outputting low in Auxiliary Communication mode as defined by the *CompactRIO Module Development Kit Hardware User Manual*. The SPI_FUNC line may not be used as a DIO line in that mode.

Using the Wait Base Clock Ticks Method

When writing your Module Support VIs, you may have to hard code some waits to meet the timing requirements of your module. For example, you may include waits between pulsing ~CONVERT and starting SPI or after SPI to run a mode transition.

LabVIEW provides a Wait primitive that you can configure to wait in units of ticks or microseconds. When configured to wait for microseconds, the Wait primitive always waits the same amount of time regardless of what the FPGA top level clock is set to. When configured to wait on a number of ticks, the Wait primitive waits different amounts of time depending on the FPGA top level clock frequency.

If you need to wait a constant period of time that needs more resolution than an integer number of microseconds, you can use the Wait Base Clock Ticks method. This method node always waits the same number of 25 ns base clock ticks regardless of the FPGA top level clock frequency.

Module Status Behavior

The C Series Communication Core automatically detects and identifies C Series modules. When the FPGA powers up, the module status starts in the Unknown state. C Series modules may take up to two seconds to power-up. The C Series Communication Core waits up to two seconds while checking the ~ID_SELECT line to see if a module is present in the slot. If the ~ID_SELECT line is pulled up before the two seconds expire, the C Series Communication Core will put the module into ID mode to identify it. If the ~ID_SELECT line is not pulled up before the two seconds expire, the C Series Communication Core will determine that a module is not present and set the module status to No Module.

Some of the interfaces on the C Series Communication Core are blocked when the module status is Unknown. The interfaces are blocked because you may access them as soon as the FPGA VI starts. At that time, there may be a module in the slot that has not yet powered up. Those method nodes on the C Series Communication Core need to wait until the module is identified as Correct, Incorrect or No Module.

When a module is inserted into the chassis or when the Identify Module command is run, the module status becomes Unknown while the module is being identified.

The following interfaces are blocked while the module status is Unknown.

- SPI
- EEPROM
- Change Mode
- Identify Module
- Pulse Convert
- Wait on Done
- Output Enable methods on DIO lines

The following interfaces are not blocked while the module status is Unknown.

- Module Status
- Configuration Register
- Debug Register
- Wait for Base Clock Ticks
- I/O nodes on DIO lines

When the module status is No Module or Incorrect Program Mode, the C Series Communication Core goes into Idle mode and tri-states all of the CompactRIO bus lines. The Configuration and Debug registers may still be read or written to when the module status is No Module or Incorrect Program Mode. All other operations are ignored when the module status is No Module or Incorrect Program Mode.

Internal Errors

Incorrect or invalid sequences of operations on the C Series Communication Core results in a fatal error. This means that when an invalid operation is attempted, the C Series Communication Core goes into a state where it can no longer be used due to the fatal error. The C Series Communication Core will transition to the Idle mode and the module status will return No Module. All of the CompactRIO bus outputs to the module will be tri-stated.

Operations that can cause a fatal error include:

- Invalid command type
- Invalid Configuration or Debug register address
- Trying to SPI byte or SPI end when a SPI transfer has not yet been started
- Trying to SPI start when a SPI transfer has already been started
- Attempting to perform a mode change while a DIO Set Output Enable Method Node is running

- Trying to SPI, pulse ~CONVERT, access the EEPROM, or Wait on Done when in a mode that does not support those operations

You can read the Debug register to get an internal error code. This internal error code indicates what invalid operation caused the fatal error.

Refer to Table 3-5, for a full list of internal error codes.

Table 3-5. Internal Errors

Error	Error Code	Description
None	0	No error.
Change Mode (invalid mode)	-1	You attempted to change a mode that is not supported.
Identify Module (not in ID mode)	-2	You attempted to run the identify module commend when not in ID mode.
EEPROM Write (incorrect module)	-3	You attempted to run an EEPROM write command when the module was not correct.
SPI (not started)	-4	The SPI transfer has not been started.
SPI (already started)	-5	You attempted to start a SPI transfer when one had already been started.
SPI (not in correct mode)	-6	You attempted to start a SPI transfer when in a mode that does not support SPI.
Invalid Command	-7	You attempted to run a invalid command type.
Change Mode and Output Enable	-8	You attempted to run a mode change command when running an output enable method for DIO.
Invalid Configuration Register	-9	You attempted to access an invalid register (or did read/write on a register that did not support read/write).
Invalid Debug Register	-10	You attempted to access an invalid debug register (or did read/write on a register that did not support
Pulse Convert (not in correct mode)	-11	You attempted to pulse ~CONVERT in a mode that does not support pulse ~CONVERT.
Wait on Done (not in correct mode)	-12	You attempted to wait for done in a mode that does not support wait for done.

Table 3-5. Internal Errors (Continued)

Error	Error Code	Description
EEPROM Read or Write (not in ID mode)	-13	You attempted an EEPROM read/write when not in ID mode.
EEPROM Write Timeout	14	EEPROM write timeout.
SPI Divide Rate (accessed during SPI transfer)	-15	Attempted to write the SPI divide rate override register during a SPI transfer.

When the C Series Communication Core encounters a fatal error, it cannot recover. The module will be unusable until the FPGA is reset. This can be done by restarting the FPGA VI.

Module Mode Details

Supported Modes

All modules must support ID and Idle modes as described in the *CompactRIO Module Development Kit Hardware User Manual*. In addition to those modes, your module may optionally support Normal Operation and Auxiliary Communication modes. You enable these modes when you specify them in the Module Support XML.

Transitioning to modes that are not enabled results in a fatal error on the C Series Communication Core.

Mode Transitions

The C Series Communication Core handles all mode changes. All of the timing requirements that are mentioned in the *CompactRIO Module Development Kit Hardware User Manual* are met when the mode is changed.

Initially, DIO lines 0:7 are set to input, tri-stated, with the output value set to 0. When transitioning to a mode where a line is configured as a digital line, input, output, or bi-directional, the line will go to whatever values were previously set through the digital I/O nodes and Set Output Enable methods.

If a line is used by the SPI engine in one mode and is used as a DO in another mode, you can run the DO and Set Output Enable methods before changing the mode to prepare the line to go into the desired state when the mode change completes.

When the module is removed from the chassis, the Output Enable settings are cleared and all of the lines go back to their default state. For bi-directional DIO lines, you must re-run the Output Enable methods to reset the line directions when the module is installed in the chassis again.

4

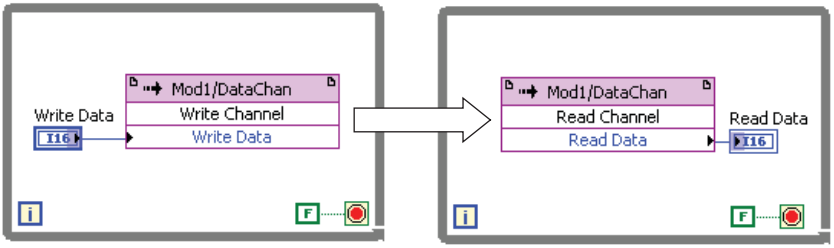
Internal Channels

Internal channels are data transfer mechanisms that allow concurrently running LabVIEW code to communicate.

You can access internal channels using method nodes on I/O channels. Each internal channel is represented in a LabVIEW project as an I/O channel on the module. This means that each module in a chassis has a unique set of internal channels.

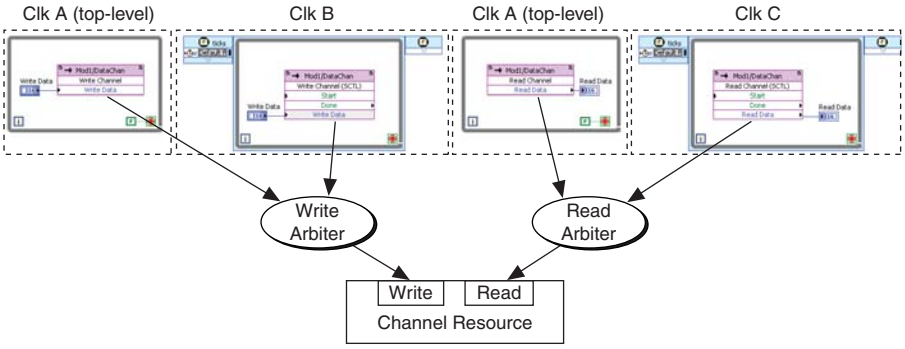
Figure 4-1 shows how you can use internal channels to communicate between concurrently running LabVIEW code.

Figure 4-1. Using Internal Channels



All of the methods on a particular channel access a shared resource that contains the data memory of the channel. The LabVIEW FPGA arbiters, shown in Figure 4-2, handle multiple accesses to the channel.

Figure 4-2. LabVIEW Arbiters Handling Multiple Accesses



You can use internal channels inside and outside of SCTLs. There is no restriction on how many nodes you can be place on a block diagram.

A module can use as many internal channels as required. However, NI recommends that you try to implement your module support with as few internal channels as possible to minimize FPGA utilization.

Internal Channel Types

MDK 2 provides three different types of internal channels. Each of these differ in how read and write nodes operate with respect to each other.

Asynchronous Internal Channel

When writing to an Asynchronous Internal Channel, the write will return as soon as the data reaches the flip-flops in the resource entity of the channel. When reading from an asynchronous channel, the read will return immediately with the most recent data from the flip-flops.

Use the Asynchronous Internal Channel when you have multiple readers or writers for the data and you do not need to synchronize the reads and writes. For example, use an Asynchronous Internal Channel for status or error information in the Module Resource VI. All of the node VIs can read the data on that channel before starting to check if the module is in an error condition.

Blocking Internal Channel

When writing to a Blocking Internal Channel, the write method node will not complete until the channel resource is read from a read method. If a read method node is called before the channel resource has been updated with new data, the Blocking Internal Channel will block until new data is written to the resource.

Use the Blocking Internal Channel when you want to synchronize parallel loops in LabVIEW. For example, use the Blocking Internal Channel to synchronize dataflow in the Module Resource VI and node VI.

Occurrence Internal Channel

The Occurrence Internal Channel has Send Occurrence and Check Occurrence methods. Sending an occurrence will set a flag in the channel resource. When you check for an occurrence, the Occurrence Internal Channel will tell you whether or not that flag has been set and clear it. Sending multiple occurrences without checking does not change the state of the flag, it just keeps setting it. These methods do not block and always return immediately.

Use the Occurrence Internal Channel when you do not have any data to send. For example, use the Occurrence Internal Channel to send simple Go or Done commands between the Module Resource and Node VIs.

Data Types

When writing the Module Support XML, you will specify the data type of each internal channel. MDK 2 supports both standard data types and custom controls. When using a custom control, it must be a Control; you cannot use Type Def or Strict Type Def.

If you change the data type of an internal channel, you must re-create any instantiations of method nodes for the internal channel on LabVIEW block diagrams.

5

Development and Export Process

You can use MDK 2 to develop module support for internal use or to ship to your customers. NI recommends you begin your module support development with the internal process to get familiar with MDK 2 before creating support for your customers.

Pre-pend all of your support files with the name of your module to prevent name collisions with other modules and LabVIEW VIs when your module support is loaded into LabVIEW.

Internal Support Development Process

Use the internal module support development process only if you need to communicate with your module internally and do not intend on deploying module support to your customers. The internal support development process does not script any Module Resource VIs or Module Support VIs beneath I/O nodes. It only allows you to use the C Series Communication Core interfaces in your LabVIEW FPGA block diagram.

When developing module support for internal use, all of the files will be created in the LabVIEW modules support folder:

```
C:\Program Files\National Instruments\LabVIEW 2011\Targets\NI\FPGA\
cRIO\other\MDK-1234
```

Note This document refers to the module in development as MDK-1234. In the following sections, use the name of your module.

Module Support Files

Create the following files for internal module support.

Module Type XML

Complete the following steps to create a Module Type XML file.

1. Name the XML file `MDK-1234_ModuleType.xml`
2. Create the following folder:

   `C:\Program Files\National Instruments\LabVIEW 2011\Targets\NI\FPGA\cRIO\other\MDK-1234\nicrio_configToolPlugin.llb`

   This folder is named so that the NI-RIO driver can find your Module Type XML file.
3. Place the XML file in the `nicrio_configToolPlugin.llb` folder.

Refer to the *Module Type XML* section of Appendix A, *Module XML*, for more information on creating the Module Type XML file.

Module Support XML

Complete the following steps to create a Module Support XML file.

1. Name the XML file `MDK-1234_ModuleSupport.xml`.
2. Place the XML file in the following folder:

   ```
   C:\Program Files\National Instruments\LabVIEW 2011\Targets\
   NI\FPGA\cRIO\other\MDK-1234
   ```

Note When creating the Module Support XML file, ensure that the `<DevelopmentMode>` tag is set to TRUE. This enables the C Series Communication Core and Internal Channel Method Nodes and disables the end user API I/O, method and property nodes.

Refer to the *Module Support XML* section of Appendix A, *Module XML*, for more information on creating the Module Support XML file.

Module Specific I/O References Control

Using a module specific I/O References Control is optional when creating internal module support. It is only needed if you are using internal channels. The module specific I/O References Control is automatically generated by the utility located in:

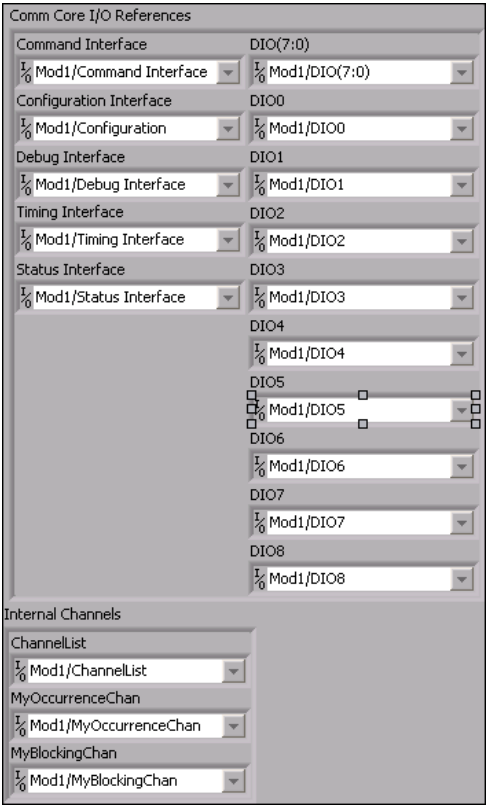
```
C:\Program Files\National Instruments\LabVIEW 2011\vi.lib\
LabVIEW Targets\FPGA\cRIO\shared\nicrio_Mdk2Utility\
Mdk2Utility_CreateIOReferenceClusterControl.vi
```

Run this VI and point it to the following folder:

```
C:\Program Files\National Instruments\LabVIEW 2011\Targets\NI\FPGA\
cRIO\other\MDK-1234
```

This creates the module specific I/O References cluster shown in Figure 5-1. The I/O References cluster contains two clusters of I/O references. The top cluster contains I/O references for the C Series Communication Core and the bottom cluster contains I/O references for internal channels. LabVIEW FPGA does not support empty clusters on the block diagram, so you must specify at least one internal channel in your XML. Having no internal channels results in an empty cluster and broken VIs.

