Important Information

**Warranty**
The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it. EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER’S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THEREFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR INDIRECT DAMAGES, PROFITS, OR LOSSES, OR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, USER IF A DEFECT IN THE PRODUCT OR OTHER ERRORS THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner’s failure to follow the National Instruments installation, operation, or maintenance instructions; owner’s modification of the product; owner’s abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

**Copyright**
Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

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

**Trademarks**
National Instruments, NI, ni.com, and LabVIEW are trademarks of National Instruments Corporation. Refer to the Terms of Use section on ni.com/legal for more information about National Instruments trademarks. Members of the National Instruments Alliance Partner Program are business entities independent from National Instruments and have no agency, partnership, or joint-venture relationship with National Instruments.

**Patents**
For patents covering National Instruments products, refer to the appropriate location: Help>Patents in your software, the patents.txt file on your CD, or ni.com/patents.

**WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS**

1. NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

2. IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREFORER COLLECTIVELY TERMED “SYSTEM FAILURES”). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONICS SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS’ TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.
The following conventions are used in this manual:

`<>` Angle brackets that contain numbers separated by an ellipsis represent a range of values associated with a bit or signal name—for example, AO <3..0>.

`>` The » symbol leads you through nested menu items and dialog box options to a final action. The sequence File»Page Setup»Options directs you to pull down the File menu, select the Page Setup item, and select Options from the last dialog box.

💡 This icon denotes a tip, which alerts you to advisory information.

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

⚠️ This icon denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.

📂 This icon denotes a directory path.

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

*italic* Italic text denotes variables, emphasis, or a cross-reference. Italic text also denotes text that is a placeholder for a word or value that you must supply.

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

`monospace bold` Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.

`monospace italic` Italic text in this font denotes text that is a placeholder for a word or value that you must supply.
Contents

Chapter 1
Introduction
Taking Your Graphical Design to Any 32-Bit Microprocessor ........................................ 1-1
Porting the LabVIEW Run-Time Library .......................................................................... 1-2
Incorporating I/O ........................................................................................................... 1-3
Creating the Target in LabVIEW and Incorporating Your Toolchain ......................... 1-3
Customizing the LabVIEW Environment ..................................................................... 1-3
Understanding the LabVIEW Directory Hierarchy and Naming Conventions ............. 1-4
LabVIEW Directory Hierarchy ...................................................................................... 1-4
Target Naming Conventions ........................................................................................... 1-5

Chapter 2
Example Targets
Selecting an Appropriate Example Target ........................................................................ 2-1
Example Targets System Requirements .......................................................................... 2-3
Setting Up the Example Targets ..................................................................................... 2-4
Axiom CMD565 Example Target ................................................................................... 2-4
VxWorks ......................................................................................................................... 2-4
eCos ............................................................................................................................... 2-11
PHYTEC Example Target ............................................................................................... 2-17
Connections .................................................................................................................. 2-17
Installing the RedBoot Bootloader .............................................................................. 2-18
Using the GNU Toolchain ............................................................................................ 2-19
Spectrum Digital DSK6713 Example Target .................................................................. 2-19
Using the TI Code Composer Studio Toolchain ............................................................... 2-19

Chapter 3
Using the Target Editor to Manage Embedded Targets
Creating a New Target ..................................................................................................... 3-1
Modifying an Existing Target ......................................................................................... 3-2
Configuring Target Properties ....................................................................................... 3-3
Configuring General Items ............................................................................................. 3-3
Defining Shortcut Menu Items ....................................................................................... 3-4
Defining Categories ......................................................................................................... 3-4
Defining Timing Sources ............................................................................................... 3-5
Defining Events ............................................................................................................... 3-5
Defining Signed Files ..................................................................................................... 3-6
Configuring Licensing .................................................................................................... 3-6
Configuring Build Specification Type Properties ............................................................ 3-7
Configuring General Build Specification Items .......................................................... 3-7
Defining Build Specification Shortcut Menu Items .................................................... 3-8
Defining Build Specification Categories ........................................................................ 3-8
Defining Build Specification Events .............................................................................. 3-9
Creating New Plug-In or Subpanel VIs ................................................................. 3-9
Creating New Plug-In VIs ........................................................................................ 3-9
Creating New Subpanel VIs .................................................................................... 3-10

Chapter 4
Defining Code Generation Options

Code Generation Attributes ............................................................................................ 4-2
BigEndian ......................................................................................................................... 4-2
DestinationFolder ............................................................................................................. 4-2
GenerateCFunctionCalls ............................................................................................... 4-2
GenerateDebugInfo ............................................................................................................ 4-3
GenerateGuardCode ........................................................................................................... 4-3
GenerateInlinePrintf ....................................................................................................... 4-4
GenerateIntegerOnly ......................................................................................................... 4-4
GenerateLibraryInterface ............................................................................................... 4-4
GenerateSerialOnly ......................................................................................................... 4-5
IncrementalBuild ............................................................................................................ 4-6
interruptServiceRoutineVIs .......................................................................................... 4-6
MemoryModel ................................................................................................................. 4-7
OCDIComments ............................................................................................................. 4-7
Silent ............................................................................................................................... 4-7
TargetAlignment ............................................................................................................. 4-8
UseStackVariables ........................................................................................................... 4-8
Generating the Fastest Code ........................................................................................... 4-8
Signal Naming Convention ............................................................................................ 4-9

Chapter 5
Defining Compiler Directives

General Macros ............................................................................................................... 5-2
Operating System Macros .............................................................................................. 5-2
Feature Macros ................................................................................................................ 5-3
Memory Mapping Macros ............................................................................................. 5-4
Debugging Macros ......................................................................................................... 5-4
VxWorks-Only Macros ................................................................................................... 5-4
Chapter 6
Configuring the Linking of Object Files and Libraries

Chapter 7
Porting the LabVIEW Embedded Run-Time and Analysis Libraries
Source Code

Directory Structure ................................................................. 7-2
Portable Pieces of the Run-Time Library ................................. 7-2
LVEstimatedMain.c ............................................................... 7-4
Include Files .............................................................................. 7-4
- LVsysIncludes.h ............................................................... 7-5
- LVDels_plat.h .................................................................. 7-5
  #defines ............................................................................ 7-5
- Feature Flags .................................................................. 7-6
OS-Specific Components .......................................................... 7-6
- Critical Sections: LVCritSect.c ............................................. 7-7
- Events: LVEvent.c ............................................................ 7-7
- Threads: LVThreads.c ....................................................... 7-8
- Non-Blocking Operations: LVNBOps.c .............................. 7-9
- LabVIEW-Based Interrupt Service Routines: OEM_LVISR.c .... 7-9
- Printf: PDAStrSupport_OS.c .............................................. 7-10
- Time: CCGTimeSupport_OS.c ............................................ 7-10
- Serial: PlatformSerial_OS.c ............................................. 7-11
- CAN: PlatformCAN_OS.c .................................................. 7-12
- TCP/UDP: CCGNetConnSupport.c ..................................... 7-13
- Static Memory Allocation Support ..................................... 7-14
- Analysis Library ............................................................... 7-14

Chapter 8
Using FuncList.dat to Improve Performance

Adding Entries to FuncList.dat .................................................. 8-2
Format of FuncList.dat ......................................................... 8-2
- 1st Line—VI Names ......................................................... 8-2
- 2nd Line—C Code ........................................................... 8-2
  - Passing Parameters Directly ......................................... 8-3
  - Passing Parameters by ArgList .................................... 8-3
- 3rd Line—Feature Section ................................................ 8-3
Debugging ................................................................................ 8-4
Troubleshooting .................................................................... 8-4
Example ................................................................................ 8-4
Chapter 9
Elemental I/O

Overview ....................................................................................................................... 9-1
I/O Devices ................................................................................................................. 9-1
Elemental I/O Classes ............................................................................................... 9-2
Pins .............................................................................................................................. 9-2
Resources ..................................................................................................................... 9-2
Plug-In VIs ............................................................................................................... 9-2

How Users Use Elemental I/O Nodes .......................................................................... 9-3

Creating Elemental I/O .................................................................................................. 9-3
Creating Elemental I/O Plug-In VIs........................................................................ 9-3
I/O Node Implementation Plug-In VI........................................................................ 9-4
I/O Property Node Implementation Plug-In VIs ...................................................... 9-4
I/O Specify Node Plug-In VI..................................................................................... 9-5
I/O Project and Node Attribute Validation Plug-In VIs ........................................ 9-5

Creating Elemental I/O Devices, Classes, Resources, and Pins ............................... 9-5
Adding Elemental I/O Devices to an Embedded Target .......................................... 9-7

Example......................................................................................................................... 9-7

Chapter 10
Implementing Interrupt Service Routine (ISR) Support

ISR Functions ............................................................................................................... 10-1
InitOEMISRs ........................................................................................................... 10-1
UninitOEMISRs ....................................................................................................... 10-1
OEMISRRegisterHandler ...................................................................................... 10-2
OEMISRUnregisterHandler .................................................................................... 10-2

Tips for Supporting ISRs on a New Platform .............................................................. 10-2

Chapter 11
Timing

Using the Target Editor to Define Timing Sources................................................. 11-1
CreateInternalTSource() ....................................................................................... 11-2
FireInternalTSource() ......................................................................................... 11-2
DeleteInternalTSource() ...................................................................................... 11-3

Example......................................................................................................................... 11-3

Using the Timed Loop VIs to Define Timing Sources ............................................. 11-3
Create External Timing Source VI................................................................. 11-3
Fire External Timing Source VI................................................................. 11-4
Delete External Timing Source VI.............................................................. 11-4
Chapter 12
Memory Mapping
How Users Use Memory Mapping .............................................................. 12-1
Implementing Memory Mapping ................................................................. 12-1
LEP_\_SectionNames.vi ........................................................................... 12-2
LEP\_MemoryMap_Default.vi ................................................................. 12-2
LEP\_MemoryMap_Cmp.vi ................................................................... 12-2
LEP\_MemoryMap.vi ........................................................................... 12-2
LEP\_MemoryMap_Query.vi ................................................................. 12-3