Figure 5-1. Module Specific I/O Reference Cluster



Validating the Internal Module Support

Once the Module Support XML file is created, validate the XML by running the Module Support Export utility. On the front panel of the VI, select **Validate XML Only (No Export)** as the Export Type. This only performs the XML validation and does not create a module support export.

```
C:\Program Files\National Instruments\LabVIEW 2011\vi.lib\
LabVIEW Targets\FPGA\cRIO\shared\nicrio_Mdk2Utility\
Mdk2Utility_GenerateModuleSupportExport.vi
```

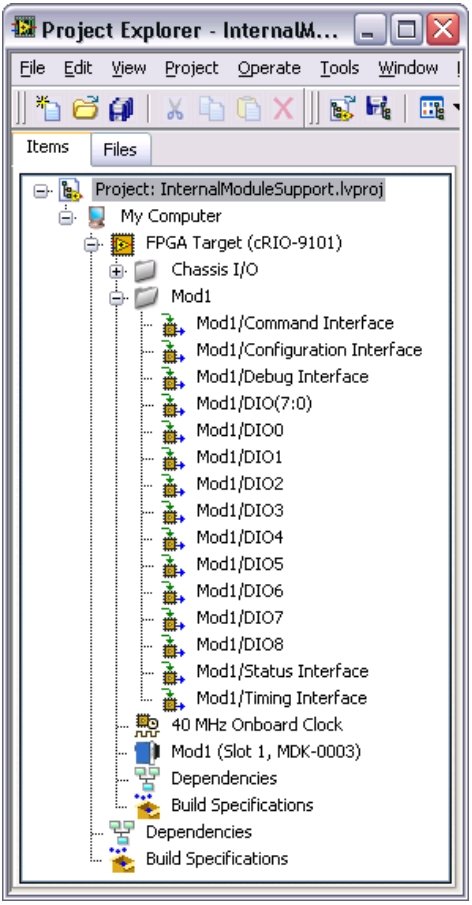
Using the Internal Module Support

You can now add your FPGA target to a LabVIEW project. LabVIEW will discover your module if it is connected. Complete the following steps to add your module to a LabVIEW project if the module is offline.

1. Right-click the FPGA Target in the Project Explorer window and select **New»C Series Modules** from the shortcut menu to display the Add Targets and Devices dialog box.
2. Click the **New target or device** radio button, select **C Series Module**, and click the **OK** button to display the New C Series Module dialog box.
3. Select your module from the **Module Type** pull-down menu and click the OK button.

After the module is added to the project, the C Series Communication Core and internal channel I/O items appear in the project. Figure 5-2 shows a LabVIEW project ready to begin writing your Module Support VIs using these I/O items.

Figure 5-2. Internal Module Support LabVIEW Project



Deployable Support Development Process

The deployable development process is similar to the internal development process, except that all of the module support files (XML and VIs) are created in a folder outside of the LabVIEW modules directory.

The directory where you create your module support files is called the development folder. This is a different folder than the LabVIEW modules support folder where the files are used by LabVIEW.

Export Utility

When the XML files are created, run the Module Support Export utility to export the support files into the LabVIEW modules support folder. It is important to use the utility because the export process manipulates some properties of the Module Support VIs and the utility modifies the files that get installed into the LabVIEW module support folder instead of the source files.

There are two different types of exports that you can perform.

- **Development mode export**—use to create the module support folder in LabVIEW that lets you create Module Support VIs.
- **Release mode export**—use to create the module support folder in LabVIEW that lets you test your module in release mode.

Set the Export Type control to perform the desired type of export.

When performing the export, the utility verifies that your XML files follow the required schema. It also validates your XML and Module Support VIs using a series of rules.

You do not need to run a separate utility to create the module specific I/O References cluster. The module specific I/O References cluster is automatically generated by the export utility.

The export utility is found in the following folder:

```
C:\Program Files\National Instruments\LabVIEW 2011\vi.lib\
LabVIEW Targets\FPGA\cRIO\shared\nicrio_Mdk2Utility\
Mdk2Utility_GenerateModuleSupportExport.vi
```

Note All development VIs and LabVIEW projects should be closed before running the export utility.

Exclude from Export

You may place a folder within your development directory called `ExcludeFromExport`. This folder and the contents in it will not be copied into the Module Support folder during the export. Use this folder to place the VIs and LabVIEW projects that were used to facilitate the development of your Module Support VIs. The Module Support VIs must be developed within the context of a LabVIEW project that contains your module in development mode. When developing your module support VIs, it is useful to have some test VIs that you can use to instantiate your Module Support VIs for debugging. NI recommends that you use the ExcludeFromExport folder to save these files.

Development Mode Export

After you create the two XML files, create a Development mode export to install the XML files into the LabVIEW hierarchy. This allows you to create your Module Support VIs.

Save your Module Support VIs in the Development folder as you create them.

The Export utility performs the following tasks in Development mode:

1. Loads the Module Type and Module Support XML files.
   a. Verifies that the XML passes the schema.
   b. Validates the XML using a series of rules.
2. Creates the I/O Reference Control based on the contents of the Module Support XML file.
3. Copies the module support files (XML and internal channel controls) to the export location in the LabVIEW modules directory.
4. Modifies the `<DevelopmentMode>` XML tag of the Module Support file to enable Development mode.

Do not modify or save any of the VIs in the Module Support folder. All changes will be lost when you run the export utility again. Also, be sure to not instantiate any controls or VIs from the LabVIEW Module Support folder in your Module Support VIs. All controls and indicators used in the Module Support VIs should be sourced from the development folder.

Your development folder will look something like this:

```
<MDK-1234>
- <nicrio_configToolPlugin.llb>
    - MDK-1234_ModuleType.xml
- MDK-1234_ModuleSupport.xml
- MDK-1234_IOReferences.ctl
- MDK-1234_ModuleResource.vi
- MDK-1234_MyInternalChannelControl.ctl
- MDK-1234_MethodNode.vi
- MDK-1234_IONode.vi
- MDK-1234_PropertyNode.vi
```

When you do a development mode export, your LabVIEW module support folder will look like this:

```
<MDK-1234>
- <nicrio_configToolPlugin.llb>
    - MDK-1234_ModuleType.xml
- MDK-1234_ModuleSupport.xml
- MDK-1234_MyInternalChannelControl.ctl
```

Notice that only the XML and internal channel control files are installed in LabVIEW. These files are necessary to have the module show up in LabVIEW and allow you to develop your Module Support VIs.

Release Mode Export

When you are ready to test the Release mode operation of your module, run the Export utility in Release mode. This copies all of the contents of the development folder to the LabVIEW module folder.

Once the export is completed, do not modify the files located in the LabVIEW modules support folder. A subsequent export will overwrite them and any changes will be lost. All file changes should be done in the development folder.

The Export utility performs the following tasks in Release mode:.

1. Loads the Module Type and Module Support XML files.
   a. Verifies that the XML passes the schema.
   b. Validates the XML using a series of rules.
2. Creates the I/O Reference Control based on the contents of the Module Support XML file.
3. Copies all of the module support files to the export location in the LabVIEW modules directory.
4. Removes `<DevelopmentMode>` XML tag of the Module Support file to enable Release mode.
5. Validates the VIs using a series of rules.
6. Adds a special tag to the exported VIs.
7. Locks and password protects the exported VIs with a password you provide.

Module Support VI Tagging

When using the module in Release mode, the Module Support VIs are scripted during code generation. It is important that the VIs follow the requirements of MDK 2 and pass a series of rules that verify this. Once the VIs are verified, they get a special tag placed using a hidden VI property. If the VI does not pass any of the rules, the tag is not be placed on the VI.

During code generation, NI-RIO verifies that all of the Module Support VIs have the special tag. If any of the VIs do not have the tag, then the compile stops and the user gets a code generation error. This is how MDK 2 prevents LabVIEW from attempting to compile invalid Module Support VIs.

During the export process, the Module Support VIs are locked and password protected after the tag is added. If the module is unlocked and saved, the tag is be removed. This means that the module support will no longer compile in LabVIEW FPGA.

Using the Deployable Module Support

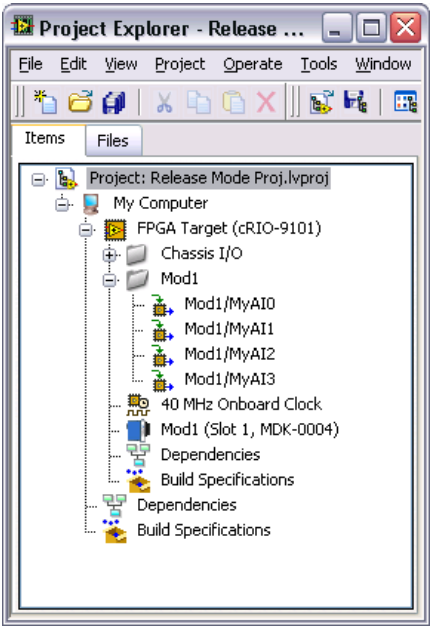
You can now add your FPGA target to a LabVIEW project. LabVIEW will discover your module if it is connected. Complete the following steps to add your module to a LabVIEW project if the module is offline.

1. Right-click the FPGA Target in the Project Explorer window and select **New»C Series Modules** from the shortcut menu to display the Add Targets and Devices dialog box.
2. Click the **New target or device** radio button, select **C Series Module**, and click the **OK** button to display the New C Series Module dialog box.
3. Select your module from the **Module Type** pull-down menu and click the OK button.

If you are doing a Development mode export, you should be able to add the module to a LabVIEW project and see the C Series Communication Core interface I/O and internal channel I/O items added to the project. Refer to Figure 5-3 for an image of a LabVIEW project with interface I/O and internal channel I/O items in the project. You are now ready to develop Module Support VIs.

If you are doing a Release mode export, you should be able to add the module to a LabVIEW project and see that the end user I/O items added to the project as shown in Figure 5-3. You are now ready to test the module in Release mode.

Figure 5-3. Release Mode Export



Shipping the Deployable Module Support

The module support folder created by the Release mode export is saved in the following LabVIEW folder:

```
C:\Program Files\National Instruments\LabVIEW 2011\Targets\NI\FPGA\
cRIO\other\MDK-1234
```

This is the folder that you will deploy to your customers and have them install onto their computers. This module support folder can be used on any computer that has the appropriate LabVIEW and NI-RIO distributions installed.

6

Modules Support VIs

Module Resource VIs and node VIs are the two types of Module Support VIs in MDK 2.

Module Resource VIs are scripted once in the FPGA block diagram for each module in the chassis and provide a place for you to put all of the code that communicates with your module.

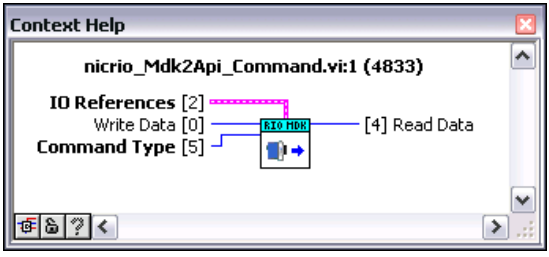
Node VIs are scripted beneath the I/O, method, and property nodes and communicate data between the end user block diagram and the Module Resource VI. In general, you should not place the C Series Communication Core API elements within node VIs. Place your complex code in the Module Resource VI since it is instantiated only once in the FPGA. The end user can place I/O, method and property nodes on the block diagram, which duplicates the logic that is used to communicate with the module if C Series Communication Core API elements are placed within the node VI.

Use the internal channels to communicate between the node VIs and the Module Resource VI.

Viewing Terminal Numbers in the Context Help

Figure 6-1 shows the VI terminal numbers in the Context Help window. It is important to enable terminal number viewing in the Context Help window to ensure that all of the terminals on Module Support VIs have the correct placement on the VI connector pane.

Figure 6-1. VI Terminal Numbers in the Context Help



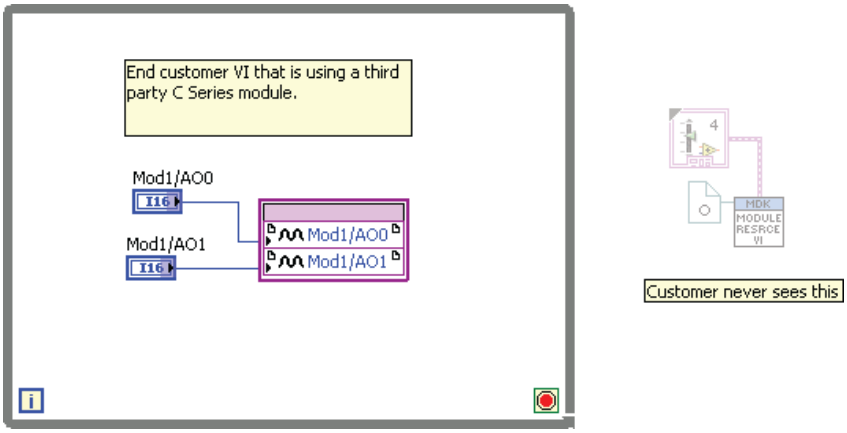
Complete the following steps to enable terminal number viewing in the context help.

1. Go to **Tools»Options**.
2. Select VI Server in the **Category** list in the **Options** dialog box.
3. Check **Show VI Scripting functions, properties, and methods** and verify that **Display additional VI Scripting information in Context Help window** contains a check in the **VI Scripting** section.

Module Resource VI

The Module Resource VI handles the execution of various API elements like the I/O, method, and property nodes. When the end user compiles the FPGA block diagram, the Module Resource VI is scripted into the FPGA. The Module Resource VI is never visible to the end user as shown in Figure 6-2.

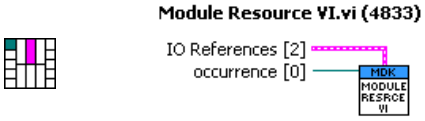
Figure 6-2. Hidden Module Resource VI



Module Resource VI Connector Pane

The Module Resource VI must use a 5-3-3-5 connector pane, which must have the Terminal 0 and Terminal 2 connections shown in Figure 6-3.

Figure 6-3. Occurrence Reference and I/O References Cluster

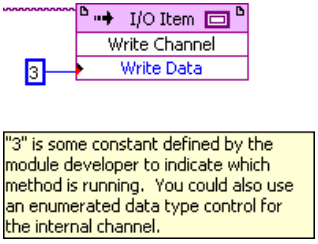


Handling API Element Operations

The Module Resource VI handles the operation of one API element at a time. If two API elements attempt to run simultaneously, you must use a Blocking Internal Channel to make one API element wait while the other is handled.

An API element first performs an internal channel write (blocking channel), as shown in Figure 6-4.

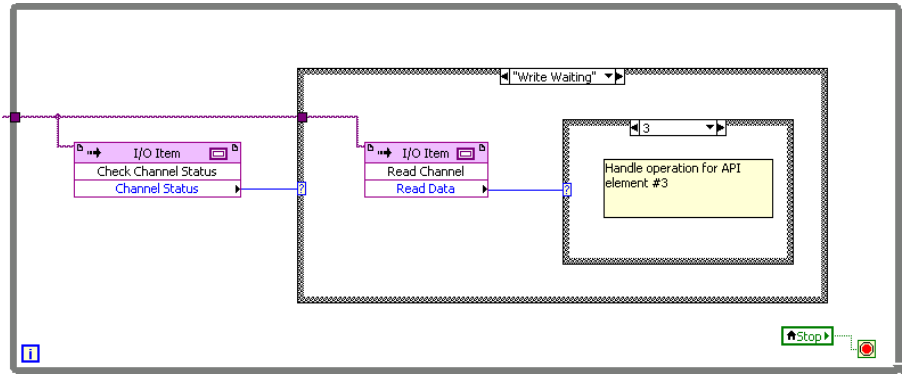
Figure 6-4. Internal Channel Write



The value written to the internal channel indicates which type of API element is running. Because this is a blocking internal channel, the method node waits until the channel is read before completing.

The Module Resource VI checks the status of that blocking internal channel. When the Module Resource VI sees that the channel has a write waiting, it reads the data from the channel. Depending on the value read from the channel, the Module Resource VI performs different operations specific to the API element that is running. Figure 6-5 shows the Module Resource VI performing these tasks.

Figure 6-5. Module Resource VI



Because this is a Blocking Internal Channel, any other API elements that attempt to run are blocked while this operation is being handled. When the Module Resource VI completes the operation of the API element, the loop goes to the next iteration and checks the channel status. At that time, if another API element is attempting to run, it will be handled.

Stopping the Module Resource VI

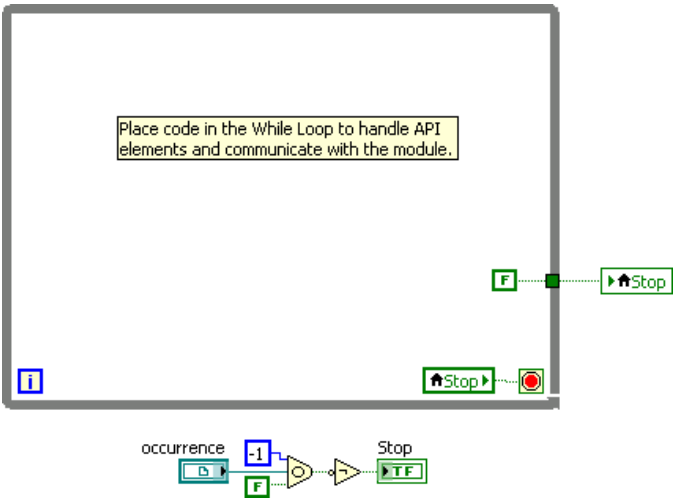
The Module Resource VI is scripted into the end user's top level VI during code generation. When the top level VI completes its execution, the Module Resource VI must also stop. Use an occurrence to stop the Module Resource VI.

The Module Resource VIs connector pane must include an occurrence reference. When the top level VI is finished, the occurrence is set. You must stop the Module Resource VI from running when the occurrence is set.

Note Use the standard LV FPGA Occurrence primitive to stop the Module Resource VI. *Do not* use the internal channel occurrence.

Figure 6-6 shows the LabVIEW code for stopping the Module Resource VI using an occurrence.

Figure 6-6. LabVIEW Code



When the Module Resource VI starts to execute, the Wait on Occurrence primitive waits until the occurrence is set. The timeout is set to `-1`, which means wait forever and the Ignore Previous input is set to FALSE. When the top level VI completes, the occurrence is set and the Wait on Occurrence primitive completes with no timeout (FALSE). This sets the Stop indicator to TRUE. A local variable on the Stop indicator stops the While Loop.

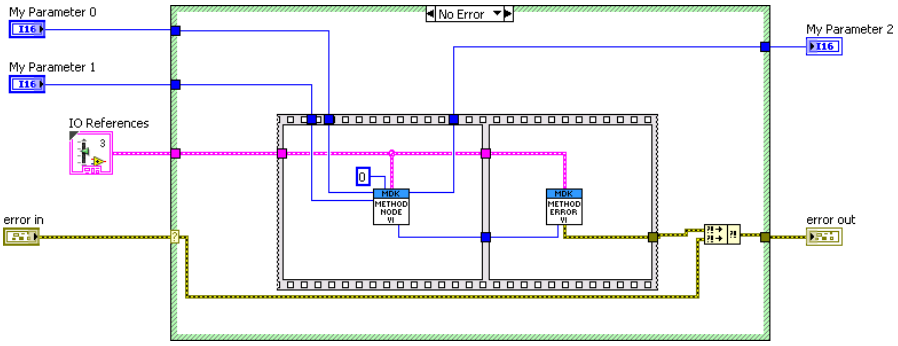
It is critical that you stop the Module Resource VI when the occurrence is set. Failure to do so will result in the end user's top level VI hanging when its execution completes. NI recommends that the loop rate of the Module Resource VIs While Loop be no longer than 500 ms, because it may take up to one full loop iteration before the loop will stop. You do not want to introduce a noticeable lag between completion of the end user's top level VI and when the Module Resource VI stops executing.

Node VIs

When the LabVIEW FPGA block diagram is compiled, the API elements of the module get replaced with the node VIs that you provide.

The method node in Figure 6-7 has its error terminals on, which cause the outer case structure to be scripted. If the Module Support XML specifies that this method node has an error handling VI, the VI would get scripted. This module support error handling VI is optional. If it is not present, then the error wire simply passes through the node and the Module Support VIs will not be able to produce any error codes on the error wire of the node.

Figure 6-7. Method Node



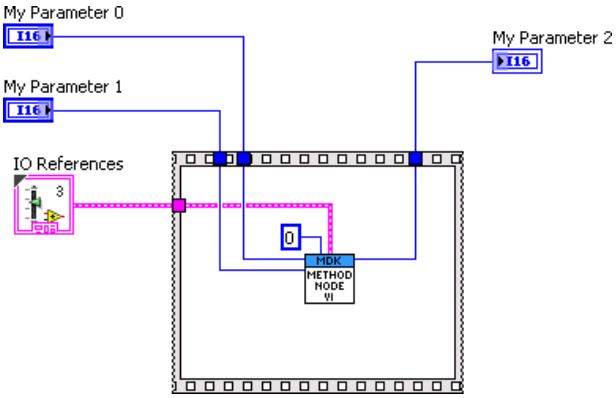
The Module Support VIs can only produce an error code. The error cluster that comes out of the node VI is merged with the incoming error. Third party Module Support VIs do not have access to the incoming error. The code executes if no error is passed in, or does not execute if an error is passed in. Any outputs of the node are zero or false when an error is passed into it.

Each Module Support VI shown in Figure 6-7 is scripted into a separate frame in the sequence structure. This order is determined by the `<SequenceOrder>` tag from the Module Support XML.

Data can transfer from one frame to the next using the Instance Data wire. Use the `<UseInstanceData>` tag in the Module Support XML to enable the Instance Data wire. When Instance Data is enabled for the method node, all VIs that are scripted must support it.

If the end user does not have enable error handling on the method node as shown in Figure 6-8, the error handling portion does not get scripted.

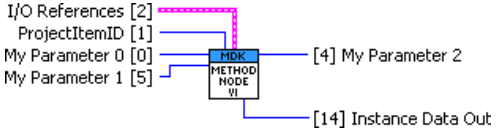
Figure 6-8. Method Node without Error Handling



Method and Property Node VIs

Only one method and property node VI is permitted in addition to the optional Error Handling VI.

Method and Property Node VI Terminals



The method node VI has three required terminals:

- **Terminal 1**—ProjectItemID
- **Terminal 2**—I/O References
- **Terminal 14**—Instance Data Out is only required when the `<UseInstanceData>` tag is set to TRUE and must match Instance Data In of the next VI in the sequence

On method node VIs, terminals 0, 5, 7, 9, 11 are write parameters and terminals 4, 6, 8, 10, 15 are read parameters.

The property node VI only permits the following parameter terminals:

- **Terminal 0**—Write Property Node Parameter
- **Terminal 4**—Read Property Node Parameter

Inside Method and Property Node VIs

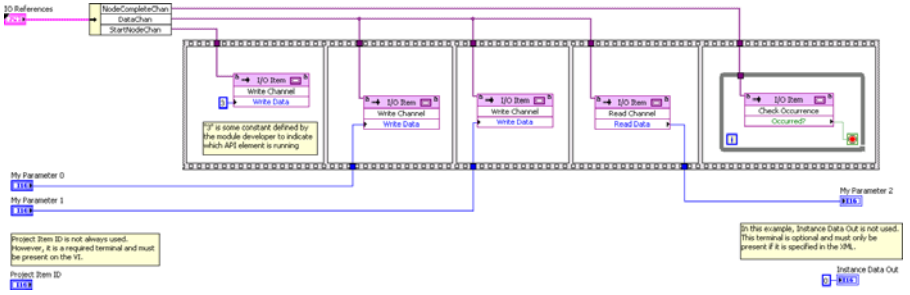
The first thing a node VI should do is use a blocking internal channel to tell the Module Resource VI that a node VI is attempting to run. The data should indicate what type of node VI it is.

After the Module Resource VI reads the channel data, the node and Module Resource VIs are free to communicate with each other using internal channels. You may use any type of internal channel within the node VI to communicate with the Module Resource VI. In general, all communication with the C Series Communication Core and all data processing should be done in the Module Resource VI.

Once the operation has completed, you may use an Occurrence internal channel to make the node VI wait until the Module Resource VI is done with the operation and ready to allow another operation to begin.

The use of the Blocking Internal Channel at the start of the node VI and the occurrence internal channel at the end of the node VI, shown in Figure 6-9, creates a request/release protocol. When the node VI begins, it will request the Module Resource VI. The node VI will wait until the Module Resource VI releases it. Using this protocol, you can ensure that only one node VI will execute at a time.

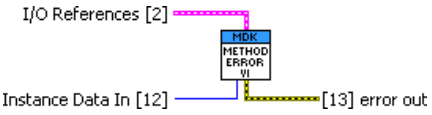
Figure 6-9. Internal Channels



Error Handling VI

The Error Handling VI can be optionally specified in the Module Support XML. When specified, the Error Handling VI will be scripted into the diagram if error terminals are enabled on the end user API node. The purpose of the Error Handling VI is to let API elements produce error codes that will be received by the end user.

Error Handling VI Terminals



The Error Handling VI has two required terminals:

- **Terminal 2**—I/O References cluster
- **Terminal 13**—error out

The Instance Data In terminal is required when the `<UseInstanceData>` tag is set to TRUE:

- **Terminal 12**—Instance Data In (must match Instance Data Out of the previous sequence VI)

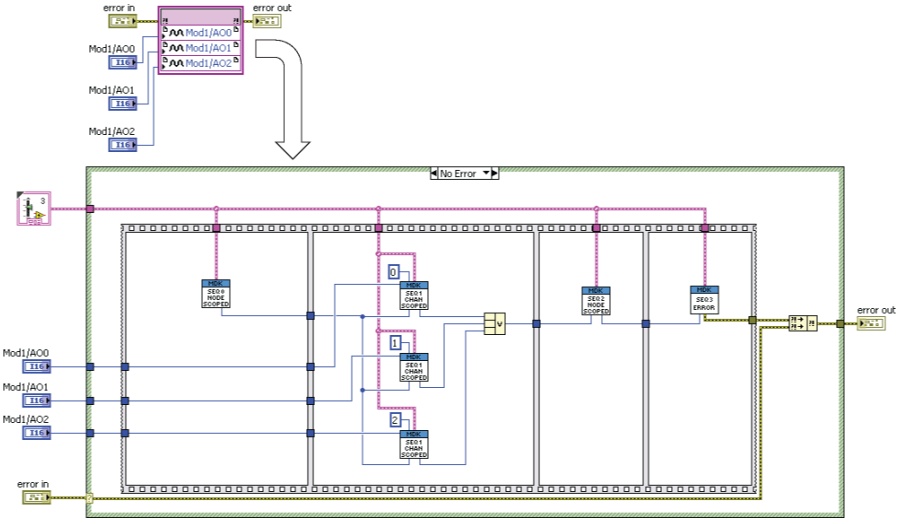
I/O Node VIs

I/O nodes differ from method and property nodes. I/O nodes are growable. This means that you can expand an I/O node to contain many I/O items. All of the I/O items in a grown node are expected to execute atomically. Refer to *Handling API Element Operations* for more information on the differences between I/O, method, and property nodes and when to use them.

Because a single grown I/O node must execute all of its I/O items at the same time, we need a more advanced scripting capability.

Figure 6-10 shows three different Node VIs, in addition to the Error Handling VI, being scripted into the grown I/O node. The first and third VIs are node scoped. The second VI is channel scoped. When specifying the VIs in the Module Support XML, you can use the `<VIScope>` tag to specify which type they are.

Figure 6-10. Node VIs Being Scripted into the Grown I/O Node



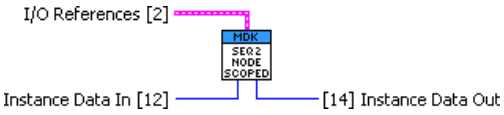
The node scoped VIs are only instantiated once for the grown I/O node. The channel scoped VIs are instantiated for every I/O item in the grown node.

When using these VIs in sequence, it is possible to make all of the I/O items in the grown node act atomically. For example, the above VI is for an analog output module. When it runs, all three channels are updated simultaneously. Here is how it works:

1. **Seq 0 (node scoped)**—Uses the blocking channel to tell the Module Resource VI that an I/O node is going to run.
2. **Seq 1 (channel scoped)**—Sends the output data to the Module Resource VI. The ProjectItemID is used to identify that the data for the AO channel is being written.
3. **Seq 2 (node scoped)**—Tells the Module Resource VI to SPI the AO data to the module and pulse ~CONVERT. This makes all three AO channels update simultaneously.

An I/O node may have multiple node and channel scoped levels. However, one of the channel scoped levels must have the `<VIHasTerminalConnections>` tag set to TRUE in the Module Support XML. That particular VI will have the data terminal of the I/O node wired to it. Other channel scoped VIs will not have any connections to the I/O node terminals.

Node Scoped I/O Node VI Terminals



The Node Scoped I/O Node VI has one required terminal:

- **Terminal 2**—I/O References cluster

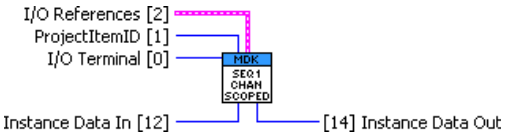
The Node Scoped I/O Node VI has required terminals if instance data is enabled:

- **Terminal 12**—Instance Data In (must match Instance Data Out of previous sequence)
- **Terminal 14**—Instance Data Out (must match Instance Data In of next sequence)

Note Instance Data In is *not* allowed on the first VI in the sequence because there is no sequence VI before it, so there is nothing to wire to it.