Chapter 13
Embedded Automated Tests
Configuration Files ................................................................................. 13-2
Test Sequences ....................................................................................... 13-2
Example .................................................................................................. 13-3

Chapter 14
Static Memory Allocation
Supporting a Static Memory Model ......................................................... 14-1
Limitations of the Static Memory Model ................................................ 14-2
Limitations for End-Users ..................................................................... 14-2
Limitations for OEMs ........................................................................... 14-4

Chapter 15
Instrumented Debugging
Implementing the Instrumented Debugging Communication Layer for LabVIEW .... 15-1
LEP\_Debug.vi ...................................................................................... 15-1
Implementing the Host Plug-In VIs ......................................................... 15-2
Open VI .............................................................................................. 15-2
Close VI .............................................................................................. 15-3
Read VI .............................................................................................. 15-3
Write VI .............................................................................................. 15-3
Implementing the Instrumented Debugging Communication Layer for Your Embedded Target ........................................................................... 15-4
Connect ............................................................................................... 15-5
Disconnect ........................................................................................ 15-5
Write ................................................................................................. 15-5
Read ................................................................................................. 15-5
Bytes Available ................................................................................ 15-5
Chapter 16
On Chip Debugging

.lvm File...................................................................................................................... 16-2
C Parser ......................................................................................................................... 16-4
C Compiler .................................................................................................................... 16-5
List Parser .................................................................................................................... .. 16-5
Linker ............................................................................................................................ 16-8
.map Parser .................................................................................................................... 16-8
Debug Database............................................................................................................. 16-9
Debug Daemon.............................................................................................................. 16-9
Emulator Plug-in ........................................................................................................... 16-9
Adding Support for an Existing Emulator to a New Target .......................................... 16-10
Adding the List Parser for a Non-GCC Compiler ......................................................... 16-11
EMB_Debug_Map_x_UpdateDD.vi ................................................................................. 16-11
EMB_Debug_Map_x_ResolveSymbols.vi ........................................................................ 16-11
Resolving variable.member (type_code) Signals ......................................................... 16-11
Adding Support for a New Emulator Interface ............................................................. 16-14
OCDI Debugging Plug-In VIs....................................................................................... 16-14
Init............................................................................................................................... 16-14
IRelease...................................................................................................................... 16-16
ISetBreakpoints........................................................................................................... 16-17
IClearBreakpoints .................................................................................................. 16-19
IGetBreakpoint........................................................................................................... 16-20
IContExecution .......................................................................................................... 16-21
IREsolveSymbols...................................................................................................... 16-22
IMemoryRead .............................................................................................................. 16-24
IMemoryWrite ............................................................................................................ 16-25
IGo .............................................................................................................................. 16-26
IStop.......................................................................................................................... 16-28
IDownload................................................................................................................... 16-29
IConfig....................................................................................................................... 16-30
IExit............................................................................................................................ 16-31
Example..................................................................................................................... 16-33
Chapter 17
Integrating LabVIEW with Another IDE

Plug-In VIs .............................................................. 17-2
InitIndication ......................................................... 17-3
IGetIndication ......................................................... 17-4
IShowSource .......................................................... 17-6
IShowCrash ........................................................... 17-7

Example Implementation ............................................ 17-9
Installing the Necessary Tools for LabVIEW and Eclipse Integration .... 17-9
Eclipse ................................................................. 17-9
CDT SDK .................................................................. 17-9
LabVIEW Embedded Eclipse Plug-In ........................... 17-10
Cygwin 1.15.18-1 or 1.15.21 ..................................... 17-10

The Implementation .................................................. 17-10
Debugging Backend ................................................ 17-11
Target Properties Dialog Box .................................... 17-11

Chapter 18
Configuring the Target Syntax

Adding Target Syntax to an Embedded Target .................. 18-1
Modifying the Target Syntax for an Embedded Target ......... 18-4

Appendix A
Technical Support and Professional Services
Introduction

Graphical system design has emerged as a new approach to embedded system design and allows you to design algorithms using interactive design tools for control, signal processing, and advanced mathematics. You then can use your code with off-the-shelf hardware for functional prototypes. Graphical system design also is scalable enough to target your custom hardware for higher volume deployments. Refer to the National Instruments Web site at ni.com/design to learn more about the LabVIEW embedded design and prototyping platform.

Taking Your Graphical Design to Any 32-Bit Microprocessor

When you are ready to deploy your embedded design, you can use the LabVIEW Embedded Development Module along with a third-party toolchain and an embedded operating system to extend the graphical LabVIEW programming experience to any 32-bit microprocessor.

The LabVIEW C Code Generator, which is a component of the Embedded Development Module, creates ANSI C code from a LabVIEW block diagram. You also can add pre-existing C code you might have. When you build a block diagram into an embedded application, LabVIEW traverses the block diagram and generates simple C primitives if possible. For example, the LabVIEW C Code Generator converts While Loops to while() statements and converts the Add function to a simple C + operation. However, a straight mapping is not possible for more complex functions so the LabVIEW C Code Generator uses the LabVIEW C Run-Time Library, which is analogous to the LabVIEW Run-Time Engine in LabVIEW for Windows. An important part of porting LabVIEW to a new target involves porting the LabVIEW C Run-Time Library, which contains such things as communication, data manipulation, timing functions, and so on. The Embedded Development Module includes the source code for the LabVIEW C Run-Time Library.

The code that the LabVIEW C Code Generator generates passes through a cross-compiler. If you add pre-existing C code, the extra C files you provide also pass through the cross-compiler and are linked into an executable you
can run on the embedded device, which is called a target in LabVIEW. You implement how this executable downloads, or deploys, to the correct memory location and begins running on the embedded target through standard communication protocols. JTAG emulator, RS-232, and Ethernet are common ways to handle the communication between LabVIEW and an embedded target, but you can use other communication protocols.

You also can implement instrumented or on-chip debugging so you can use LabVIEW to debug an embedded application you build with LabVIEW.

Refer to the *LabVIEW Embedded Development Module Release Notes*, available by selecting **Start**>**All Programs**>**National Instruments**>**LabVIEW**>**LabVIEW Manuals** and opening EMB_Release Notes.pdf, for information about target recommendations and required prerequisite knowledge to port LabVIEW to a new target. The following main steps are involved in porting, each of which is described in the following sections:

- Porting the LabVIEW Run-Time Library
- Incorporating I/O
- Creating the Target in LabVIEW and incorporating your toolchain
- Customizing the LabVIEW environment

**Porting the LabVIEW Run-Time Library**

Each LabVIEW function that does not map directly to a C primitive is implemented using a function call. These functions are implemented in the LabVIEW Run-Time Library on top of a lower layer of generic functions that handle memory movement, string manipulation, timing, and so on.

You must provide the mapping for these generic, low-level function calls to functions your target supports. This mapping allows your toolchain to compile the generated C code for your target platform. A major component of porting the LabVIEW Run-Time Library involves providing these mappings.

Refer to Chapter 7, *Porting the LabVIEW Embedded Run-Time and Analysis Libraries Source Code*, for more information about porting the LabVIEW Embedded Run-Time Library.
Incorporating I/O

In traditional embedded C programming, analog and digital I/O is typically done at the register level. Most data acquisition must be done by direct memory access. The Embedded Development Module introduces a way to abstract the low-level I/O implementation in a unified API called Elemental I/O. Although you still need to include the register-level access when you implement the Elemental I/O Nodes, you only have to do it once per target platform.

Refer to Chapter 9, *Elemental I/O*, for information about implementing Elemental I/O.

Creating the Target in LabVIEW and Incorporating Your Toolchain

Managing an embedded target in LabVIEW involves creating and editing the TgtSupp.xml file, which LabVIEW uses to incorporate your target implementation into the LabVIEW development environment. You use the Target Editor to manage targets you create with the Embedded Development Module.

Refer to Chapter 3, *Using the Target Editor to Manage Embedded Targets*, for information about using the Target Editor to create your target support for LabVIEW.

Customizing the LabVIEW Environment

Customizing the LabVIEW environment for your target makes your target easier to use in LabVIEW, which is even more important if you plan on reselling your target support for LabVIEW. Refer to the *LabVIEW Embedded Development Module Target Distribution Guide*, available by selecting Start»All Programs»National Instruments»LabVIEW»LabVIEW Manuals and opening EMB_Distribution_Guide.pdf, for more information about customizing LabVIEW.
Understanding the LabVIEW Directory Hierarchy and Naming Conventions

Understanding the LabVIEW directory hierarchy and naming conventions is fundamental to successfully creating new LabVIEW embedded targets.

LabVIEW Directory Hierarchy

You do not need to be familiar with the entire labview directory structure, but you need to be aware of the following directories:

- **autotest**—Contains the embedded autotest framework and test VIs. The tests you run as part of the embedded autotests are in the tests\Large semi-auto tests subdirectory.
- **CCodeGen**—Contains FuncList.dat and the following subdirectories:
  - **analysis**—Contains the source for the analysis library.
  - **build**—Contains build scripts and makefiles. The Unix Console target is the only example target in this directory. The build scripts and the makefiles for the other embedded targets are located in the labview\Targets\NI\Embedded directory.
  - **Include**—Contains the include files for the run-time library. This directory contains several subdirectories, but the \os subdirectory is the directory you need to be familiar with.
    - **os**—Contains subfolders for the operating systems the example targets use. Each subfolder contains only header files. Do not put C files in this directory. Instead, put the associated C files in the labview\CCodeGen\libsrc\os directory.
  - **libsrc**—Contains the same directory structure as the \Include subdirectory. Like the \Include subdirectory, this directory contains several subdirectories, but the \os subdirectory is the directory you need to be familiar with.
    - **os**—Contains subfolders for the operating systems the example targets use. Each subfolder contains only C files. Do not put header files in this directory. Instead, put the associated header files in the labview\CCodeGen\Include\os directory.
  - **examples**—Contains examples. The \lvemb subdirectory contains some embedded-specific examples for working with interrupts and the Inline C Node.
  - **help**—Contains help files.
– manuals—Contains documentation in PDF format.
– readme—Contains readme files for LabVIEW and any additional modules or toolkits.
– Targets\NI\Embedded—Contains the target directory folders for NI embedded targets. The target directory organizes targets into a hierarchy for greater reuse of target implementation code. This hierarchy organizes targets by operating systems. The Embedded Development Module target implementations—which contain the plug-in VIs, libraries, helper scripts, and other files required to implement a target—are located in this directory. The target directory hierarchy is OS-centric and supports sub-targeting. The top-level targets are self-contained and do not rely on any sub-target implementations below the sub-target. In contrast, the subtargets rely on the top-level target implementation.
  • eio—Contains Elemental I/O implementations for embedded targets.