Note Node scoped VIs do *not* have the ProjectItemID. This is because the grown I/O node will contain many I/O items, each with a different ProjectItemID.

Channel Scoped I/O Node VI Terminals



The Channel Scoped I/O Node VI has two required terminals:

- **Terminal 1**—ProjectItemID
- **Terminal 2**—I/O References cluster

The Channel Scoped I/O Node VI has required terminals if instance data is enabled:

- **Terminal 12**—Instance Data In (must match Instance Data Out of previous sequence)
- **Terminal 14**—Instance Data Out (must match Instance Data In of next sequence)

Note Instance Data In is *not* allowed on the first VI in the sequence because there is no sequence VI before it, so there is nothing to wire to it.

The Channel Scoped I/O Node VI has one required terminal if it is going to connect to the node terminal:

- **Terminal 0**—Write I/O Node parameter (to I/O node)

or

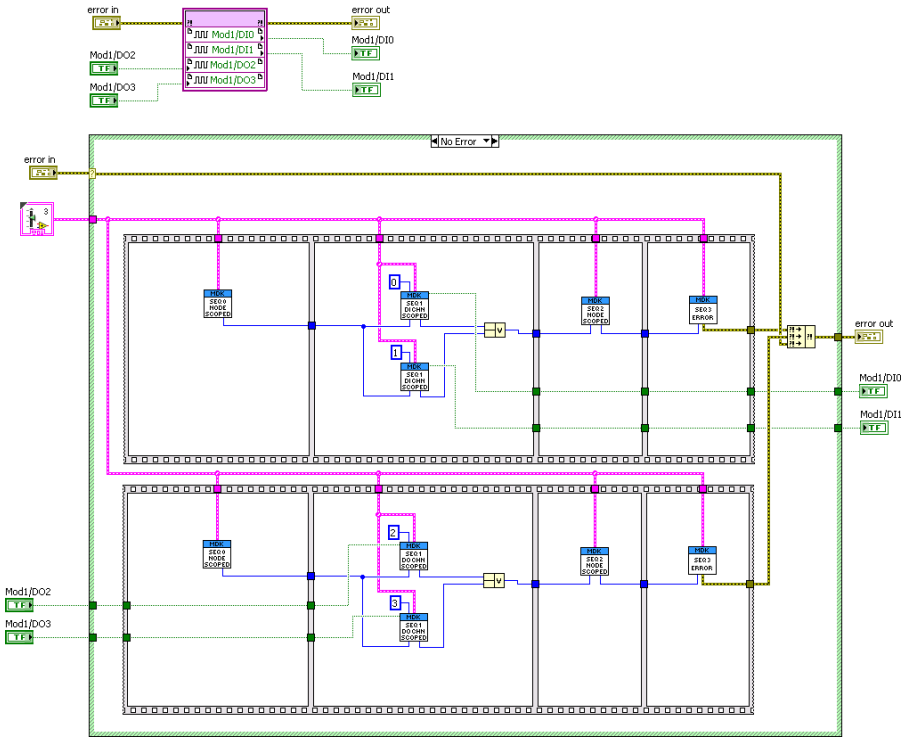
- **Terminal 4**—Read I/O Node parameter (from I/O node)

Merged I/O Node VIScriptInfo (Advanced)

I/O items that use the same interface and VISriptInfo will be scripted together when placed in the same grown I/O node. When the I/O items in a grown node are scripted together, the node scoped VIs in the VIScriptInfo will only be scripted once per node, and the channel scoped VIs will be scripted once per each channel in the grown I/O node.

I/O items that use a different interface and VIScriptInfo will be scripted separately, as shown in Figure 6-11. For example, if your module has DI and DO, the DI I/O items will be scripted together and the DO I/O items will be scripted together.

Figure 6-11. DI and DO from Same Module Scripted Separately

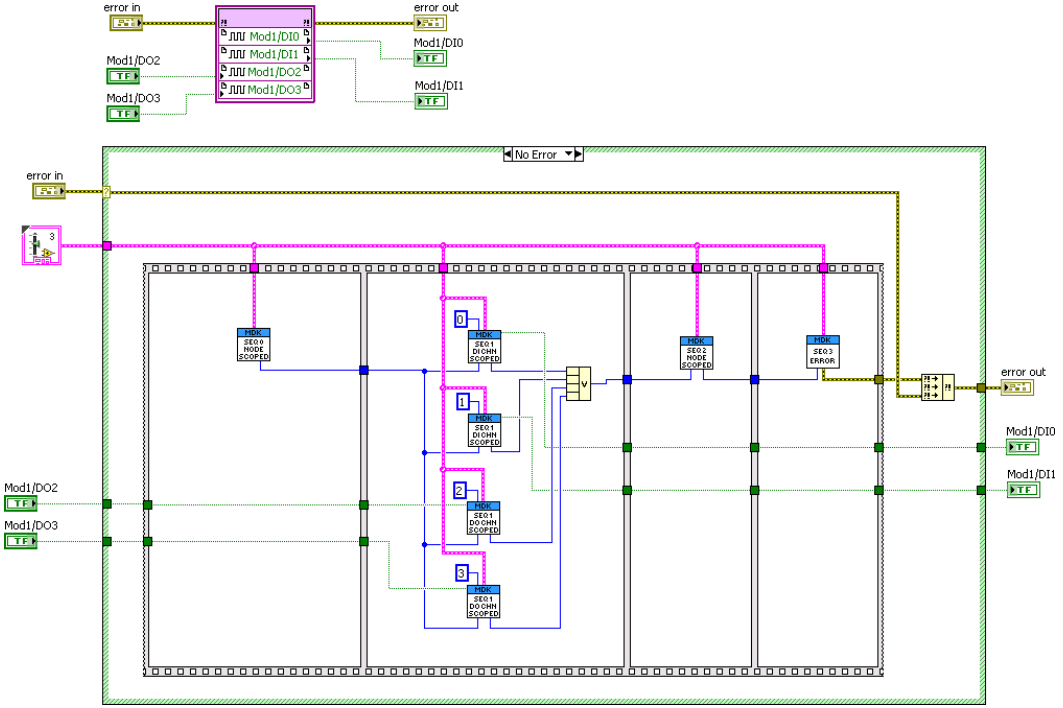


To script I/O items with different interfaces and VIScriptInfos together, you must specify it in the MergedIONodeVIScriptInfoList portion of the XML. When merging the VIScriptInfos for different interfaces, those VIScriptInfos must be compatible and must follow the following rules:

- They have the exact same number of VIs specified
- They have the same settings for UseInstanceData
- All node-specific VIs (including Error Handling VI) must be the same VIs

Figure 6-12 shows how the I/O node would be scripted if the DI and DO were specified in the XML to have merged VIScriptInfo.

Figure 6-12. DI and DO from Same Module Scripted Together



Error Codes

NI recommends you use error codes in the API of your module. NI reserved error codes ±358600–358619 that you can use for your modules.

In addition to module specific error codes that are defined by third party module developers, there are also two error codes that we encourage you to use:

- 65536: Module Communication Error
  - Use this error code when the module is removed or invalid (**No Module** or **Invalid** module status) and when you are unable to communicate with the module
- 65537: Incorrect Module Error
  - Use this error code when the module is incorrect (**Incorrect** module status)
- 65673: Incorrect Program Mode Error
  - Use this error code when the slot is in the incorrect program mode (**Incorrect Program Mode** module status)

Creating Custom Error Code Files

If you create your own module-specific error codes, we recommend that you ship an error code file as part of your deployable module support files. This allows your customers to see a custom error description through the LabVIEW Explain Error help menu. Refer to the *Defining Custom Error Codes in Text Files* topic in the *LabVIEW Help* for more information on creating a custom error code file.

7

Modules Support VI Best Practices

This section provides best practices for writing Module Support VIs.

Error Terminals on Interface Method Nodes

Although LabVIEW FPGA allows you to enable error terminals on the C Series Communication Core and Internal Channel Method Nodes, NI does not recommend that you use error terminals on these method nodes. These error terminals contains a 33-bit wide signal that must be registered in the FPGA, which causes an unnecessary increase in FPGA utilization.

The C Series Communication Core and Internal Channel Method Nodes do not return any error codes. Internal error information from the C Series Communication Core can be retrieved from the debug register. Also, the Internal Channel cannot produce any errors.

Do not use the error wire to force execution order in the FPGA. Use the flat sequence structure to control execution order. The flat sequence structure utilizes the FPGA efficiently and does not waste any resources.

Changing Interfaces

When your XML is written and you are using your module in LabVIEW projects, use caution when changing the end user API or the internal channels.

If you change the type or data type of the internal channel, you must delete and replace all method nodes on the block diagram using that channel. If you change the terminals of any API elements, you must delete and replace the terminals on the block diagram. If you add or remove any internal channels, you should delete and re-add the module in any project that uses it in Release mode.

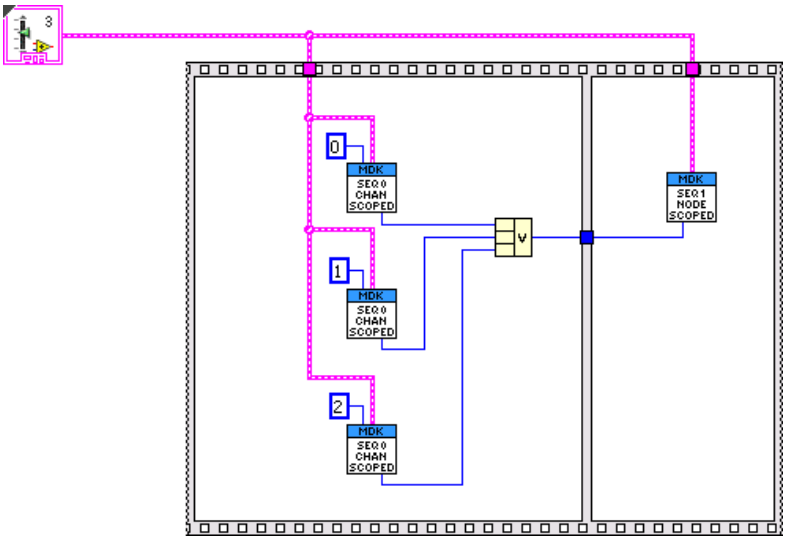
If you do not replace the nodes to the I/O items that have changed on the block diagram, you may encounter various LabVIEW and code generation errors.

The XML for these LabVIEW project items is intended to be static. LabVIEW may not properly mutate the nodes on the block diagram when the interfaces to those project items change, which may cause some warnings when you close LabVIEW after you make these changes.

Using Channel Scoped VIs to Create a Channel List

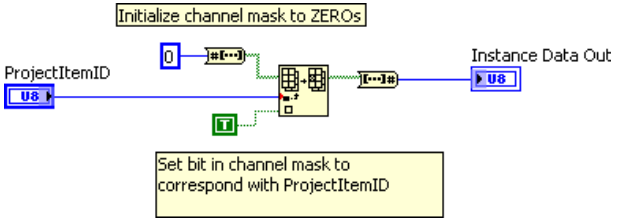
InstanceData sends information from one sequence frame of your Module Support VIs to the next. Because channel scoped VIs are scripted in parallel, the InstanceData out of channel scoped VIs must be ORed together, as shown in Figure 7-1, before wiring them to the InstanceData input of the next frame. You can use this functionality to create a channel mask that tells the Module Resource VI what channels are in the node that is executing.

Figure 7-1. Channel Scoped VIs ORed Together



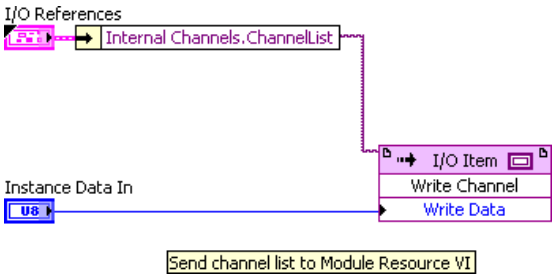
Each channel scoped VI sets a bit in the instance data that corresponds to the ProjectItemID of the I/O item that the VI is being scripted for. Figure 7-2 shows a channel scoped VI setting a bit in the Instance Data Out. In the XML, the ProjectItemID corresponds with the channel number of the I/O. When the Instance Data Out signals from each of the channel scoped VIs are ORed together, it creates a channel mask.

Figure 7-2. Setting a Bit in the Instance Data Out



The node scoped VI sends the channel mask to the Module Resource VI, as shown in Figure 7-3. The Module Resource VI handles all of the I/O items together.

Figure 7-3. Sending the Channel List to the Module Resource VI



Refer to the MDK-9902 example for information on creating a channel list. Using this programming method can make multiple I/O items in a grown I/O node execute atomically in the Module Resource VI and have a single convert pulse for all of them.

Click **Start»Programs»National Instruments»NI-RIO»CompactRIO» CompactRIO MDK 2»cRIO MDK Examples** to open the MDK 2 example directory.

Using the Module Status

The Module Resource VI should constantly monitor the module status and optionally execute the API elements depending on what the module status is. Use the following guidelines for using the module status in the Module Resource VI.

- **Unknown**—Wait until the module status is determined before deciding how to proceed. The module status will only be unknown for a short period of time while the module is being identified.
- **Correct**—Allow all API elements to execute.
- **Incorrect**—Only allow the Module ID, Vendor ID and Serial Number Property Nodes to execute. These property nodes should return with a warning 65537. All other API elements should not be executed and should be returned immediately with error 65537.
- **No Module, Invalid**—Do not execute any API elements. Return them immediately with error 65536.

The Module Resource VI must only utilize the Idle and ID modes of the C Series Communication Core when the module status is incorrect or invalid. Never enter Normal Operation and Auxiliary Communication modes when the module status is Incorrect or Invalid.

When the module status is no module, the C Series Communication Core automatically goes into Idle mode and you cannot transition into another mode until a module is present.

8

Module Manufacturing

The C Series Communication Core automatically identifies your module in the chassis slot and operates on it. The EEPROM of the module is blank during manufacturing, so the C Series Communication Core sees your module as invalid.

To write to the EEPROM of the module during manufacturing, you can create a separate set of module support files to make a different module type that is used during manufacturing. Use the export utility to create a development mode export of manufacturing module support. When writing the EEPROM of your module, you must calculate several CRC values.

During manufacturing, use the Module Status Override Debug register to make the module status Correct in the C Series Communication Core. This allows you to perform EEPROM writes and reads and simple manufacturing tests of your module hardware such as SPI and DIO.

Refer to the MDK-MFG example directory for an example of manufacturing module support. Click **Start»Programs»National Instruments»NI-RIO»CompactRIO»CompactRIO MDK 2»cRIO MDK Examples** to open the MDK 2 example directory.

9

Using the MDK 2 Examples

Click **Start»Programs»National Instruments»NI-RIO»CompactRIO»CompactRIO MDK 2»cRIO MDK Examples** to open the MDK 2 example directory. The MDK examples directory includes the Module SubItem Icons, Module Support Development, and Release Mode Projects folders.

Module SubItem Icons

This folder contains example Module SubItem Icons that you can use with Module SubItems. In addition to these example icons, you can create your own custom icon for your SubItem.

Module Support Development

This folder contains the module support source code for four example modules. Each of the module support folders contain an ExcludeFromExport folder that contains a Development mode LabVIEW project. In order to use the module in Development mode, you must first run the Module Support Export Utility for that particular example module.

Release Mode Projects

This folder contains LabVIEW projects and VIs that use the example modules in Release mode. In order to use the module in Release mode, you must first run the Module Support Export Utility for that particular example module.

MDK 2 Example Modules

MDK 2 provides four example modules.

MDK-MFG

Only use the MDK-MFG module in Development mode. There is no Release mode project for the MDK-MFG module. This example module is useful when manufacturing your module hardware. The ExcludeFromExport folder contains examples of reading and writing the module EEPROM and generating CRC values. It also shows how you can use the Debug interface to override the module status.

MDK-9901

The MDK-9901 module is a four channel analog output module. The Module Support VIs use the same communication protocol as the NI 9263. You can update the EEPROM of the NI 9263 to match the MDK-9901 Vendor ID, Product ID, and Module Model Code. This allows you to run the FPGA VI in the Release mode project and see voltages on the AO channels

of the NI 9263. The MDK-9901 EEPROM values can be obtained from the `MDK-9901_ModuleType.xml` file. The ExcludeFromExport folder contains a Development mode project that shows how the various API elements get scripted behind the I/O nodes.

MDK-9902

The MDK-9902 module contains examples of all of the API elements that you can create using MDK 2. It does not utilize any of the C Series Communication Core interfaces. Instead, it uses the Module Resource VI, node VIs and internal channels to demonstrate how to create different API elements. The ExcludeFromExport folder contains a Development mode project that shows how merged VI Script Info interfaces get scripted behind a grown I/O node.

MDK-9903

The MDK-9903 module demonstrates how to use the C Series Communication Core interfaces inside of an SCTL. Its API is a method node that may also be run inside of an SCTL. The ExcludeFromExport folder contains a Development mode project that shows how the method node gets scripted behind the I/O nodes.

A

Module XML

Data Types

Table A-1 lists the Module XML data types.

Table A-1. XML Data Types

Data Type	Valid Range / Values
Integer	Hexadecimal (i.e., 0x1234) & Decimal
String	String characters
Boolean	true, false
Enumerated	Enumerated type; select from a list of values
SimpleName	a-z, A-Z, 0-9, -, _
ProjectItemName	a-z, A-Z, 0-9, -, _, (,), [,], :
RestrictedString	a-z, A-Z, 0-9, -, _, (,), [,], :, {, }, \, !, @, #, /, +, *, ?, |, ., ^, *space*

Module XML

Module Type XML

All the Module Type XML tags are required. Table A-2 lists the Module Type XML tags.

Table A-2. Module Type XML Tags

XML Level	XML Tag	Data Type	Description
1	ModuleName	String	Specifies the name of the module. It typically consists of a two letter acronym followed by the module model code. This module name is also used to name the files and folders that make up the module support.
1	Description	String	Appears in the New C Series Module dialog box when adding a module to a LabVIEW project.

Table A-2. Module Type XML Tags

XML Level	XML Tag	Data Type	Description
1	VendorID	Integer	It must match the Vendor ID that is stored in the module EEPROM. The VendorID is typically specified with a leading 0x to indicate that it is hexadecimal.
1	ProductID	Integer	It must match the Product ID that is stored in the module EEPROM. The ProductID is typically specified with a leading 0x to indicate that it is hexadecimal.
1	ModelCode	Integer	It must match the Module Model code that is stored in the module EEPROM. The ModelCode is typically specified as decimal since it corresponds with the model number of the module.

Note When you are prototyping your module support, make the VendorID, ProductID, and Model Code for your module unique. NI-RIO uses these values to identify the module and will produce an error if these values are duplicated in other module support files.

Module Support XML

All of the Module Support XML tags and attributes are either required or optional. If an optional tag is not present, the default value is used or the particular functionality that the tag specifies is not included in the module support.

Table A-3 lists the Module Support XML tags, sections, and attributes.

Table A-3. MDKVersion and DevelopmentMode Tags

XML Level	XML Tag	Data Type	Required/ Optional	Description
1	MDKVersion Tag	String	Required	Specifies the version of the module development kit used to develop the module support. Use the `Mdk2Utility_GetinstalledMDKVersion.vi` utility located at `labview\vi.lib\LabVIEW Targets\FPGA\cRIO\shared\nicrio_Mdk2Utility` to verify the MDK support version that is installed with the NI-RIO version on your computer.
1	DevelopmentMode Tag	Boolean	Optional	Specifies what mode (Release or Development) the module appears in the LabVIEW project. If this tag is not present, the module support will be in Release mode.

Table A-4 list the tags, sections, and attributes for the Module section.

Table A-4. Module Section

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
1	Module Section	—	Required	Defines the end user APIs of the module. Each of the items listed in the module section of the XML corresponds to a LabVIEW project item that appears in the project when the module is added. Each of these project items can specify what interfaces (I/O, method and property nodes) can be used with them. The module itself is also a project item.
2	Name Attribute	ProjectItemName	Required	Specifies the name of the module. This must match the module name defined in the ModuleType XML file.
2	ProjectItemID Tag	Integer	Optional	Specifies the value used by the Module Support VI scripters. When your Module Support VIs are scripted, one of the inputs to your VI is the value specified by the ProjectItemId. This identifies what project item the particular VI is operation on.
2	SupportedInterfaceList Section	—	Optional	Contains a list of interfaces, defined in another part of the XML, that are available on the module. For the module, these interfaces can be method or property nodes. I/O nodes are not supported directly on the module. Each item in the SupportedInterfaceList is put inside of Interface XML tags.
2	ResourceVI Section	—	Optional	Specifies a VI that will be scripted into the FPGA diagram at compile time. It does not correspond to any API elements that are placed on the block diagram. Instead, this VI is instantiated once per module in the LabVIEW project.

Table A-4. Module Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
3	Name Attribute	Simple Name	Required	Specifies the name of the VI that will be scripted into the FPGA diagram.
2	ModuleSubItemList Section	—	Optional	Contains a list of module sub-items. Modules that have functionality that are not typical I/O can use module sub-items instead of I/O channels. For example, the NI 9802 uses sub-items in the LabVIEW project for two SD card slots and the NI 9871 uses sub-items in the LabVIEW project for two serial ports.
2	ModuleSubItem Section	—	Optional	Appears in the LabVIEW project under the module item.
3	Name Attribute	ProjectItemName	Required	Specifies the name of the module sub-item that appears in the LabVIEW project
3	ProjectItemId Tag	Integer	Optional	Specifies the value used by the Module Support VI scripters. The range for this tag is 0 to 255. When your Module Support VIs are scripted, one of the inputs to your VI is the value specified by the ProjectItemId. This is used to identify what project item the particular VI is operation on.
3	SupportedInterfaceList Section	—	Optional	Contains a list of interfaces, defined in another part of the XML, that are available on the module sub-item. For module sub-items, these interfaces can be method or property nodes. I/O nodes are not supported for module sub-items. Each item in the SupportedInterfaceList is put inside of Interface XML tags.
4	Interface Tag	String	Optional	—

Table A-4. Module Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
3	ModuleSubItemIcon Tag	SimpleName (.png)	Optional	Specifies the icon used in the project to show the module sub-item. Sample icons are located in the `Module SubItem Icons` directory. Click **Start»Programs»National Instruments»NI-RIO»CompactRIO» CompactRIO MDK2»cRIO MDK Examples** to open the MDK 2 example directory.
2	IOChannelList Section	—	Optional	Contains a list of the I/O channels that will be available on your module.
3	IOChannel Section	—	Optional	Appears as an I/O item in the LabVIEW project. Like NI I/O channels, the I/O items will be automatically placed into a module folder in the LabVIEW project when the module is added to the FPGA target.
4	Name Attribute	ProjectItemName	Required	Specifies the name of the I/O channel that appears in the LabVIEW project.
4	ProjectItemId Tag	Integer	Optional	Specifies the value used by the Module Support VI scripters. The range for this tag is 0 to 255. When your Module Support VIs are scripted, one of the inputs to your VI is the value specified by the ProjectItemId. This is used to identify what project item the particular VI is operation on.
4	SupportedInterfaceList Section	—	Optional	Contains a list of interfaces, defined in another part of the XML, that are available on the module. For I/O channel items, these interfaces can be I/O, method, or property nodes. Each of these items in the SupportedInterfaceList is put inside of Interface XML tags.

Table A-4. Module Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
5	Interface Tag	RestrictedString	Optional	—
5	ParallelDigitalInterface Tag	Enumerated	Optional	Contains interfaces for parallel digital I/O lines. The options for this tag are DIO0-DIO7, DI0-DI7, DO0-DO7. The support for these interfaces is provided by the MDK 2 software. When using parallel digital I/O lines, you do not need to specify VIs that will get scripted beneath the I/O nodes like you do for other interfaces. Each parallel digital I/O line is put inside ParallelDigitalInterface XML tags.

Table A-5 list the tags, sections, and attributes for the PropertyNodeInterfaceList section.

Table A-5. PropertyNodeInterfaceList Section

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
1	PropertyNodeInterface List Section	—	—	Contains the details for each of the Property Node interfaces that were specified in the SupportedInterfaceList sections of the project items.
2	Interface Section	—	—	—
3	Name Attribute	RestrictedString	Required	Specifies the name of the property node that is available on the LabVIEW block diagram.

Table A-5. PropertyNodeInterfaceList Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
3	DataType Tag	Enumerated	Required	Specifies the data type of the property node. The options for this tag are I8, U8, I16, U16, I32, U32, and Boolean. You may also specify a control name in this tag. This control may be a cluster, array, fixed-point, or any other data type that is allowed in LabVIEW FPGA. The control name should be specified with the `.ctl` extension, for example `MDK-1234_MyControl.ctl`. The control must be located directly inside of the module support folder and not within a subfolder.
3	Direction Tag	Enumerated	Required	Specifies the direction of the property node. The options for this tag are Read, Write, BiDirectional.
3	DefaultDirection Tag	Enumerated	Required when the direction is BiDirectional	Specifies the direction of the node when it is first placed on the block diagram. It is not allowed when the direction is Read or Write. The options for the DefaultDirection are Read and Write.
3	NodeIcon Tag	Enumerated	Required	Specifies the icon that appears in the node on the block diagram. The options for this tag are AI, AO, DI, DO, DIO, and Port.
3	WriteVIScriptInfo Section	—	Required when the direction of the property node is Write or BiDirectional	Specifies the Module Support VIs for the property node and how they should be connected.

Table A-5. PropertyNodeInterfaceList Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
4	Name Attribute	SimpleName	Required	Specifies the name of the WriteVIScriptInfo section. This name does not appear in the end user API and is only used internally within the XML file. This name must be unique and not used for any other VIScriptInfo section.
4	UseInstanceData Tag	Boolean	Optional	Specifies if the instance data terminal will be used on the Module Support VIs.
5	VIList Section	—	—	Contains a list of Module Support VIs that will be scripted beneath the property node. For property nodes, you may only specify one VI in addition to the optional Error Handling VI.
5	VI Section	—	—	—
6	Name Attribute	SimpleName (`.vi`)	Required	Specifies the name of the VI that will be scripted beneath the property node. Specify the VI name with the .vi extension, for example, `MDK-1234_MyVI.vi`. The VI must be located directly inside of the module support folder and not within a subfolder.
6	SequenceOrder Tag	Integer	Required	Specifies where in the sequence of frames this particular VI will appear when multiple VIs are scripted in sequence beneath the property node. The range for this tag is 0 to 255.
6	VIHasTerminalConnection Tag	Boolean	Optional	Indicates that this particular VI contains a terminal that will be connected to the input or output terminal of the property node. This tag is optional, but one and only one VI in the VIList must contain this tag set to true.

Table A-5. PropertyNodeInterfaceList Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
6	ErrorHandling Tag	Boolean	Optional	Indicates that this particular VI contains a terminal that will be connected to the error output of the property node. To have your property node produce error codes, one and only one VI in the VIList must contain this tag set to true.
3	ReadVIScriptInfo Section	—	Required when the direction of the property node is Read or BiDirectional	Specifies the Module Support VIs for the property node and how they should be connected.

Table A-6 list the tags, sections, and attributes for the MethodNodeInterfaceList section.

Table A-6. MethodNodeInterfaceList Section

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
1	MethodNodeInterfaceList Section	—	—	Contains the details for each of the Method Node interfaces that are specified in the MethodInterfaceList sections of the project items.
2	Interface Section	—	—	—
3	Name Attribute	RestrictedString	Required	Specifies the name of the method node that is available on the LabVIEW block diagram. This name must be unique and not used for any other interface section.

Table A-6. MethodNodeInterfaceList Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
3	MethodNodeTerminalList Section	—	Optional	Defines the terminals that will be shown on the method node in the LabVIEW block diagram. If this section is not defined, the method node will not have any terminals.
4	MethodNodeTerminal Section	—	—	—
5	Name Attribute	RestrictedString	Required	Specifies the name of the method node terminal that appears on the LabVIEW block diagram.
5	DataType Tag	Enumerated or SimpleName (`.ctl`)	Required	Specifies the data type of the method node. The options for this tag are I8, U8, I16, U16, I32, U32, and Boolean. You may also specify a control name in this tag. This control may be a cluster, array, fixed-point, or any other data type that is allowed in LabVIEW FPGA. The control name should be specified with the `.ctl` extension, for example `MDK-1234_MyControl.ctl`. The control must be located directly inside of the module support folder and not within a subfolder.
5	Direction Tag	Enumerated	Required	Specifies the direction of the method node terminal. The options for this tag are Read and Write.
5	Required Tag	Boolean	Optional	Specifies the wiring on a Write terminal. This tag can only be used with Write terminals. When set to true, the write terminal on the method node must have something wired to it on the block diagram. When set to false, the write terminal can be left unconnected and a default value will be used.

Table A-6. MethodNodeInterfaceList Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
5	TerminalOrder Tag	Integer	Required	Specifies the order in the method node in which the terminal appears. The order starts with the top most terminal which has a TerminalOrder of zero. The range for this tag is 0 to 255.
3	NodeIcon Tag	Enumerated	Required	Specifies the icon that appears in the node on the block diagram. The options for this tag are AI, AO, DI, DO, DIO, and Port.
3	MethodVIScriptInfo Section	—	Required	Specifies the Module Support VIs for the method node and how they should be connected.
4	Name Attribute	SimpleName	Required	Specifies the name of the MethodVIScriptInfo section. This name does not appear in the end user API and is only used internally within the XML file. This name must be unique and not used for any other VIScriptInfo section.
4	UseInstanceData Tag	Boolean	Optional	Specifies if the instance data terminal will be used on the Module Support VIs.
5	VIList Section	—	—	Contains a list of Module Support VIs that will be scripted beneath the method node. For method nodes, you may only specify one VI in addition to the optional Error Handling VI.
5	VI Section	—	—	—

Table A-6. MethodNodeInterfaceList Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
6	Name Attribute	SimpleName (`.vi`)	Required	Specifies the name of the VI that will be scripted beneath the method node. Specify the VI name with the .vi extension, for example, `MDK-1234_MyVI.vi`. The VI must be located directly inside of the module support folder and not within a subfolder.
6	SequenceOrder Tag	Integer	Required	Specifies where in the sequence of frames this particular VI will appear when multiple VIs are scripted in sequence beneath the method node. The range for this tag is 0 to 255.
6	TerminalConnectionList Section	—	Optional	This tag is optional, but if the method node contains any, one and only one VI in the VIList must contain this section.
7	TerminalConnection Section	—	Optional	Each terminal of the method node must have a corresponding terminal connection in the TerminalConnectionList.
7	Name Attribute	SimpleName	Required	Specifies the name of the TerminalConnection section. This name does not appear in the end user API and is only used internally within the XML file. This name must be unique and not used for any other TerminalConnection section.
7	NodeTerminalName Tag	RestrictedString	Required	Specifies the node terminal from the MethodNodeTerminalList section that is being connected.