Note LabVIEW does not recognize targets outside of the labview\Targets\<company name>\Embedded directory. You must have an Embedded subdirectory under your company name directory.

• vi.lib—Contains libraries of built-in VIs. You need to be familiar with the following subdirectories:
  – Embedded—Contains VIs that run on the target when a user builds an embedded VI into an embedded application. Use this directory for the VIs you want to add to the Functions palette. Create a subdirectory for your target.
  – LabVIEW Targets\Embedded—Contains the plug-in VIs and any utility or helper VIs. The VIs in this directory are not intended to be on the Functions palette or used by end users.

Target Naming Conventions

National Instruments recommends, but does not require, you follow the same naming convention as the Embedded Development Module example targets. The example targets have the following naming convention:

<Hardware target>,<OS> <Hardware Variant>

For example, Axiom CMD565, eCos ROM. Do not use underscores in the target name.
Example Targets

The LabVIEW Embedded Development Module includes several example targets. Use the example targets as a starting point when you create new embedded targets. Example targets are located in the following directory:

`labview\Targets\NI\Embedded`

Selecting an Appropriate Example Target

It is important to select an appropriate example target to use as a template when you create a new embedded target. Use the target that is closest to the target and toolchain you are creating. If you are using a GNU C/C++-based (gcc) toolchain, consider using an eCos target. If you are using a VxWorks-based toolchain with different hardware, you might want to use a VxWorks subtarget. Subtargets are targets that reuse existing functionality from another target.

The Embedded Development Module also includes a blank template target. This target is not intended to be an implementation example, but the target can serve as a good starting point when none of the example targets are appropriate. National Instruments recommends you use the blank target when porting LabVIEW to a new operating system.

None of the example targets are meant to be fully featured, ready to use targets. The different example targets have different implementations. When you are implementing a feature for a new embedded target, look for an existing implementation in an existing target that might be similar to your target.

The following table lists some of the implementation features for the example targets. Use this table to find an example of a feature you are implementing for your target.
## Example Targets

<table>
<thead>
<tr>
<th>Target Name</th>
<th>Instrumented Debugging</th>
<th>On Chip Debugging</th>
<th>Pre-Built Run-Time Library</th>
<th>Static Memory Model</th>
<th>Memory Mapping</th>
<th>Elemental I/O</th>
<th>IDE Integration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Generation Only</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Axiom CMD565, eCos ROM Image</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Axiom CMD565, eCos RAM Image</td>
<td>Serial</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Unix Console</td>
<td>TCP</td>
<td>Eclipse</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Simulated</td>
<td>Eclipse</td>
</tr>
<tr>
<td>Axiom CMD565, VxWorks RAM Image</td>
<td>No</td>
<td>iSYSTEM ic3000</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Axiom CMD565, VxWorks ROM Image</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Axiom CMD565, VxWorks Module</td>
<td>Serial</td>
<td>Wind River WTX</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>VxWorks Simulation</td>
<td>TCP</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Windows Console Application</td>
<td>TCP</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Simulated</td>
<td>No</td>
</tr>
<tr>
<td>PHYTEC LPC229x, eCos</td>
<td>Serial</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Spectrum Digital DSK6713, DSP/BIOS</td>
<td>RTDX</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
### Example Targets System Requirements

The Embedded Development Module example targets have the following requirements:

<table>
<thead>
<tr>
<th>Target Name</th>
<th>Hardware Requirements</th>
<th>Software Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Generation Only</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Axiom CMD565, eCos ROM Image</td>
<td>Axiom CMD-565 Development Board</td>
<td>Cygwin 1.5.x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>eCos 2.0 PowerPC toolchain</td>
</tr>
<tr>
<td>Axiom CMD565, eCos RAM Image</td>
<td>Axiom CMD-565 Development Board</td>
<td>Cygwin 1.5.x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>eCos 2.0 PowerPC toolchain</td>
</tr>
<tr>
<td>Unix Console</td>
<td>None</td>
<td>Cygwin 1.5.x with gcc package</td>
</tr>
<tr>
<td>Axiom CMD565, VxWorks RAM Image</td>
<td>Axiom CMD-565 Development Board</td>
<td>Wind River Tornado 2.2.1</td>
</tr>
<tr>
<td></td>
<td>iSYSTEM iC3000ActiveEmulator</td>
<td>VxWorks 5.5.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BSP for CMD565</td>
</tr>
<tr>
<td></td>
<td></td>
<td>iSYSTEM winIDEA</td>
</tr>
<tr>
<td>Axiom CMD565, VxWorks ROM Image</td>
<td>Axiom CMD-565 Development Board</td>
<td>Wind River Tornado 2.2.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VxWorks 5.5.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BSP for CMD565</td>
</tr>
<tr>
<td>Axiom CMD565, VxWorks Module</td>
<td>Axiom CMD-565 Development Board</td>
<td>Wind River Tornado 2.2.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VxWorks 5.5.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BSP for CMD565</td>
</tr>
<tr>
<td>VxWorks Simulation</td>
<td>None</td>
<td>Wind River Tornado 2.2.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VxWorks 5.5.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Optional) ULIP Ethernet driver</td>
</tr>
<tr>
<td>Windows Console Application</td>
<td>None</td>
<td>One of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Visual Studio 6.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Visual Studio .NET</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Visual C ++ .NET</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TI Code Composer Studio 3.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Core SDK Platform</td>
</tr>
<tr>
<td>PHYTEC LPC229x, eCos</td>
<td>phyCORE-ARM7/LPC229x Rapid Development Kit</td>
<td>Cygwin 1.5.x</td>
</tr>
<tr>
<td></td>
<td>GPIO Expansion Board</td>
<td>eCos 2.0 ARM toolchain</td>
</tr>
<tr>
<td>Spectrum Digital DSK6713, DSP/BIOS</td>
<td>DSP Starter Kit (DSK) for the TMS320C6713</td>
<td>etyl Code Composer Studio 3.1</td>
</tr>
</tbody>
</table>

Contact the respective vendors for more information about their hardware and software products.

Refer to [ecos.sourceforge.org/getstart.html](http://ecos.sourceforge.org/getstart.html) for information about downloading and installing eCos.
Refer to gcc.gnu.org for information about downloading and installing GCC.

The VxWorks Development Kit for the LabVIEW Embedded Development Module includes the Tornado 2.2.1 integrated development environment and evaluation run-times for VxWorks 5.5.1 for the purpose of demonstrating the features, performance, and capabilities of these Wind River products in association with the Labview Embedded Development Module. Refer to windriver.com/alliances/eval-cd, click National Instruments Evaluation CD Program, and follow the instructions to receive the VxWorks Development Kit. Refer to windriver.com for more information about Wind River's Device Software Optimization products, including VxWorks real-time operating systems and Tornado, an integrated development environment.

### Setting Up the Example Targets

How you set up an example target to work with the Embedded Development Module depends on the target.

#### Axiom CMD565 Example Target

The Embedded Development Module includes a VxWorks Axiom CMD565 example target and an eCos Axiom CMD565 example target.

**VxWorks**

Use the RAM Module example target during normal development to build, download, and run applications from external RAM on the CMD565 board. Use the ROM Image example target to build an image you can download into the external flash memory.

**Module**

This example target configuration expects the VxWorks ROM resident image with a WDB serial connection in the external flash array. You download the object module you build into the external RAM of the board using the WTX protocol. The VxWorks ROM resident image must be in the external flash array, and you must configure the board to boot from the external flash array.
Complete the following steps to download the VxWorks ROM resident image to the external flash array.

1. Configure the Axiom CMD565 board as shown in Figure 2-1. This configuration is the default configuration that allows you to start the Axiom Monitor.

Figure 2-1. Axiom CMD565 VxWorks Default Configuration
2. Connect COM 1 to the available COM port on the host machine using a straight cable. A NULL modem cable does not work.

3. Launch AxIDE or another communication program, Tera Term for example, and open a connection to the serial port. The communication settings are 9600, 8, 1, N.


5. Press the <3> key to download to the external flash memory.

6. Press the <2> key to select external flash on CS2.

7. Press the <5> key to erase the external flash array.

8. Press the <6> key to program the external flash array.

9. Press the **Upload** button. Navigate to and run the following file. It takes a few minutes to program the external flash memory.

   ```
   labview\Targets\NI\Embedded\vxworks\cmd565\bin\
   vxWorks_rom Resident.S19
   ```
10. Power off the Axiom board and configure the board as shown in Figure 2-2.

**Figure 2-2.** Axiom CMD565 VxWorks Module Target Configuration

11. Power on the Axiom board. The VxWorks logo appears on the terminal. The board is configured and ready to use.
12. Run an example application to verify the setup. Refer to the Running a VxWorks Example Application section for more information about how to run an example application.

Running a VxWorks Example Application

You must have two serial ports to run the example application. COM 1 displays diagnostic information. COM 2 uses the target server to download an application you build. Verify you received the VxWorks logo on one serial port. Verify you received WDB READY on the other serial port. COM 2 communications at speed 57600.

Complete the following steps to run a VxWorks example application.
1. Launch LabVIEW and create a blank LabVIEW project.
2. Right-click the project in the Project Explorer window and select New»Targets and Devices from the shortcut menu to open the Add Targets and Devices dialog box.
3. Expand the Embedded folder and select the Axiom CMD565, VxWorks target.
4. Click the OK button to add the target to the project.
5. Right-click the Axiom CMD 565, VxWorks target and select New VI from the shortcut menu to create a blank VI and add it to the project.
6. Create a simple VI that prints something on the diagnostic output.
7. Click the Run button on the block diagram or front panel window. Follow the LabVIEW prompts to save the VI, create a new build specification, save the project, and build the embedded VI into an embedded project. The terminal that is connected to the diagnostic output displays the message the VI prints.

ROM

This target configuration allows you to download an application you build with the Embedded Development Module into the external flash array. The application automatically starts after you power on or reset the board. The application you build is self-contained in the external flash memory.

Complete the following steps to download the application image to the external flash array.
1. Configure the Axiom CMD565 board as shown in Figure 2-3. This is the default Axiom configuration that allows you to start the Axiom Monitor.
2. Connect COM 1 to the available COM port on the host machine using a straight cable. A NULL modem cable does not work.

3. Launch AxIDE or another communication program, Tera Term for example, and open a connection to the serial port. The communication settings are 9600, 8, 1, N.


5. Press the <3> key to download to the external flash memory.
6. Press the <2> key to select external flash on CS2.
7. Press the <5> key to erase the external flash array.
8. Press the <6> key to program the external flash array.
9. Press the **Upload** button on the AxIDE terminal.
10. Power off the Axiom board and configure the board as shown in Figure 2-4.

**Figure 2-4.** Axiom CMD565 VxWorks ROM Target Configuration
eCos

Use the RAM Module example target during normal development to build, download, and run applications from external RAM on the CMD565 board. Use the ROM Image example target to build an image you can download into the external flash memory.

RAM

This target configuration uses the RedBoot boot monitor to download and run a built application in the external RAM of the Axiom board. The RedBoot boot monitor is usually in the internal flash array of the MPC565 microcontroller and the board is configured to boot from the internal flash array.

Complete the following steps to download to the internal flash array.

1. Configure the board as shown in Figure 2-5. This is the default configuration that allows you to start the Axiom Monitor. Verify that CONFIG SWITCH 5 and 6 are set to ON to enable the internal flash array.
2. Launch AxIDE or another communication program, Tera Term for example, and open a connection to the serial port. The communication settings are 9600, 8, N, 1, N.


4. Press the <2> key to download to the internal flash memory.
5. Press the <E> key to erase the internal flash array.
6. Press the <P> key to program the internal flash array.
7. Press the Upload button.
8. Navigate to and run labview\Targets\NI\Embedded\ecos\cmd565\bin\redboot.s19.
9. Power off and configure the Axiom board as shown in Figure 2-6.
   Change the communication speed of the terminal program to 57,600 bps.
11. Run an example application to verify the setup. Refer to the Running
    an eCos Example Application section for more information on how to
    run an example application.
Figure 2-6. Axiom CMD565 eCos RAM Target Configuration

Running an eCos Example Application

You must have two serial ports to run the example application. COM 1 downloads an application you build. COM 2 displays diagnostic information. Verify you receive the RedBoot splash screen on both the COM ports. Both ports communicate at speed 57,600 bps.
Complete the following steps to run an eCos example application.

1. Launch LabVIEW and create a blank LabVIEW project.
2. Right-click the project in the Project Explorer window and select New»Targets and Devices from the shortcut menu to open the Add Targets and Devices dialog box.
3. Expand the Embedded folder and select the Axiom CMD565, eCos target.
4. Click the OK button to add the target to the project.
5. Right-click the Axiom CMD 565, eCos target and select New VI from the shortcut menu to create a blank VI and add it to the project.
6. Create a simple VI that prints something on the diagnostic output.
7. Click the Run button on the block diagram or front panel window. Follow the LabVIEW prompts to save the VI, create a new build specification, save the project, and build the embedded VI into an embedded project. The terminal that is connected to the diagnostic output displays the message the VI prints.

**ROM**

This target configuration allows you to download an application you build with the Embedded Development Module into the external flash array. The application automatically starts after you power on or reset the board. The application you build is self-contained in the external flash memory.

Complete the following steps to download the application image to the external flash array.

1. Configure the board as shown in Figure 2-7. This is the default configuration that allows you to start the Axiom Monitor. Verify that CONFIG SWITCH 5 and 6 are set to ON to enable the internal flash array.
2. Connect COM 1 to the available COM port on the host machine using a straight cable. A NULL modem cable does not work.

3. Launch AxIDE or another communication program, Tera Term for example, and open a connection to the serial port. The communication settings are 9600, 8, N, 1, N.
5. Press the <2> key to download to the internal flash memory.
6. Press the <E> key to erase the internal flash array.
7. Press the <P> key to program the internal flash array.
8. Press the **Upload** button.
9. Power off and configure the Axiom board. Change the communication speed of the terminal program to 57,600 bps.

**PHYTEC Example Target**

The PHYTEC example target uses the following hardware:

- phyCORE-ARM7/LPC229x Rapid Development Kit (Part Number KPCM-023-SK-2294-IAR)
- GPIO (General Purpose Input/Output) Expansion Board (Part Number PCM-989)

Refer to the PHYTEC Web site at [www.phytec.com](http://www.phytec.com) for more information about PHYTEC hardware.

**Connections**

You must establish the following connections on the expansion board between the I/O connector and Patch Field pins to use this target with the Embedded Development Module.

![Note](image-url) The **LPC2294 Function** column in the following table is the function to select when you configure the pin in software. Refer to the *LPC2119/2129/2194/2292/2294 User Manual*, available from the Philips Semiconductors Web site at [www.semiconductors.philips.com](http://www.semiconductors.philips.com), for information about how to configure the pins.

<table>
<thead>
<tr>
<th>Description</th>
<th>LPC2294 Function</th>
<th>Patch Field</th>
<th>I/O Connector</th>
</tr>
</thead>
<tbody>
<tr>
<td>LED 1</td>
<td>GPIO P1.16 (0)</td>
<td>8F</td>
<td>LED IN 1</td>
</tr>
<tr>
<td>LED 2</td>
<td>GPIO P1.17 (0)</td>
<td>9E</td>
<td>LED IN 2</td>
</tr>
<tr>
<td>LED 3</td>
<td>GPIO P1.18 (0)</td>
<td>10C</td>
<td>LED IN 3</td>
</tr>
<tr>
<td>LED 4</td>
<td>GPIO P1.19 (0)</td>
<td>10E</td>
<td>LED IN 4</td>
</tr>
<tr>
<td>LED 5</td>
<td>GPIO P1.20 (0)</td>
<td>10B</td>
<td>LED IN 5</td>
</tr>
<tr>
<td>LED 6</td>
<td>GPIO P1.21 (0)</td>
<td>11A</td>
<td>LED IN 6</td>
</tr>
</tbody>
</table>
Installing the RedBoot Bootloader

You must install the RedBoot bootloader on the phyCORE-ARM7/LPC229x processor before you can use it. Complete the following steps to write the RedBoot bootloader.

1. Install and start the LPC2000 Flash Utility, which is located on the PHYTEC CD.
2. Connect the board using the first serial interface at P1A, which is the lower socket of the double DB-9 connector at P1.
3. In the Filename text box, navigate to and select the redboot.hex file, which is located in the following directory:

   labview\Targets\NI\Embedded\ecos\phytec_lpc229x\libs\redboot
4. Select **LPC2294, XTAL Freq. [kHz]: 10000** from the **Device** list, which is the serial port on the host computer.

5. Select **9600** from the **Use Baud Rate** list.

6. Click the **Upload to Flash** button.

7. On the board, press the **Boot** button to start the In-System Programmer (ISP) while you press the **Reset** button to reset the board.

Refer to the documentation on the PHYTEC CD that comes with the Rapid Development Kit for more information about writing the RedBoot bootloader.

**Using the GNU Toolchain**

The PHYTEC example target uses the GNU arm-elf toolchain. Refer to the eCos Web site at [ecos.sourceforge.net](http://ecos.sourceforge.net) for information about how to install and build an ARM toolchain and how to install Cygwin for use with eCos.

You must install the toolchain in one of the following directories to use the PHYTEC example target with the Embedded Development Module:

- cygwin\gnutools
- cygwin\opt\gnutools
- cygwin

If you do not install the toolchain into one of these directories, you must add the location to the Windows path system variable.

**Spectrum Digital DSK6713 Example Target**

The Spectrum Digital DSK6713 example target uses the DSP Starter Kit (DSK) for the TMS320C6713. Refer to the Spectrum Digital Web site at [www.spectrumdigital.com](http://www.spectrumdigital.com) for more information about Spectrum Digital DSP Starter Kits.

**Using the TI Code Composer Studio Toolchain**

The Spectrum Digital DSK6713 uses the Code Composer Studio toolchain. Install Code Composer Studio from the CD in the DSP Starter Kit. Refer to the **DSP Starter Kit (DSK) for the TMS320C6713 (16Mb) Quick Start Installation Guide** for installation and set up instructions. The quick start guide is available with the DSP Starter Kit or on the Spectrum Digital Web site at [www.spectrumdigital.com](http://www.spectrumdigital.com).
If you install Code Composer Studio somewhere other than C:\CCStudio_v3.1, which is the default location, you must change the Compiler path and the Linker path in the Build Specification Properties dialog box for every LabVIEW project you create that uses the Code Composer Studio toolchain.
3

Using the Target Editor to Manage Embedded Targets

Managing an embedded target in LabVIEW involves creating and editing the TgtSupp.xml file, which LabVIEW uses to integrate your target implementation into the LabVIEW development environment. Use the Target Editor to generate and edit the TgtSupp.xml file for your target. The Target Editor is a tree-based configuration utility to help you manage targets you create with the LabVIEW Embedded Development Module. You can create new targets, modify existing targets, or delete existing targets.

When you select File→Save or File→Save As in the Target Editor, LabVIEW generates the TgtSupp.xml file for a new target or edits the TgtSupp.xml file for an existing target.

Tip Use the Target Editor to explore the various implementations in the Embedded Development Module example targets. Refer to Chapter 2, Example Targets, for information about the example targets.