Table A-6. MethodNodeInterfaceList Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
7	ConnectorPaneTerminal Tag	Integer	Required	Specifies the terminal on the Module Support VI connector pane that is being connected to the terminal of the method node. The range for this tag is 0 to 255.
6	ErrorHandling Tag	Boolean	Optional	Indicates that this particular VI contains a terminal that will be connected to the error output of the method node. To have your method node produce error codes, one and only one VI in the VIList must contain this tag set to true.

Table A-7 list the tags, sections, and attributes for the IONodeInterfaceList section.

Table A-7. IONodeInterfaceList Section

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
1	IONodeInterfaceList Section	—	—	Contains the details for each of the I/O Node interfaces that were specified in the IONodeInterfaceList sections of the project items.
2	Interface Section	—	—	—
3	Name Attribute	SimpleName	Required	Specifies the name of the Interface section, This name does not appear in the end user API. because the I/O nodes inherit names from the project items that they are associated with. For example, the I/O node for the AI0 project will be called AI0. This name is only used internally within the XML file. This name must be unique and not used for any other Interface sections.

Table A-7. IONodeInterfaceList Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
3	DataType Tag	Enumerated or SimpleName (`.ctl`)	Required	Specifies the data type of the I/O node. The options for this tag are I8, U8, I16, U16, I32, U32, and Boolean. You may also specify a control name in this tag. This control may be a cluster, array, fixed-point, or any other data type that is allowed in LabVIEW FPGA. The control name should be specified with the `.ctl` extension, for example `MDK-1234_MyControl.ctl`. The control must be located directly inside of the module support folder and not within a subfolder.
3	Direction Tag	Enumerated	Required	Specifies the direction of the I/O node terminal. The options for this tag are Read, Write, and BiDirectional.
3	DefaultDirection Tag	Enumerated	Required when the direction is BiDirectional	Specifies the direction of the node when it is first placed on the block diagram. It is not allowed when the direction is Read or Write. The options for the DefaultDirection are Read and Write.
3	NodeIcon Tag	Enumerated	Required	Specifies the icon that appears in the node on the block diagram. The options for this tag are AI, AO, DI, DO, DIO, and Port.
3	WriteVIScriptInfo Section	—	Required when the direction of the I/O node is Write or BiDirectional	Specifies the Module Support VIs for the I/O node and how they should be connected.

Table A-7. IONodeInterfaceList Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
4	Name Attribute	SimpleName	Required	Specifies the name of the WriteVIScriptInfo section. This name does not appear in the end user API and is only used internally within the XML file. This name must be unique and not used for any other VIScriptInfo section.
4	UseInstanceData Tag	Boolean	Optional	Specifies if the instance data terminal will be used on the Module Support VIs.
5	VIList Section	—	—	Contains a list of Module Support VIs that will be scripted beneath the I/O node. For I/O nodes, you can specify multiple Module Support VIs in addition to the Error Handling VI.
5	VI Section	—	—	—
6	Name Attribute	SimpleName (`.vi`)	Required	Specifies the name of the VI that will be scripted beneath the I/O node. Specify the VI name with the .vi extension, for example, `MDK-1234_MyVI.vi`. The VI must be located directly inside of the module support folder and not within a subfolder.
6	SequenceOrder Tag	Integer	Required	Specifies where in the sequence of frames this particular VI will appear when multiple VIs are scripted in sequence beneath the I/O node. The range for this tag is 0 to 255.

Table A-7. IONodeInterfaceList Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
6	VIScope Tag	Enumerated	Optional	Specifies how many times this VI will be scripted into a grown I/O Node. The options for this tag are NodeScoped and ChannelScoped. When set to NodeScoped, the VI will only be scripted once beneath the I/O node no matter how many channels are in the grown node. When set to ChannelScoped, the VI will be scripted once for each channel in the grown I/O node. When the tag is not present, the VI will default to NodeScoped.
6	VIHasTerminalConnection Tag	Boolean	Optional	Indicates that this particular VI contains a terminal that will be connected to the input or output terminal of the I/O node. This tag is optional, but one and only one VI in the VIList must contain this tag set to true.
6	ErrorHandling Tag	Boolean	Optional	Indicates that this particular VI contains a terminal that will be connected to the error output of the I/O node. To have your I/O node produce error codes, one and only one VI in the VIList must contain this tag set to true.
3	ReadVIScriptInfo Section	—	Required when the direction of the I/O node is Read or BiDirectional	Specifies the Module Support VIs for the I/O node and how they should be connected.

Table A-8 list the tags, sections, and attributes for the MergedIONodeVISCriptInfoList section.

Table A-8. MergedIONodeVISCriptInfoList Section

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
1	MergedIONodeVISCriptInfoList Section	—	—	Specifies how particular I/O Node interfaces will be merged when different I/O items are in the same grown I/O node.
2	MergeIONodeVIScript Info Section	—	Optional	Specifies VIScriptInfos that can be merged together when used in the same grown I/O node.
3	Name Attribute	SimpleName	Required	Specifies the MergedIONodeVIScriptInfo section name. This name does not appear in the end user API and is only used internally within the XML file. This name must be unique and not used for any other MergedIONodeVIScriptInfo section.
3	VIScriptInfoName Tag	SimpleName	—	Two or more VIScriptInfoName tags may appear within the MergedIONOdeVIScriptInfo section. These names correspond to the names given to the particular VIScriptInfo (WriteVIScriptInfo, ReadVIScriptInfo) sections that were listed in the different interfaces.

Table A-9 list the tags, sections, and attributes for the InternalChannelList section.

Table A-9. InternalChannelList Section

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
1	InternalChannelList Section	—	Optional	Lists the different internal channels that will be used within the Module Support VIs.
2	InternalChannel Section	—	Optional	Specifies how the internal channel will operate.

Table A-9. InternalChannelList Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
3	Name Attribute	SimpleName	Required	Specifies the InternalChannel section name. This name does not appear in the end user API since internal channels are hidden when the module is used in Release mode. All internal channel names must be unique.
3	InternalChannelType Tag	Enumerated	Required	Specifies which type of functionality the internal channel will have. The options for this tag are Asynchronous, Blocking, and Occurrence.
3	DataType Tag	Enumerated	Required when the type is Asynchronous or Blocking	Specifies the data type of the internal channel. The options for this tag are I8, U8, I16, U16, I32, U32, and Boolean. You may also specify a control name in this tag. This control may be a cluster, array, fixed-point, or any other data type that is allowed in LabVIEW FPGA. The control name should be specified with the `.ctl` extension, for example `MDK-1234_MyControl.ctl`. The control must be located directly inside of the module support folder and not within a subfolder. This tag is optional because the Occurrence internal channel type does not specify a data type. If you are using the Blocking or Asynchronous internal channel types, then you must specify a data type.

Table A-10 list the tags, sections, and attributes for the ModuleModeDefinition section.

Table A-10. ModuleModeDefinition Section

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
1	ModuleModeDefinition Section	—	Required	Specifies how the C Series Communication Core will operate in its different modes of operation.
2	NormalOperationMode Section	—	Optional	This section is optional, but all modules should contain a NormalOperationMode section since this is the mode in which the primary communication to the module occurs.
3	SPIConfiguration Section	—	Optional	Reserves the following CompactRIO bus lines for the SPI engine in Normal Operation mode: SPI_CLK, ~SPI_CS, MISO, MOSI. When reserved by the SPI engine, these lines may not be used for digital I/O in Normal Operation mode. This section must be present in order to use the SPI interfaces of the C Series Communication Core when the module is in Normal Operation mode.
4	SPIHalfTauTicks Tag	Integer	Required	Specifies how wide the SPI Clock 1/2τ period is in 25 ns base clock ticks. The range for this tag is 2 to 65535. If your module has a SPI Tau of 150 ns, the SPIHalfTauTicks tag will be set to 3. The 150 ns SPI Clock period will be 6 base clock ticks wide. The SPI Clock signal must be symmetric, so it will have 3 high ticks and 3 low ticks.
3	Convert PulseConfiguration Section	—	Optional	Reserves the ~CONVERT line of the CompactRIO bus for the Convert Pulse Logic when present. This section must be present in order to use the Pulse Convert interface of the C Series Communication Core.

Table A-10. ModuleModeDefinition Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
4	ConvertPulseWidth Tag	Enumerated	Required	Specifies how long the convert pulse will be. The options for this tag are Short, Medium, and Long. These three pulse widths correspond to the convert pulse widths described in the *CompactRIO Module Development Kit Hardware User Manual*.
3	DoneWaitConfiguration Section	—	Optional	Reserves the ~DONE line of the CompactRIO bus for the Done Wait Logic when present. This section must be present in order to use the Wait on Done interface of the C Series Communication Core.
4	DoneWaitTimeoutTicks Tag	Integer	Required	Specifies how long the C Series Communication Core will wait to see the ~DONE line as low before completing with a Timed Out error. The range for this tag is 0 to 4294967294. This value is specified in units of 25 ns base clock ticks.
3	DigitalLines Section	—	Optional	Lists the digital I/O lines of the CompactRIO bus that will be used by the Module Support VIs to perform basic DIO.

Table A-10. ModuleModeDefinition Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
4	DIO0 to DIO8 Tags	Enumerated	Optional	Specifies the function of the DIO0 to DIO8 lines. The options for these tags are: BiDirectional—The line defaults to an input on module insertion and may also be used as a digital output. ConstantOutputHigh—The line defaults to a HIGH output on module insertion. Digital output operations on this line are ignored. ConstantOutputLow—The line defaults to a LOW output on module insertion. Digital output operations on this line are ignored. InputOnly—The line defaults to an input on module insertion and may not be used as a digital output. Unused—The line defaults to an input on module insertion and may not be used as a digital output. If a DIO line is reserved by the SPI Engine, Convert Pulse logic or Done Wait logic, it may not be listed in the DigitalLines section. If a DIO line is not reserved or specified in the DigitalLines section, it will default to Unused.
2	AuxiliaryCommunication Section	—	Optional	—

Table A-10. ModuleModeDefinition Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
3	SPIConfiguration Section	—	Optional	Reserves the following CompactRIO bus lines for the SPI engine in Auxiliary mode: SPI_CLK, ~SPI_CS, MISO, MOSI. When reserved by the SPI engine, these lines may not be used for digital I/O in Auxiliary mode. This section must be present in order to use the SPI interfaces of the C Series Communication Core when the module is in Normal Operation mode.
4	SPIHalfTauTicks Tag	Integer	Required	Specifies how wide the SPI Clock half Tau period is in 25 ns base clock ticks. The range for this tag is 2 to 65535. If your module has a SPI Tau of 150 ns, the SPIHalfTauTicks tag will be set to 3. The 150 ns SPI Clock period will be 6 base clock ticks wide. The SPI Clock signal must be symmetric, so it will have 3 high ticks and 3 low ticks.
3	DigitalLines Section	—	Optional	Lists the digital I/O lines of the CompactRIO bus that will be used by the Module Support VIs to perform basic DIO.

Table A-10. ModuleModeDefinition Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
4	DIO0 to DIO8 Tags	Enumerated	Optional	Specifies the function of the DIO0 to DIO8 lines. The options for these tags are: BiDirectional—The line defaults to an input on module insertion and may also be used as a digital output. ConstantOutputHigh—The line defaults to a HIGH output on module insertion. Digital output operations on this line are ignored. ConstantOutputLow—The line defaults to a LOW output on module insertion. Digital output operations on this line are ignored. InputOnly—The line defaults to an input on module insertion and may not be used as a digital output. Unused—The line defaults to an input on module insertion and may not be used as a digital output. If a DIO line is reserved by the SPI Engine it may not be listed in the DigitalLines section. The C Series Specification dictates that the FUNC line is used to put the module in Auxiliary Communication mode. This means that the FUNC line is reserved and may not be used as DIO in Auxiliary mode. If a DIO line is not reserved or specified in the DigitalLines section, it will default to Unused.
2	DigitalLineInfo Section	—	Optional	Specifies advanced settings on the DIO interfaces of the C Series Communication Core.

Table A-10. ModuleModeDefinition Section (Continued)

XML Level	XML Tag/ Section/Attribute	Type	Required/ Optional	Description
3	SCTLOutputSyncRegs Section	—	—	Specifies how many synchronization registers are placed in the FPGA between the digital output I/O node on the block diagram and the CompactRIO bus pin.
4	DIO0 to DIO8 Tags	Enumerated	Optional	Specifies the value that sets the number of output registers for that DIO line. The options for these tags are 0 and 1. If a line is not listed in the SCTLOutputSyncRegs section, it will default to 1 sync register.
3	Arbitration Section	—	—	Specifies how the FPGA will arbitrate between multiple digital output I/O nodes that are placed on the block diagram.
4	DIO0 to DIO8 Tags	Enumerated	Optional	Specifies the value that sets the arbitration option for that DIO line. The options for these tags are NeverArbitrate, AlwaysArbitrate, and ArbitrateIfMultipleRequestorsOnly. The behavior of each of these arbitration options is identical to the NI 9401. Refer to the *LabVIEW Help* for more information on the NI 9401 arbitration options. If a line is not listed in the Arbitration section, it will default to NeverArbitrate.

B

Module Support XML Example

Use the following XML example for information on important XML tags, sections, and attributes when creating your Module Support XML.

```
<ModuleSupport>

  <MDKVersion>2.0</MDKVersion>

  <Module name="MDK-9902">

    <ProjectItemID>99</ProjectItemID>

    <ResourceVI name="MDK-9902_ModuleResource.vi"/>

    <SupportedInterfaceList>
      <Interface>Update Error Status</Interface>
    </SupportedInterfaceList>


    <ModuleSubItemList>

      <ModuleSubItem name="Module SubItem 1">
        <ProjectItemID>101</ProjectItemID>
        <SupportedInterfaceList>
          <Interface>Write FIFO</Interface>
          <Interface>Read FIFO</Interface>
        </SupportedInterfaceList>
        <ModuleSubItemIcon>MDK-9902_SubItemIcon.png</ModuleSubItemIcon>
      </ModuleSubItem>

      <ModuleSubItem name="Module SubItem 2">
        <ProjectItemID>102</ProjectItemID>
        <SupportedInterfaceList>
          <Interface>Write FIFO</Interface>
          <Interface>Read FIFO</Interface>
        </SupportedInterfaceList>
        <ModuleSubItemIcon>MDK-9902_SubItemIcon.png</ModuleSubItemIcon>
      </ModuleSubItem>

    </ModuleSubItemList>
```

The MDKVersion tag should match what is returned by MDK2Utility_GetInstalledMDKVersion.vi

The name attribute of the module should match the ModuleName listed in the ModuelType XML file. This is what appears in the LabVIEW project.