Creating a New Target

Complete the following steps to create a new target.

1. Select Tools→Embedded Tools→Target Editor from the Getting Started window, the Project Explorer window, or a blank VI to open the Target Editor.
2. Select File→New.
3. Navigate to and create a folder in the labview\Targets directory.

Note National Instruments recommends using your company name, or some variation of your company name, when you create a new folder. For example, National Instruments targets are located in the labview\Targets\NI directory.

4. Open the folder you created in step 3 and click the Current Folder button. LabVIEW adds an Embedded folder to the Target Editor.
5. Right-click the **Embedded** folder and select **New»Folder** from the shortcut menu. A **New Folder** folder appears under the **Embedded** folder.

**Tip** Right-click **New Folder** and select **Rename** from the shortcut menu to rename the folder. You can rename the folder to something more descriptive, such as the OS name.

6. Right-click **New Folder** and select **New»Target** from the shortcut menu. A **New Target** item appears under the **New Folder** folder.

7. Right-click **New Target** and select **Rename** from the shortcut menu. Rename **New Target** to your target name.

8. Right-click your renamed target and select **Properties** from the shortcut menu to open the **Target Properties** dialog box. Refer to the **Configuring Target Properties** section for information about how to use the **Target Properties** dialog box to configure the target properties.

9. Right-click your renamed target and select **New»Build Specification Type** from the shortcut menu to open the **Build Specification Type Properties** dialog box. Refer to the **Configuring Build Specification Type Properties** section for information about how to use the **Build Specification Type Properties** dialog box to configure the build specification type properties.

### Modifying an Existing Target

Complete the following steps to modify an existing target.

1. Select **Tools»Embedded Tools»Target Editor** from the **Getting Started** window, the **Project Explorer** window, or a blank VI to open the Target Editor.

2. Select **File»Open**.

3. Navigate to and select the **TgtSupp.xml** file for the target you want to modify.

4. Right-click the target and select **Properties** from the shortcut menu to open the **Target Properties** dialog box. Refer to the **Configuring Target Properties** section for information about how to use the **Target Properties** dialog box to configure the target properties.

5. Right-click the build specification type and select **Properties** from the shortcut menu to open the **Build Specification Type Properties** dialog box. Refer to the **Configuring Build Specification Type Properties** section for information about how to use the **Build**
Specifying Type Properties dialog box to configure the build specification type properties.

Configuring Target Properties

You use the Target Properties dialog box in the Target Editor to configure the target properties for new and existing embedded targets.

Configuring General Items

Complete the following steps to use the General tab in the Target Properties dialog box to configure general items for your target.

1. (Optional) Place a checkmark in the Preallocate UI Enable (Static Memory Allocation) checkbox if you want the preallocate user interface for your target. If you place a checkmark in this checkbox, all VIs that open under your target contain a Set Dimension Size shortcut menu option for block diagram array constants.

2. Navigate to and select or enter the relative path to the target icon file, which is the icon that shows in the Project Explorer window for your target, in the Project Explorer window icon text box. Icons must be in .png format. You can specify the icon by name. Icons are located in the following directory:

   labview\resource\Framework\Providers\Icons

3. Navigate to or enter the relative path to the directory that contains the TargetInfo.ini file—which is the file that defines the target syntax, palette, and controls specific to your target—in the Path to TargetInfo.ini text box. You can reuse an existing configuration that works for a majority of embedded targets or you can copy and modify an existing TargetInfo.ini file.

   Most example targets use the default configuration for targets that use dynamic memory allocation, which is located in the following directory:

   labview\Targets\NI\Embedded\common\TargetInfo

   The default configuration for example targets that use static memory allocation is located in the following directory:

   labview\Targets\NI\Embedded\common\static\TargetInfo
4. (Optional) Navigate to or enter the relative path to the directory that contains the Elemental I/O configuration file. Leave the Path to Elemental I/O configuration file text box blank if your target does not support Elemental I/O.

5. (Optional) Navigate to or enter the relative path to the subtarget directory in the Path to subtarget text box. You can use a single TgtSupp.xml file to define multiple targets. Use this text box when the TgtSupp.xml file defines a hierarchy of targets where the subtarget VIs call the base target VIs.

**Defining Shortcut Menu Items**

Complete the following steps to use the Shortcut Menu Items tab in the Target Properties dialog box to define the target-specific shortcut menu items for your target. Users see the shortcut menu items when they right-click your target in the Project Explorer window.

1. Click the blue plus button, shown at left, to enable the Item Properties section. ?? appears in the Menu Items list.

2. Select if you are adding a menu item or a separator from the Item Type pull-down menu. If you add a separator, repeat step 1 to add another menu item or separator.

3. If you added a menu item in step 2, enter the menu item name in the Item Name text box. This is the shortcut menu item users see.

4. Navigate to or enter the relative path to the plug-in VI that implements the functionality for the menu item. If you do not have an existing plug-in VI, click the Create New Plug-In VI button, shown at left, to open the Create New Plug-In VI dialog box. Refer to the Creating New Plug-In or Subpanel VIs section for more information about creating plug-in VIs.

5. Repeat steps 1 through 4 for each menu item you want to define. Click the red x button, shown at left, to remove the selected menu item.

**Defining Categories**

Complete the following steps to use the Category tab in the Target Properties dialog box to define the target-specific pages in the Properties dialog box for your target. Users see the pages you add in the Category list when they right-click your target in the Project Explorer window and select Properties from the shortcut menu.

- **Note** LabVIEW adds the Conditional Disable Symbols Properties page, which you use with the Conditional Disable structure, by default.
1. Click the blue plus button, shown at left, to enable the **Page Properties** section. ??? appears in the **Category** list.

2. Enter the name that you want to appear in the **Category** list in the **Target Properties** dialog box for your target.

3. Navigate to or enter the relative path to the subpanel VI. If you do not have an existing subpanel VI, click the **Create New Subpanel VI** button, shown at left, to open the **Create New Subpanel VI** dialog box. Refer to the *Creating New Plug-In or Subpanel VIs* section for more information about creating subpanel VIs.

4. Repeat steps 1 through 3 for each new category you want to define. Click the red x button, shown at left, to remove the selected category.

**Defining Timing Sources**

Complete the following steps to use the **Timing Sources** tab in the **Target Properties** dialog box to define the timing sources, in addition to the 1 kHz clock, for your target. Users see the timing sources you define as options in the **Configure Timed Loop** dialog box.

1. Click the blue plus button, shown at left, to enable the **Timing Source Properties** section. ??? appears in the **Timing Source** list.

2. Enter the timing source in the **Source Type** text box.

3. Enter the name that appears in the **Configure Timed Loop** dialog box in the **Source Name** text box.

4. Repeat steps 1 through 3 for each new timing source you want to define. Click the red x button, shown at left, to remove the selected timing source.

**Defining Events**

Use the **Events** tab in the **Target Properties** dialog box to define the callback VI that is called every time a user removes your target from the **Project Explorer** window or closes a project. Navigate to or enter the relative path to the subpanel VI in the **OnRelease Command Path** text box. If you do not have an existing subpanel VI, click the **Create New Subpanel VI** button, shown at left, to open the **Create New Subpanel VI** dialog box. Refer to the *Creating New Plug-In or Subpanel VIs* section for more information about creating subpanel VIs.
Defining Signed Files

Use the **Signed Files** tab in the **Target Properties** dialog box to define which plug-in VIs and files end users cannot modify.

Click the blue plus button, shown at left, to navigate to and select the files you do not want end users modifying. For example, you do not want end users modifying your target-specific plug-in VIs.

To remove a file from the list, select the file and click the red x button, shown at left, to remove the selected file from the list. The Target Editor calculates signatures for all files you list and stores the signatures in TgtSupp.xml.

Configuring Licensing

Complete the following steps to use the **License** tab in the **Target Properties** dialog box to enter the name and version National Instruments sends you if you are reselling your target.

1. Enter your OEM-specific feature name. National Instruments assigns this name. The name matches the name in the OEM-specific license file.
2. Enter the OEM-specific feature version. National Instruments assigns this version and the version matches the version in the OEM-specific license file.

**Note** Refer to the *LabVIEW Embedded Development Module Target Distribution Guide*, available by selecting **Start»All Programs»National Instruments»LabVIEW»LabVIEW Manuals** and opening EMB_Distribution_Guide.pdf, for information about licensing requirements if you want to resell your target and how to obtain the feature name and version.
Configuring Build Specification Type Properties

You use the **Build Specification Type Properties** dialog box in the Target Editor to configure the build specification properties for new and existing embedded targets.

### Configuring General Build Specification Items

Complete the following steps to use the **General** tab in the **Build Specification Type Properties** dialog box to configure the general build specification items for your target.

1. (Optional) Place a checkmark in the **Default Build Specification Type** checkbox to indicate that this is the default build specification to use when a user clicks the **Run** button to build the embedded VI into an embedded application.

2. (Optional) Place a checkmark in the **Enable Interrupt VIs Property Page** checkbox to add the **Interrupts VIs** page to the **Build Specification Properties** dialog box.

3. Navigate to and select or enter the relative path to the build specification type icon file, which is the icon that LabVIEW displays in the **Project Explorer** window for your build specification type, in the **Project Explorer Window Icon** text box. Icons must be in `.png` format. You can specify the icon by name. Add your icon to the following directory:

   ```
   labview\resource\Framework\Providers\Icons
   ```

4. Enter an ID in the **Build Specification Type ID** text box to uniquely identify the build specification type. Each build specification type for a target must have a unique name.

5. Enter a name for the build specification type in the **Default Build Specification Name** text box, which is the name that appears when a user selects **Build Specification** in the **Project Explorer** window and selects **New** from the shortcut menu.

6. Enter the file extensions for the type of additional files a user can add to the project in the **Additional Files** text box. Users can add only the file types you enter. Enter the file extensions as a comma-separated list.

*Note* Do not include the dot (.) before the file extensions.
Defining Build Specification Shortcut Menu Items

Complete the following steps to use the **Shortcut Menu Items** tab in the **Build Specification Type Properties** dialog box to define the target-specific shortcut menu items for the build specification type. Users see the shortcut menu items when they right-click a build specification under your target in the **Project Explorer** window.

1. Click the blue plus button, shown at left, to enable the **Item Properties** section. ??? appears in the **Menu Items** list.

2. Select if you are adding a menu item or a separator from the **Item Type** pull-down menu. If you add a separator, repeat step 1 to add another menu item or separator.

3. If you added a menu item in step 2, enter the menu item name in the **Item Name** text box. This is the shortcut menu item users see.

4. Navigate to or enter the relative path to the plug-in VI that implements the functionality for the menu item. If you do not have an existing plug-in VI, click the **Create New Plug-In VI** button, shown at left, to open the **Create New Plug-In VI** dialog box. Refer to the **Creating New Plug-In or Subpanel VIs** section for more information about creating plug-in VIs.

5. Repeat steps 1 through 4 for each menu item you want to define. Click the red x button, shown at left, to remove the selected menu item.

Defining Build Specification Categories

Complete the following steps to use the **Category** tab in the **Build Specification Type Properties** dialog box to define the target-specific pages in the **Build Specification Properties** dialog box for your target. Users see the pages you add in the **Category** list when they right-click a build specification under your target in the **Project Explorer** window and select **Properties** from the shortcut menu.

1. Click the blue plus button, shown at left, to enable the **Page Properties** section. ??? appears in the **Category** list.

2. Enter the name that you want to appear in the **Category** list in the **Target Properties** dialog box for your target.

3. Navigate to or enter the relative path to the subpanel VI. If you do not have an existing subpanel VI, click the **Create New Subpanel VI** button, shown at left, to open the **Create New Subpanel VI** dialog box. Refer to the **Creating New Plug-In or Subpanel VIs** section for more information about creating subpanel VIs.
4. Repeat steps 1 through 3 for each new category you want to define. Click the red x button, shown at left, to remove the selected category.

Defining Build Specification Events

Complete the following steps to use the Events tab in the Build Specification Type Properties dialog box to define the events related to building an embedded VI into an embedded application.

1. In the Path to OnBuild Plug-In VI text box, navigate to or enter the relative path to the callback VI that is called when a user presses the <Ctrl> key while clicking the Run button. If you do not have an existing plug-in VI, click the Create New Plug-In VI button, shown at left, to open the Create New Plug-In VI dialog box. Refer to the Creating New Plug-In or Subpanel VIs section for more information about creating plug-in VIs.

2. In the Path to OnRun Plug-In VI text box, navigate to or enter the relative path to the callback VI that is called when a user clicks the Run button. If you do not have an existing plug-in VI, click the Create New Plug-In VI button to open the Create New Plug-In VI dialog box.

3. In the Path to OnDebug Plug-In VI text box, navigate to or enter the relative path to the callback VI that is called when a user clicks the Pause button and then the Run button. If you do not have an existing plug-in VI, click the Create New Plug-In VI button to open the Create New Plug-In VI dialog box.

4. In the Path to OnAbort Plug-In VI text box, navigate to or enter the relative path to the callback VI that is called when a user clicks the Abort Execution button. If you do not have an existing plug-in VI, click the Create New Plug-In VI button to open the Create New Plug-In VI dialog box.

Creating New Plug-In or Subpanel VIs

You use plug-in and subpanel VIs to implement your target in LabVIEW. Subpanel VIs implement the LabVIEW user interface for your target.

Creating New Plug-In VIs

Complete the following steps to create a new plug-in VI in the Target Editor.

1. Click one of the Create New Plug-in VI buttons, shown at left, in the Target Properties or Build Specification Type Properties dialog boxes to open the Create New Plug-In VI dialog box.
2. Select which type of plug-in VI you want to create.
   a. Select **VI with test dialog** to create a new plug-in VI that includes the One Button Dialog function for testing purposes.
   b. Select **Blank VI** to create a new, blank plug-in VI.
3. Navigate to or enter the relative path to the plug-in VI you are creating in the **Path to Plug-in VI** text box.
4. Click the **OK** button.

### Creating New Subpanel VIs

Complete the following steps to create a new subpanel VI in the Target Editor.

1. Click one of the **Create New Subpanel VI** buttons, shown at left, in the **Target Properties** or **Build Specification Type Properties** dialog boxes to open the **Create New Subpanel VI** dialog box.
2. Select which type of subpanel VI you want to create.
   a. Select **Subpanel VI with test controls** to create a new subpanel VI that includes some front panel controls for testing purposes.
   b. Select **Blank VI** to create a new, blank subpanel VI.
3. Enter the category name, which is the name that appears in the **Category** list in the **Target Properties** or **Build Specification Properties** dialog box, in the **Name** text box.
4. Navigate to or enter the relative path to the subpanel VI you are creating in the **Path to Subpanel VI** text box.
5. Click the **OK** button.
Defining Code Generation Options

A VI Server call translates the VI hierarchy into C code. The LabVIEW C Code Generator generates each VI in the VI hierarchy into a separate C file using a C function name, which is more restrictive than a VI name, when a user builds an embedded VI into an embedded application. Any non-alphanumeric characters become underscores. If the VI name begins with a non-alphanumeric character, the LabVIEW C Code Generator prepends `A_` to the beginning of the C function name.

This VI Server call also generates the following header files.

<table>
<thead>
<tr>
<th>Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVForms.h</td>
<td>Contains declarations for the entry point functions for each VI.</td>
</tr>
<tr>
<td>LVGlobs.h</td>
<td>Contains declarations for global initialization functions and data type information used throughout the embedded application.</td>
</tr>
<tr>
<td>LVISRList.h</td>
<td>Contains a list of Interrupt Service Routines (ISRs).</td>
</tr>
<tr>
<td>LVFuncsUsed.h</td>
<td>Contains usage information to support source-level dead stripping.</td>
</tr>
<tr>
<td>LVDebugTable.h</td>
<td>Contains lookup information for serial, TCP/IP, and CAN debugging.</td>
</tr>
</tbody>
</table>

The input to this VI Server call is a variant that contains attributes (name, value pairs) that determine how the LabVIEW C Code Generator generates the C code from the block diagram. Any attributes you do not set use the default behavior.

Many attributes determine how to generate the C code. Most of the attributes are optimizations that make the generated C code smaller and run faster. Refer to the Code Generation Attributes section for information about when you can use different attributes. The generated C code might differ in behavior from VIs you create under the My Computer target, or Windows.
Chapter 4  Defining Code Generation Options

Note  This VI Server call is licensed. The user must have a valid, activated Embedded Development Module license to generate C code for the target you are creating.

Code Generation Attributes

The following attributes, which you set in the GenerateCCodeVariant Invoke Node in the CGen plug-in VI, determine how the LabVIEW C Code Generator generates the C code.

BigEndian

Type: Boolean  
Default: FALSE

Specifies if a platform is big endian or little endian. The LabVIEW C Code Generator must know if a platform is big endian or little endian in some cases to generate C code in byte form for compound data.

DestinationFolder

Type: Path  
Default: <empty path>

Indicates where you want the LabVIEW C Code Generator to save the generated C files. Unless the IncrementalBuild attribute is TRUE, any C files in the destination folder with the same name as VIs in the embedded VI are overwritten.

GenerateCFunctionCalls

Type: Boolean  
Default: FALSE

Determines the calling interface to subVIs if you set GenerateSerialOnly to TRUE. The LabVIEW C Code Generator uses default data when generating the calling interface to a subVI if an input to or output from the subVI is unwired, which increases the overall amount of generated code and data for a VI relative to what you might use in a normal C application. If all the inputs and outputs are wired, the LabVIEW C Code Generator does not need the default data and can generate a more efficient interface.

Set this attribute to TRUE to generate the interface to all VIs as C-style function calls without any default data initialization, which can reduce the code size by as much as 50% for a small VI. An error occurs if any input or
output to any VI is unwired when the LabVIEW C Code Generator generates the C code.

If you set `GenerateCFunctionCalls` to TRUE, you also must set `GenerateSerialOnly` to TRUE.

### GenerateDebugInfo

**Type:** Boolean  
**Default:** FALSE

Enables instrumented debugging and adds the extra code needed to debug over serial, Ethernet, or CAN protocols if you set this attribute to TRUE. The extra code contains function calls that update the program state and communicate the state to the host computer for display. Setting this attribute to TRUE usually results in a 25%–40% increase in code size. You cannot use the `GenerateCFunctionCalls` attribute with `GenerateDebugInfo`.

### GenerateGuardCode

**Type:** Boolean  
**Default:** TRUE

Determines whether to generate guard code. Guard code prevents a user from making common mistakes that can cause an application to crash. For example, guard code can prevent dividing by zero and indexing out of range in an array. Guard code makes an application slightly larger and slower so you are trading performance for reliability. Set this attribute to FALSE to not generate guard code, which makes the code smaller and faster but less safe and more likely to crash because of user programming mistakes.

The following LabVIEW functions use guard code:

- All floating-point arithmetic operations
- Compound Arithmetic
- Index Array
- Quotient & Remainder
- Replace Array Subset
GenerateInlinePrintf

**Type:** Boolean  
**Default:** FALSE

Indicates you want to use the C run-time library function `printf` if it is available on your target. For example, the Format Into String function is completely implemented in the LabVIEW C Code Generator run-time library to support LabVIEW data types. However, if you use only integer, floating-point, or Boolean data, you can use the C run-time library function `printf` instead. Set this attribute to TRUE to use `printf` when possible. The LabVIEW C Code Generator run-time library is used if you cannot use `printf`. The `printf` function is usually smaller and faster than the LabVIEW C Code Generator run-time version for simple data types.

GenerateIntegerOnly

**Type:** Boolean  
**Default:** FALSE

Generates C code that does not have any floating-point declarations or operations if the block diagram does not contain any floating-point data types. You can link the generated code with a run-time library compiled with the `_Integer_Only` flag set to produce applications that run without hardware or software floating-point support. **GenerateIntegerOnly** does not support fixed-point operations in the Embedded Development Module.