Every project item (module, subitem, channel) has a ProjectItemID. These numbers should be unique. This particular valued was selected so that it does not collide with other ProjectItemID values defined in the XML.

A ResourceVI may be specified for the module. This VI handles all of the communication with the module and executes different code depending on what API elements in running on the end user block diagram.

A module can support multiple Method Node and Property Node interfaces.

A module may contain one or more SubItems that show up under the module in the LabVIEW project.

The name attribute of the SubItem is what gets displayed in the LabVIEW project.

The ProjectItemID is used by the Module Support VIs to know which SubItem the API element is scoped to.

SubItems appear in the LabVIEW project under the module. You must define an icon that will be used to represent the SubItem. Example icons are provided with MDK 2.0.

A SubItem can have multiple Method Node and Property Node interfaces.

```
<IOChannelList>

  <IOChannel name="AI0">
    <ProjectItemID>0</ProjectItemID>
    <SupportedInterfaceList>
      <Interface>Analog Input Channel</Interface>
    </SupportedInterfaceList>
  </IOChannel>

  <IOChannel name="AI1">
    <ProjectItemID>1</ProjectItemID>
    <SupportedInterfaceList>
      <Interface>Analog Input Channel</Interface>
    </SupportedInterfaceList>
  </IOChannel>

  <IOChannel name="AO0">
    <ProjectItemID>2</ProjectItemID>
    <SupportedInterfaceList>
      <Interface>Analog Output Channel</Interface>
    </SupportedInterfaceList>
  </IOChannel>

  <IOChannel name="AO1">
    <ProjectItemID>3</ProjectItemID>
    <SupportedInterfaceList>
      <Interface>Analog Output Channel</Interface>
    </SupportedInterfaceList>
  </IOChannel>

  <IOChannel name="CustomChannel">
    <ProjectItemID>4</ProjectItemID>
    <SupportedInterfaceList>
      <Interface>Custom Channel</Interface>
      <Interface>Read and Set Data Value</Interface>
      <Interface>Data Value</Interface>
    </SupportedInterfaceList>
  </IOChannel>
```

A module may contain one or more I/O channels.

The name attribute of the I/O channel is what appears in the LabVIEW project. When the module is added to the project, all of the I/O channels are automatically added to a folder in the project.

Because I/O channels are typically numbered, it is convenient to align the ProjectItemID to the channel number. This make mapping between the two easier in the Module Support VIs.

An I/O channel may support I/O, Method, and Property Node interfaces.

```
    <IOChannel name="Parallel DIO">
      <ProjectItemID>5</ProjectItemID>
      <SupportedInterfaceList>
        <ParallelDigitalInterface>DIO0</ParallelDigitalInterface>
      </SupportedInterfaceList>
    </IOChannel>


  </IOChannelList>

</Module>


<InternalChannelList>

  <InternalChannel name="AnalogChannelNumberChan">
    <InternalChannelType>Blocking</InternalChannelType>
    <DataType>U8</DataType>
  </InternalChannel>

   <InternalChannel name="AnalogDataChan">
     <InternalChannelType>Blocking</InternalChannelType>
     <DataType>I16</DataType>
  </InternalChannel>

   <InternalChannel name="SubItemFIFODataChan">
     <InternalChannelType>Blocking</InternalChannelType>
     <DataType>I16</DataType>
  </InternalChannel>

  <InternalChannel name="StartNodeChan">
    <InternalChannelType>Blocking</InternalChannelType>
    <DataType>MDK-9902_NodeOperationType.ctl</DataType>
  </InternalChannel>

  <InternalChannel name="ChannelListChan">
    <InternalChannelType>Blocking</InternalChannelType>
    <DataType>U8</DataType>
  </InternalChannel>
```

When a ParallelDigitalInterface is specified on an I/O channel, it directly uses the digital I/O nodes of the C Series Communication Core. Module Support VIs are not written to handle the I/O nodes for this I/O channel.

A module may have zero, one, or more internal channels. However, if the module is being used in Release mode, the minimum number of internal channels is one.

Modules that support multiple API elements should use a StartNodeChan. This blocking internal channel is used to ensure that the Module Resource VI executes only one API element at a time. The data type of the internal channel is used to indicate what API element is executing.

```
  <InternalChannel name="OccurrenceChan">
    <InternalChannelType>Occurrence</InternalChannelType>
  </InternalChannel>

  <InternalChannel name="ModuleErrorChan">
    <InternalChannelType>Asynchronous</InternalChannelType>
    <DataType>MDK-9902_ErrorChan.ctl</DataType>
  </InternalChannel>

  <InternalChannel name="CustomChannelDataChan">
    <InternalChannelType>Asynchronous</InternalChannelType>
    <DataType>I32</DataType>
  </InternalChannel>

</InternalChannelList>

<IONodeInterfaceList>

  <Interface name="Analog Input Channel">
    <DataType>I16</DataType>
    <Direction>Read</Direction>
    <NodeIcon>AI</NodeIcon>

    <ReadVIScriptInfo name="AIChanVIScriptInfo">

      <UseInstanceData>true</UseInstanceData>

      <VIList>

        <VI name="MDK-9902_AnalogIONodeReserve.vi">
          <SequenceOrder>0</SequenceOrder>
          <VIScope>NodeScoped</VIScope>
        </VI>

        <VI name="MDK-9902_AnalogInputNodeCreateChannelList.vi">
          <SequenceOrder>1</SequenceOrder>
          <VIScope>ChannelScoped</VISc
        </VI>
```

A custom data type may be specified for internal channels.

The IONodeInterfaceList contains information about the I/O Node interfaces that were specified for the I/O channels.

The name attribute of an I/O Node interface is only used within the XML file. It does not appear in the LabVIEW project or on the block diagram.

An I/O node may be Read, Write, or BiDirectional. The value of the direction determines the types of VIScriptInfo that are defined for the interface.

The ReadVIScriptInfo section of the XML must be specified for this I/O node because it is a Read I/O Node.

The name attribute of the VIScriptInfo is only used within the XML file. It does not appear in the LabVIEW project or on the block diagram.

Because this I/O node specifies multiple VIs, the SequenceOrder tag must be used to determine the order of execution.

Channel Scoped I/O Node VIs are scripted for each channel in a grown I/O node.

```
      <VI name="MDK-9902_AnalogIONodeSendChannelList.vi">
        <SequenceOrder>2</SequenceOrder>
        <VIScope>NodeScoped</VIScope>
      </VI>

      <VI name="MDK-9902_AnalogIONodeGetInputData.vi">
        <SequenceOrder>3</SequenceOrder>
        <VIScope>ChannelScoped</VIScope>
        <VIHasTerminalConnection>true</VIHasTerminalConnection>
      </VI>

      <VI name="MDK-9902_AnalogIONodeError.vi">
        <SequenceOrder>4</SequenceOrder>
        <ErrorHandling>true</ErrorHandling>
      </VI>

    </VIList>
  </ReadVIScriptInfo>
</Interface>

<Interface name="Analog Output Channel">
  <DataType>I16</DataType>
  <Direction>Write</Direction>
  <NodeIcon>AI</NodeIcon>

  <WriteVIScriptInfo name="AOChanVIScriptInfo">

    <UseInstanceData>true</UseInstanceData>

    <VIList>

      <VI name="MDK-9902_AnalogIONodeReserve.vi">
        <SequenceOrder>0</SequenceOrder>
        <VIScope>NodeScoped</VIScope>
      </VI>

      <VI name="MDK-9902_AnalogOutputNodeCreateChannelList.vi">
        <SequenceOrder>1</SequenceOrder>
        <VIScope>ChannelScoped</VIScope>
        <VIHasTerminalConnection>true</VIHasTerminalConnection>
      </VI>
```

Node Scoped I/O Node VIs are scripted once for the entire node on the block diagram.

The NodeIcon tag specifies the type of icon that appears in the I/O node on the block diagram.

The UseInstanceData tag allows you to have an Instance Data wire scripted between the VIs in the sequence.

One and only one VI in the VIList must have the VIHasTerminalConnection tag set to true. This VI will only contain the signal that is wired to the terminal of the I/O node.

```
      <VI name="MDK-9902_AnalogIONodeSendChannelList.vi">
        <SequenceOrder>2</SequenceOrder>
        <VIScope>NodeScoped</VIScope>
      </VI>

      <VI name="MDK-9902_AnalogIONodeOutputEmpty.vi">
        <SequenceOrder>3</SequenceOrder>
        <VIScope>ChannelScoped</VIScope>
      </VI>

      <VI name="MDK-9902_AnalogIONodeError.vi">
        <SequenceOrder>4</SequenceOrder>
        <ErrorHandling>true</ErrorHandling>
      </VI>

    </VIList>
  </WriteVIScriptInfo>
</Interface>

<Interface name="Custom Channel">
  <DataType>MDK-9902_CustomChannel.ctl</DataType>
  <Direction>Read</Direction>
  <NodeIcon>AI</NodeIcon>

  <ReadVIScriptInfo name="CustomChannelReadVIScriptInfo">

    <VIList>

      <VI name="MDK-9902_CustomChannelIONode.vi">
        <SequenceOrder>0</SequenceOrder>
        <VIScope>ChannelScoped</VIScope>
        <VIHasTerminalConnection>true</VIHasTerminalConnection>
      </VI>

      <VI name="MDK-9902_GeneralNodeError.vi">
        <SequenceOrder>1</SequenceOrder>
        <ErrorHandling>true</ErrorHandling>
      </VI>

    </VIList>
  </ReadVIScriptInfo>
```

A VIList can specify one VI that is used for error handling. This VI will contain a signal that is wired to the error out terminal of the I/O node.

An I/O channel can specify a custom datatype.

```
  </Interface>
</IONodeInterfaceList>

<MethodNodeInterfaceList>
  <Interface name="Update Error Status">
    <MethodNodeTerminalList>
      <MethodNodeTerminal name="Error Status">
        <DataType>Boolean</DataType>
        <Direction>Write</Direction>
        <TerminalOrder>0</TerminalOrder>
      </MethodNodeTerminal>
      <MethodNodeTerminal name="Error Code">
        <DataType>I32</DataType>
        <Direction>Write</Direction>
        <TerminalOrder>1</TerminalOrder>
      </MethodNodeTerminal>
    </MethodNodeTerminalList>
    <NodeIcon>AI</NodeIcon>
    <MethodVIScriptInfo name="ModuleMethodScriptInfo">
      <VIList>
        <VI name="MDK-9902_ModuleMethod.vi">
          <SequenceOrder>0</SequenceOrder>
          <TerminalConnectionList>
            <TerminalConnection name="FirstConnection">
              <NodeTerminalName>Error Status</NodeTerminalName>
              <ConnectorPaneTerminal>0</ConnectorPaneTerminal>
            </TerminalConnection>
```

The MethodNodeInterfaceList contains information about the Method Node interfaces that were specified for the module, module SubItems, and I/O channels.

The name attributes of the Method Node interface is shown in the method node on the LabVIEW block diagram.

The name attribute of the MethodNodeTerminal is shown in the method node on the LabVIEW block diagram.

The TerminalOrder determines the placement of the terminal in the method node on the LabVIEW block diagram.

The direction specifies whether the terminal is into or out of the method node.

The TerminalConnectionList maps the MethodNodeTerminalList to the terminals of the Method Node subVI.

The name attribute of the TerminalConnection is only used within the XML file. It does not appear on the LabVIEW block diagram.

```
        <TerminalConnection name="SecondConnection">
          <NodeTerminalName>Error Code</NodeTerminalName>
          <ConnectorPaneTerminal>5</ConnectorPaneTerminal>
        </TerminalConnection>

        </TerminalConnectionList>

      </VI>

      <VI name="MDK-9902_GeneralNodeError.vi">
        <SequenceOrder>1</SequenceOrder>
        <ErrorHandling>true</ErrorHandling>
      </VI>

    </VIList>
  </MethodVIScriptInfo>
</Interface>

<Interface name="Write FIFO">

  <MethodNodeTerminalList>

    <MethodNodeTerminal name="Write Data">
      <DataType>I16</DataType>
      <Direction>Write</Direction>
      <TerminalOrder>0</TerminalOrder>
    </MethodNodeTerminal>

  </MethodNodeTerminalList>

  <NodeIcon>AI</NodeIcon>

  <MethodVIScriptInfo name="SubItemWriteMethodScriptInfo">

    <VIList>

      <VI name="MDK-9902_SubItemWriteMethod.vi">
        <SequenceOrder>0</SequenceOrder>
```

The ConnectorPaneTerminal specifies the terminal number of the Method Node subVI that will be connected to that particular terminal of the method node.

```
        <TerminalConnectionList>

          <TerminalConnection name="FirstConnection">
            <NodeTerminalName>Write Data</NodeTerminalName>
            <ConnectorPaneTerminal>0</ConnectorPaneTerminal>
          </TerminalConnection>

        </TerminalConnectionList>

      </VI>

      <VI name="MDK-9902_GeneralNodeError.vi">
        <SequenceOrder>7</SequenceOrder>
        <ErrorHandling>true</ErrorHandling>
      </VI>

    </VIList>
  </MethodVIScriptInfo>
</Interface>

<Interface name="Read FIFO">

  <MethodNodeTerminalList>

    <MethodNodeTerminal name="Read Data">
      <DataType>I16</DataType>
      <Direction>Read</Direction>
      <TerminalOrder>0</TerminalOrder>
    </MethodNodeTerminal>

  </MethodNodeTerminalList>

  <NodeIcon>AI</NodeIcon>

  <MethodVIScriptInfo name="SubItemReadMethodScriptInfo">

    <VIList>

      <VI name="MDK-9902_SubItemReadMethod.vi">
        <SequenceOrder>0</SequenceOrder>

        <TerminalConnectionList>
```

```
          <TerminalConnection name="FirstConnection">
            <NodeTerminalName>Read Data</NodeTerminalName>
            <ConnectorPaneTerminal>4</ConnectorPaneTerminal>
          </TerminalConnection>

        </TerminalConnectionList>

      </VI>

      <VI name="MDK-9902_GeneralNodeError.vi">
        <SequenceOrder>7</SequenceOrder>
        <ErrorHandling>true</ErrorHandling>
      </VI>

    </VIList>
  </MethodVIScriptInfo>
</Interface>

<Interface name="Read and Set Data Value">

  <MethodNodeTerminalList>

    <MethodNodeTerminal name="New Data Value">
      <DataType>I32</DataType>
      <Direction>Write</Direction>
      <TerminalOrder>0</TerminalOrder>
    </MethodNodeTerminal>

    <MethodNodeTerminal name="Previous Data Value">
      <DataType>I32</DataType>
      <Direction>Read</Direction>
      <TerminalOrder>1</TerminalOrder>
    </MethodNodeTerminal>

  </MethodNodeTerminalList>

  <NodeIcon>AI</NodeIcon>

  <MethodVIScriptInfo name="CustomChannelMethodScriptInfo">

    <VIList>
```

Method nodes can have zero, one, or multiple terminals.

```
        <VI name="MDK-9902_CustomChannelMethod.vi">
          <SequenceOrder>0</SequenceOrder>

          <TerminalConnectionList>

            <TerminalConnection name="FirstConnection">
              <NodeTerminalName>New Data Value</NodeTerminalName>
              <ConnectorPaneTerminal>0</ConnectorPaneTerminal>
            </TerminalConnection>

            <TerminalConnection name="SecondConnection">
              <NodeTerminalName>Previous Data Value</NodeTerminalName>
              <ConnectorPaneTerminal>4</ConnectorPaneTerminal>
            </TerminalConnection>

          </TerminalConnectionList>

        </VI>

        <VI name="MDK-9902_GeneralNodeError.vi">
          <SequenceOrder>1</SequenceOrder>
          <ErrorHandling>true</ErrorHandling>
        </VI>

      </VIList>
    </MethodVIScriptInfo>
  </Interface>

</MethodNodeInterfaceList>

<PropertyNodeInterfaceList>

  <Interface name="Data Value">

    <DataType>I32</DataType>
    <Direction>BiDirectional</Direction>
    <DefaultDirection>Read</DefaultDirection>
    <NodeIcon>AI</NodeIcon>

    <ReadVIScriptInfo name="CustomChannelReadPropVIInfo">
```

The PropertyNodeInterfaceList contains information about the property node interfaces that were specified for the module, module SubItems, and I/O channels.