GenerateLibraryInterface

**Type:** Boolean  
**Default:** FALSE

Generates additional code so that external, non-VI code can call the VIs as if the VIs are library functions. This attribute is supported only for strings, 1D arrays, and flat clusters. For example, you cannot use a cluster that contains a string.

The LabVIEW C Code Generator generates a C-style function interface for each VI, which configures the inputs and calls the normal LabVIEW VI interface. The C function name is the same as the VI name unless the VI name contains a disallowed character, such as a space, in the filename. Underscores replace disallowed characters. The VI behaves the same as a VI running under the My Computer target, or Windows, by using default data if any input or output is null. Finally, a header file is created that
contains all of the function prototypes for the VIs that can be included where they are called. The header file has the same name as the top-level VI appended with Lib.h. Inputs are passed by value except structs, which are passed by address.

A size parameter for both inputs and outputs follows the array and string parameters. For an array, the size is the total number of elements in the array. For a string, the size is the length of the string. If you pass in a multi-dimensional array, it is copied into an internal array with the LabVIEW data type determining the number of dimensions. All of the dimension sizes are set to 1 except for the innermost dimension, which is equal to the size passed in from internal data. For an output array, the data is block copied to the output up to the specified size limit. The easiest way to pass arrays is as 1D arrays that get reshaped as necessary on the block diagram.

GenerateSerialOnly

Type: Boolean
Default: FALSE

Determines whether to generate cooperative multitasking code. The default generated C code has the same execution behavior as LabVIEW VIs running under the My Computer target, or Windows. For example, parallel While Loops run in parallel. However, the LabVIEW C Code Generator generates code that uses cooperative multitasking in a single thread. Additional threads are used only by Timed Loops. The generated C code is difficult to read and does not resemble human-written C code.

Set this attribute to TRUE if you do not want to generate the cooperative multitasking code. The code is easier to read and usually runs much faster without the cooperative multitasking code. However, you lose the parallel execution behavior. Parallel While Loops execute in sequence with one While Loop executing only after the other While Loop completely finishes. This execution behavior most closely resembles subroutine priority VIs in LabVIEW Windows VIs. If a user sets the priority of an individual VI to subroutine on the Execution page on the VI Properties dialog box, the C code is generated for serial only execution regardless of the value of GenerateSerialOnly. Setting GenerateSerialOnly to TRUE causes the LabVIEW C Code Generator to generate the C code for all VIs in the VI hierarchy as if all VIs are subroutine priority VIs. The main difference between the subroutine priority setting in the VI Properties dialog box and the GenerateSerialOnly attribute is that subroutine Windows VIs do not allow asynchronous functions. Ordinary VIs that generate C code with GenerateSerialOnly set to TRUE can contain asynchronous functions,
which the LabVIEW C Code Generator turns into synchronous functions and does not return to the caller until complete. TCP Read is an example of an asynchronous function.

**Note** If you set `UseStackVariables` or `GenerateCFunctionCalls` to TRUE, you also must set `GenerateSerialOnly` to TRUE.

### IncrementalBuild

**Type:** Boolean  
**Default:** FALSE

Indicates whether to rebuild everything every time a change is made or to rebuild just the files that changed. Set this attribute to TRUE to overwrite only the C files that are older than the VIs from which they are generated during code generation. For example, if `MyProg.vi` has a modification date that is more recent than `MyProg.c`, the C file is overwritten.

Incremental builds allow users to write and combine large applications with a *makefile* build process that does not rebuild everything every time a minor change is made. If the VIs are in a `.llb`, all VIs in the `.llb` are regenerated if any of the VIs in the `.llb` change.

### interruptServiceRoutineVIs

**Type:** Path array  
**Default:** <empty array>

Lists the VIs that are interrupt service routine (ISR) VIs, which are very specialized and restrictive VIs for use as interrupts. The Embedded Development Module does not support the ability to set a flag in a VI indicating that it is an ISR VI. Refer to Chapter 10, *Implementing Interrupt Service Routine (ISR) Support*, for more information about implementing ISR support.
**MemoryModel**

**Type:** Integer (0 = dynamic, 1 = static)

**Default:** 0

Defines how to manage memory in an embedded application. The following table describes the two memory models.

<table>
<thead>
<tr>
<th>Memory Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dynamic</td>
<td>Memory is allocated and then freed as soon as possible, which keeps the memory use at any one time to a minimum. Memory management overhead might result in some jitter and reduced performance, but applications with a small memory model have the smallest footprint possible.</td>
</tr>
<tr>
<td>static</td>
<td>No memory allocation occurs while the application is running. All variables are declared as static in the generated C code and allocated prior to the main entry point. Static memory can provide good performance and low jitter, but might require more memory than other memory models.</td>
</tr>
</tbody>
</table>

**OCDIComments**

**Type:** Boolean

**Default:** FALSE

Determines whether to generate extra comments necessary for OCDI debugging. The LabVIEW C Code Generator places extra comments in the generated C code to help figure out where certain items, such as wires and nodes, are in the generated C code during OCDI debugging. These comments make the generated C code harder to read. Use this attribute to turn off these comments when you do not need them.

**Silent**

**Type:** Boolean

**Default:** FALSE

Suppresses any dialog boxes that might appear while the LabVIEW C Code Generator generates the C code, which is useful for unattended testing.
TargetAlignment

Type: 32-bit integer
Default: 4

Defines the alignment for such things as structures for the target. You must pass in the same number as you set in the ALIGNMENT macro in LVDefs_plat.h.

UseStackVariables

Type: Boolean
Default: FALSE

Determines whether to use stack variables to represent wires in the generated C code. Set UseStackVariables to TRUE to use stack variables.

C stack variables cannot represent wires that execute parallel While Loops or For Loops because more than one C function can access wire values. However, if GenerateSerialOnly is TRUE, you can use stack variables because the LabVIEW C Code Generator generates the entire VI block diagram as one C function. The C compiler you use might be able to make more optimizations in the C code, which results in a faster executable if the stack is large enough for the variables.

Note If you set UseStackVariables to TRUE, you also must set GenerateSerialOnly to TRUE.

Generating the Fastest Code

Use the following code generation options to generate the fastest code:

- GenerateGuardCode=FALSE
- GenerateSerialOnly=TRUE
- UseStackVariables=TRUE
- GenerateCFunctionCalls=TRUE
Signal Naming Convention

The LabVIEW C Code Generator generates C-style source code comments and variable names from the labels on the block diagram. Underscores ( _ ) replace special characters and spaces.

Use the VarNames VI, located in the labview\CodeGen directory to set the maximum size of variable names, data type prefix strings, and tunnel name abbreviation strings.

Tip Keep prefix names and tunnel abbreviation strings short to make comments more readable and to avoid truncated variable names or comments.

The following table shows the LabVIEW signal source and corresponding generated C variable name and comment.

<table>
<thead>
<tr>
<th>LabVIEW Signal Source</th>
<th>Attribute Used</th>
<th>Block Diagram Examples</th>
<th>Generated C Variable Name</th>
<th>Generated C Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front panel control</td>
<td>Label</td>
<td>Motor On</td>
<td>Motor_On</td>
<td>/* Motor On */</td>
</tr>
<tr>
<td>Constant</td>
<td>Label</td>
<td>Trigger Level</td>
<td>Trigger_Level</td>
<td>/* Trigger Level */</td>
</tr>
<tr>
<td>Function</td>
<td>Label, Terminal Name</td>
<td>Over Voltage</td>
<td>Over_Voltage_x_y_</td>
<td>*/ Over Voltage: x &gt; y? */</td>
</tr>
<tr>
<td>Loop tunnel</td>
<td>Source name, “LT”</td>
<td>Trigger_Level_LT</td>
<td>Trigger_Level_LT</td>
<td>/* Trigger Level: Loop Tunnel */</td>
</tr>
<tr>
<td>Case selector</td>
<td>Source name, “CS”</td>
<td>Relay_CS</td>
<td>Relay_CS</td>
<td>/* Relay: Case Selector */</td>
</tr>
<tr>
<td>LabVIEW Signal Source</td>
<td>Attribute Used</td>
<td>Block Diagram Examples</td>
<td>Generated C Variable Name</td>
<td>Generated C Comment</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------------</td>
<td>------------------------</td>
<td>---------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Case tunnel</td>
<td>Structure Label, “CT”</td>
<td></td>
<td>My_Structure_CT</td>
<td>/* My Structure: CT */</td>
</tr>
<tr>
<td>Shift register</td>
<td>Source name, “SR”</td>
<td></td>
<td>Init_SR, Init_SR_1</td>
<td>/* Init: SR <em>/ /</em> Init: SR 1 */</td>
</tr>
<tr>
<td>Shift register (uninitialized)</td>
<td>Source name, “SR”</td>
<td></td>
<td>Foo_Loop_SR, Foo_Loop_SR_1, Foo_Loop_SR_2</td>
<td>/* Foo Loop: SR <em>/ /</em> Foo Loop: SR 1 <em>/ /</em> Foo Loop: SR 2 */</td>
</tr>
<tr>
<td>Sequence Local</td>
<td>Source name, “SL”</td>
<td></td>
<td>Init_SL</td>
<td>/* Init: Sequence local */</td>
</tr>
<tr>
<td>Sequence tunnel</td>
<td>Source name, “ST”</td>
<td></td>
<td>Level_Y_ST</td>
<td>/* Level Y: ST */</td>
</tr>
</tbody>
</table>
## Note
Generated variables must be unique. If multiple variables have identical names, the LabVIEW C Code Generator appends a sequential number to the end of the variable name to make the name unique.
Compiler directives enable and disable features. The directives are passed to the compiler from the command line by LEP_x_ScriptCompiler.vi. The **Build Specification Properties** dialog box exposes the implementation of the compiler directives.

The macros in this chapter define how to compile generated and external C code. The LabVIEW Embedded Development Module example targets compile each generated C file by scripting the C compiler from LEP_x_ScriptCompiler.vi, which is a plug-in VI located in the following directory:

```
labview\Targets\Embedded\os\os_LEP_TargetPlugin
```

where `os` is the target operating system. Some example targets include the target name in the subdirectory.