The name attribute of the Property Node interface is shown in the property node on the LabVIEW block diagram.

A property node can be Read, Write, or BiDirectional.

When the direction is BiDirectional, the DefaultDirection tag must be specified.

Read and BiDirectional property nodes must specify ReadVIScriptInfo.

```
      <VIList>

        <VI name="MDK-9902_CustomChannelReadProperty.vi">
          <SequenceOrder>0</SequenceOrder>
          <VIHasTerminalConnection>true</VIHasTerminalConnection>
        </VI>

        <VI name="MDK-9902_GeneralNodeError.vi">
          <SequenceOrder>7</SequenceOrder>
          <ErrorHandling>true</ErrorHandling>
        </VI>

      </VIList>

    </ReadVIScriptInfo>

    <WriteVIScriptInfo name="CustomChannelWritePropVIInfo">

      <VIList>

        <VI name="MDK-9902_CustomChannelWriteProperty.vi">
          <SequenceOrder>0</SequenceOrder>
          <VIHasTerminalConnection>true</VIHasTerminalConnection>
        </VI>

        <VI name="MDK-9902_GeneralNodeError.vi">
          <SequenceOrder>7</SequenceOrder>
          <ErrorHandling>true</ErrorHandling>
        </VI>

      </VIList>

    </WriteVIScriptInfo>

  </Interface>

</PropertyNodeInterfaceList>


<MergedIONodeVIScriptInfoList>
  <MergedIONodeVIScriptInfo name="AnalogMergedVIScriptInfo">
```

Write and BiDirectional Property Nodes must specify WriteVIScripInfo.

The name attribute of the MergedIONodeVIScriptInfo is only used within the XML file.

```
    <VIScriptInfoName>AIChanVIScriptInfo</VIScriptInfoName>
    <VIScriptInfoName>AOChanVIScriptInfo</VIScriptInfoName>
  </MergedIONodeVIScriptInfo>
</MergedIONodeVIScriptInfoList>

<ModuleModeDefinition>

  <NormalOperationMode>

    <SPIConfiguration>
      <SPIHalfTauTicks>10</SPIHalfTauTicks>
    </SPIConfiguration>

    <ConvertPulseConfiguration>
      <ConvertPulseWidth>Long</ConvertPulseWidth>
    </ConvertPulseConfiguration>

    <DoneWaitConfiguration>
      <DoneWaitTimeoutTicks>500</DoneWaitTimeoutTicks>
    </DoneWaitConfiguration>

    <DigitalLines>
      <DIO0>BiDirectional</DIO0>
      <DIO1>Unused</DIO1>
    </DigitalLines>

  </NormalOperationMode>


  <AuxiliaryCommunicationMode>

    <SPIConfiguration>
      <SPIHalfTauTicks>15</SPIHalfTauTicks>
    </SPIConfiguration>

    <DigitalLines>
      <DIO0>OutputOnly</DIO0>
      <DIO1>InputOnly</DIO1>
      <DIO2>ConstantOutputHigh</DIO2>
      <DIO3>ConstantOutputLow</DIO3>
    </DigitalLines>
```

The MergedIONodeVIScriptInfo contains multiple VIScriptInfoName tags. These tags refer to the name attribute that was set on the VIScriptInfo sections of the I/O node interfaces. All VIScriptInfo tags listed here must be compatible.

The ModuleModeDefinition section contains information about the module hardware and what modes it supports.

Defining the NormalOperationMode section in the XML enables that mode for the module.

Defining the SPIConfiguration section in the XML enables SPI for that mode for the module.

Defining the ConvertPulseConfiguration section in the XML enables the Pulse Convert Method Node of the Timing interface for that mode for the module.

Defining the DoneWaitConfiguration section of the XML enables the Wait on Done Method Node of the Timing interface for that mode for the module.

All DIO lines that are not reserved by the functions that are enabled by that mode are not available for DIO by default. The DigitalLines section is used to specify functionality of DIO lines in that mode.

Defining the AuxiliaryCommunicationMode section in the XML enables that mode for the module.

```
    </AuxiliaryCommunicationMode>

    <DigitalLineInfo>

      <SCTLOutputSyncRegs>
        <DIO0>1</DIO0>
        <DIO1>0</DIO1>
        <DIO2>1</DIO2>
        <DIO3>0</DIO3>
      </SCTLOutputSyncRegs>

      <Arbitration>
        <DIO0>NeverArbitrate</DIO0>
        <DIO1>AlwaysArbitrate</DIO1>
        <DIO2>ArbitrateIfMultipleRequestorsOnly</DIO2>
        <DIO3>NeverArbitrate</DIO3>
        <DIO4>NeverArbitrate</DIO4>
        <DIO5>NeverArbitrate</DIO5>
        <DIO6>NeverArbitrate</DIO6>
        <DIO7>NeverArbitrate</DIO7>
        <DIO8>NeverArbitrate</DIO8>
        <DIOPort>NeverArbitrate</DIOPort>
      </Arbitration>

    </DigitalLineInfo>

  </ModuleModeDefinition>


</ModuleSupport>
```

The SCTLOutputSyncRegs section is used to specify whether or not a sync register is placed on the output of a DO line when used inside of an SCTL. This section is optional and all lines default to one sync register when the section is omitted.

The Arbitration section is used to specify the arbitration option that is used when multiple DO nodes are placed on the LabVIEW block diagram. This section is optional and all lines default to NeverArbitrate when the section is omitted.

C

Using MDK with cRIO-904x Controllers

cRIO-904x Controllers

cRIO-904x is a new family of controllers that support different program modes on a slot by slot basis. The program modes include Real-Time (DAQmx), Real-Time Scan (RSI), and LabVIEW FPGA (FPGA I/O Nodes).

MDK only operates in LabVIEW FPGA mode. If a module occupies a slot that is not configured for LabVIEW FPGA, you will not be able to interact with that module using MDK.

MDK 1.5

cRIO-904x controllers support MDK 1.5. Behavior for MDK 1.5 differs between cRIO-904x controllers and non-904x controllers. For more information on using MDK 1.5 with cRIO-904x controllers, refer to the *cRIO Module Developers Kit* board at `forums.ni.com`.

MDK 2

cRIO-904x controllers support MDK 2. NI-RIO 17.6 releases with MDK 2.1 which enables MDK 2 module support on cRIO-904x targets. If you do not upgrade module support from MDK 2.0 to MDK 2.1, your module will not work on cRIO-904x targets. However, your module will continue to work on non-904x targets. Refer to the following sections for behavioral differences between MDK 2.0 and MDK 2.1 and for behavioral differences between cRIO-904x targets and non-904x targets.

Differences between MDK 2.0 and MDK 2.1

The C Series Communication Core MDK API contains a **Check Module Status** method node which outputs an enum indicating the current status of the module. The previous values of this enum were **Unknown**, **Correct**, **Incorrect**, **No Module**, and **Invalid**. Module support VIs may use this node in the module resource VI or may use the wrapper VI from the MDK API palette.

The C**heck Module Status** node has been updated for MDK 2.1 to output the new **Incorrect Program Mode** value. This value will appear when the program mode for a slot is not configured for LabVIEW FPGA. The Incorrect Program Mode value will never appear on non-904x targets.

Pre-existing module support will show a deprecated version of the module status enum in NI-RIO 17.6. The wrapper VI from the MDK API palette will have a red line through the VI icon to indicate the VI is deprecated.

Using MDK 2 on cRIO-904x Controllers

The C Series Communication Core in MDK 2 exhibits different behavior between cRIO-904x targets and non-904x targets. Refer to the following sections for more information on using MDK 2 with cRIO-904x targets.

Delayed Output Enable Direction Change

Output Enable changes require a typical delay of 500 ns. In rare cases, changes could require a maximum delay of 1.75 µs. This affects the amount of time Set Output Enable method nodes and Change Mode commands will take to complete. If your module relies on either of those operations to complete within a fixed amount of time, you may encounter issues.

Loss of Direct Control of ID Select Line

You cannot control the ~ID_SELECT line through LabVIEW FPGA. You can read the status of the ~ID_SELECT line using the Debug Register in Development Mode. Reading the ~ID_SELECT line status on the Debug Register requires a typical delay of 500 ns. In rare cases, reading the status could require a maximum delay of 1.75 µs. If your module relies on reading ~ID_SELECT line within a fixed amount of time, try adding delays to the MDK VIs.

Tighter Timing Constraints

Tighter I/O timing constraints may cause failure in some modules that currently meet timing requirements. NI expects developers to redo timing analysis with the new constraints and make necessary changes. Module support tested on cabled expansion chassis should continue to work on the new controllers.

Increased size of I/O Node on a per slot basis

All I/O Nodes increase by a constant adder of approximately 1.0% of a Series 7 Kintex 70T (XC7K70T) on a slot by slot basis. If all eight slots use MDK, the available FPGA space could reduce by approximately 10%. A VI using over 90% of the LUTs on a cRIO-9035 may fail compilation on a 904x target with an equivalent FPGA due to the increased complexity of the I/O Node.

Module Insertion and Removal

The C Series Communication Core will take longer to respond to module insertion and removal events. When a module is removed from the chassis, the C Series Communication Core could take up to half a second to report the module status as **No Module**. This may result in invalid data being returned to the user during this time.

Upgrading Module Support

Follow these steps to upgrade MDK support from 2.0 to 2.1:

1. Update the MDK version tag in the Module Support XML file to `2.1`.
2. Regenerate the Development Mode export of your module using the Module Support Export utility.
3. Replace any instances of the deprecated Module Status wrapper VI with the new VI, which can be found in the MDK API palette.

Note If the VI is inside a SCTL, make sure to use the SCTL version of the new Module Status VI instead.

Note Replace instances of the deprecated Check Module Status method node with the new method node if you are not using the Module Status VI from the MDK API palette to read the module status.

4. Update your module support VIs to handle the **Incorrect Program Mode** module status.

Note NI recommends handling the **Incorrect Program Mode** status the same as the **No Module** status. However, if your module resource VI generates error codes, it should output error code **65673** rather than **65536** when the status is **Incorrect Program Mode**. For an example, examine `MDK-9901_ModuleResource.vi` of the MDK-9901 shipping example from the CompactRIO MDK installer.

5. Validate that the module works in Development Mode on a cRIO-904x target and a non-904x target.
6. Regenerate the Release Mode export of your module using the Module Support Export utility.
7. Validate that the module works in Release Mode on both a cRIO-904x target and a non-904x target.
8. Verify that you are not running into timing issues on the new target.
9. Ship your new module support. The new support will retain compatability with all other chassis, but will not work in CompactRIO Device Drivers versions prior to 17.6.

D

NI Services

NI provides global services and support as part of our commitment to your success. Take advantage of product services in addition to training and certification programs that meet your needs during each phase of the application life cycle; from planning and development through deployment and ongoing maintenance.

To get started, register your product at ni.com/myproducts.

As a registered NI product user, you are entitled to the following benefits:

- Access to applicable product services.
- Easier product management with an online account.
- Receive critical part notifications, software updates, and service expirations.

Log in to your MyNI user profile to get personalized access to your services.

Services and Resources

- **Maintenance and Hardware Services**—NI helps you identify your systems' accuracy and reliability requirements and provides warranty, sparing, and calibration services to help you maintain accuracy and minimize downtime over the life of your system. Visit ni.com/services for more information.
  - **Warranty and Repair**—All NI hardware features a one-year standard warranty that is extendable up to five years. NI offers repair services performed in a timely manner by highly trained factory technicians using only original parts at an NI service center.
  - **Calibration**—Through regular calibration, you can quantify and improve the measurement performance of an instrument. NI provides state-of-the-art calibration services. If your product supports calibration, you can obtain the calibration certificate for your product at ni.com/calibration.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. To learn more, call your local NI office or visit ni.com/alliance.

- **Training and Certification**—The NI training and certification program is the most effective way to increase application development proficiency and productivity. Visit ni.com/training for more information.
  - The Skills Guide assists you in identifying the proficiency requirements of your current application and gives you options for obtaining those skills consistent with your time and budget constraints and personal learning preferences. Visit ni.com/skills-guide to see these custom paths.
  - NI offers courses in several languages and formats including instructor-led classes at facilities worldwide, courses on-site at your facility, and online courses to serve your individual needs.
- **Technical Support**—Support at ni.com/support includes the following resources:
  - **Self-Help Technical Resources**—Visit ni.com/support for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on. Registered users also receive access to the NI Discussion Forums at ni.com/forums. NI Applications Engineers make sure every question submitted online receives an answer.
  - **Software Support Service Membership**—The Standard Service Program (SSP) is a renewable one-year subscription included with almost every NI software product, including NI Developer Suite. This program entitles members to direct access to NI Applications Engineers through phone and email for one-to-one technical support, as well as exclusive access to online training modules at ni.com/self-paced-training. NI also offers flexible extended contract options that guarantee your SSP benefits are available without interruption for as long as you need them. Visit ni.com/ssp for more information.
- **Declaration of Conformity (DoC)**—A DoC is our claim of compliance with the Council of the European Communities using the manufacturer's declaration of conformity. This system affords the user protection for electromagnetic compatibility (EMC) and product safety. You can obtain the DoC for your product by visiting ni.com/certification.

For information about other technical support options in your area, visit ni.com/services, or contact your local office at ni.com/contact.

You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office websites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.