LEP_x_ScriptCompiler.vi can directly execute the C compiler by calling the System Exec VI. You also can use a batch file to directly execute your C compiler. Use a batch file if you need to set compiler-specific system variables before the compiler can execute. For example, the VxWorks examples use a batch file because you must set the Tornado system variables before the C compiler executes.

When a user builds an embedded VI into an embedded application, LEP_x_Build.vi calls LEP_x_ScriptCompiler.vi for each generated source file. The VxWorks and eCos example targets do not have pre-compiled run-time libraries, so you must compile the run-time source files along with the generated and external C files. LEP_x_Build.vi executes the default implementation of LEP_x_ScriptCompiler.vi unless you override the default implementation and create a custom LEP_x_ScriptCompiler.vi. You can override the default implementation by wiring a LEP_x_ScriptCompiler.vi VI reference to the **Compiler Script** input of LEP_x_Build.vi. Your custom implementation of LEP_x_ScriptCompiler.vi must have the same connector pane as the default implementation. Refer to the Axiom CMD565, VxWorks RAM Image target for an example of how to override the default implementation.
For most targets, the base target provides a complete, tools-aware embedded OS build framework with the default implementation of LEP_x_ScriptCompiler.vi. You can use the base target default implementation of the build framework and LEP_x_ScriptCompiler.vi and override just the necessary features.

The VxWorks CMD565 example target uses a subtarget-specific implementation, which is located in the following file:

```
labview\Targets\NI\Embedded\vxworks\cmd565\vxworks_cmd565_LEP_TargetPlugin\LEP_vxworks_cmd565_Build.vi
```

This custom implementation overrides the default implementation of LEP_x_ScriptCompiler.vi, LEP_x_ScriptLinker.vi, and LEP_x_Config.vi. Notice the Case structure with target-specific VIs in this custom implementation. The VIs in the Case structure never execute, but are always loaded into memory. Because the VIs are loaded into memory, the Open VI Reference function can refer to VIs by name instead of absolute path.

**Note** Defining unsupported features can result in compiler errors or run-time errors.

### General Macros

The following table describes the general macros.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHeadless</td>
<td>Always define this macro. PDA targets are currently the only code-generated targets that support user interfaces.</td>
</tr>
<tr>
<td>_Integer_Only</td>
<td>Define for integer-only targets.</td>
</tr>
<tr>
<td>CStatic</td>
<td>Define for the static memory model.</td>
</tr>
</tbody>
</table>

### Operating System Macros

The following table describes the operating system macros.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win32</td>
<td>Define for all Windows targets.</td>
</tr>
<tr>
<td>_vxworks</td>
<td>Define for all VxWorks targets except VxWorks Simulation targets.</td>
</tr>
</tbody>
</table>
### Feature Macros

The following table describes the feature macros.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_vxworksim</td>
<td>Define for all VxWorks Simulation targets.</td>
</tr>
<tr>
<td>_ECOS</td>
<td>Define for all eCos targets.</td>
</tr>
<tr>
<td>linux</td>
<td>Define for all Linux targets.</td>
</tr>
<tr>
<td>CANSupport</td>
<td>Set to 1 to enable VxWorks CAN support. To use Embedded CAN with your target, you must obtain a driver and libraries for the board from Wind River and compile it into your VxWorks image.</td>
</tr>
<tr>
<td>SocketSupport</td>
<td>Set to 1 to enable BSD socket support.</td>
</tr>
<tr>
<td>_Include_OSScheduler</td>
<td>Set to 1 to enable multithreading support. The LabVIEW C Code Generator usually sets this macro.</td>
</tr>
<tr>
<td>HeapListSupport</td>
<td>Set to 1 to enable heap support. You usually set this macro in LVDefs_plat.h.</td>
</tr>
<tr>
<td>BluetoothSupport</td>
<td>Set to 1 to enable Bluetooth support. You usually set this macro in LVDefs_plat.h.</td>
</tr>
<tr>
<td>FileSupport</td>
<td>Set to 1 to enable file I/O support. You usually set this macro in LVDefs_plat.h.</td>
</tr>
<tr>
<td>IrDA_Support</td>
<td>Set to 1 to enable IrDA support. You usually set this macro in LVDefs_plat.h.</td>
</tr>
<tr>
<td>SerialSupport</td>
<td>Set to 1 to enable serial support. You usually set this macro in LVDefs_plat.h.</td>
</tr>
<tr>
<td>DatalogSupport</td>
<td>Set to 1 to enable datalogging support. You usually set this macro in LVDefs_plat.h.</td>
</tr>
<tr>
<td>TCPUDPSupport</td>
<td>Set to 1 to enable TCP/UDP support. You usually set this macro in LVDefs_plat.h.</td>
</tr>
</tbody>
</table>
Memory Mapping Macros

The following table describes the memory mapping macros.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DATA_SECTION</td>
<td>Defines the input memory section for array structures.</td>
</tr>
<tr>
<td>_TEXT_SECTION</td>
<td>Defines the input memory section for generated code.</td>
</tr>
</tbody>
</table>

Debugging Macros

The following table describes the debugging macros.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UsesSerialDebugger</td>
<td>Define to enable debugging over a serial line.</td>
</tr>
<tr>
<td>DBGBAUD</td>
<td>Specify the baud rate of the serial line for serial debugging.</td>
</tr>
<tr>
<td>UsesTCPDebugger</td>
<td>Define to enable debugging over a TCP/IP connection.</td>
</tr>
<tr>
<td>UsesCANDebugger</td>
<td>Define to enable debugging over a CAN bus.</td>
</tr>
<tr>
<td>OCDI_RDM</td>
<td>Define to enable the Release Debug Mode for on chip debugging.</td>
</tr>
</tbody>
</table>

VxWorks-Only Macros

The following table describes the VxWorks-only macros.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Define to indicate the CPU family.</td>
</tr>
<tr>
<td></td>
<td>• SIMNT for VxWorks Simulation running on Windows.</td>
</tr>
<tr>
<td></td>
<td>• PPC5XX for Power PC 5XX class CPU.</td>
</tr>
<tr>
<td></td>
<td>• PPC405 for Power PC 405 CPU.</td>
</tr>
<tr>
<td></td>
<td>• XSCALE for Intel XScale class CPU.</td>
</tr>
</tbody>
</table>
Configuring the Linking of Object Files and Libraries

The target-specific LEP_x_SCRIPTLINKER.vi links all intermediate object files (.o), external libraries, and operating system libraries to generate the application image. LEP_x_SCRIPTLINKER.vi is located in the following directory:

\labview\Targets\Embedded\os\os_LEP_TargetPlugin

where os is the target operating system.

You must implement LEP_x_SCRIPTLINKER.vi if you require behavior that differs from the default implementation.

LEP_x_BUILD.vi executes LEP_x_SCRIPTLINKER.vi only once to produce the final application image. LEP_x_SCRIPTLINKER.vi must enumerate all intermediate object files, external files, run-time libraries, and operating system libraries.

Some targets use prjObjs.lst to get a list of all modules to avoid long command lines. LEP_x_BUILD.vi generates prjObjs.lst and lists all binary modules and libraries.

VxWorks module targets do not link with the operating system libraries because the VxWorks module loader resolves all symbols. The VxWorks and eCos example targets do not have pre-compiled run-time libraries, so LEP_x_SCRIPTLINKER.vi also must link with the run-time library intermediate files. LEP_x_BUILD.vi executes the default implementation of the LEP_x_SCRIPTLINKER.vi unless you override the default implementation and create a custom LEP_x_SCRIPTLINKER.vi.

You can override the default implementation by wiring a LEP_x_SCRIPTLINKER.vi VI reference to the Linker Script input of LEP_x_BUILD.vi. Your custom implementation of LEP_x_SCRIPTLINKER.vi must have the same connector pane as the default implementation.
LEP_x_ScriptLinker.vi can execute the C linker directly by calling the System Exec VI. You also can use a batch file to directly execute the linker. Use a batch file if you need to set linker-specific system variables before the linker can execute. For example, the VxWorks example targets use a batch file because you must set the Tornado system variables before the C compiler executes.

For most targets, the base target provides a complete, tools-aware embedded OS build framework with the default implementation of LEP_x_ScriptLinker.vi. You can use the base target default implementation of the build framework and LEP_x_ScriptLinker.vi and override just the necessary features.

The VxWorks CMD565 example target uses a subtarget-specific implementation, which is located in the following file:

labview\Targets\NI\Embedded\vxworks\cmd565\vxworks_cmd565_LEP_TargetPlugin\LEP_vxworks_cmd565_Build.vi

This implementation overrides the default implementation of the LEP_x_ScriptCompiler.vi, LEP_x_ScriptLinker.vi, and LEP_x_Config.vi. Notice the Case structure with target-specific VIs in this custom implementation. The VIs in the Case structure never execute but are always loaded into memory. Because the VIs are loaded into memory, you can use the Open VI Reference function to refer to VIs by name instead of absolute path.
Each LabVIEW function that does not map directly to a C primitive is implemented using a function call. These functions are implemented in the LabVIEW Embedded Run-Time Library on top of a lower layer of generic functions that handle memory movement, string manipulation, timing, and so on.

One of the major components to porting the run-time library is the mapping of these generic, low-level function calls to functions that are supported by your target run-time environment. You must provide this mapping so your toolchain can compile the generated C code for your target platform.

The source code for the LabVIEW Embedded Run-Time Library is located in the following directory:

```
labview\CCodeGen\libsrc
```
Directory Structure

Figure 7-1 shows the directory structure of the run-time library source code.

![Directory Structure Diagram]

The header files for the run-time library source code are located in the following directory:

```
labview\CCodeGen\include
```

The directory structure of the `include` directory mirrors the directory structure of the `libsrc` directory shown in Figure 7-1.

Portable Pieces of the Run-Time Library

With the exception of the `os` directory, you usually do not have to modify the code in the `labview\CCodeGen\libsrc` subdirectories. The C files in these directories contain `#include` directives for a header file that contains `#include` directives for an OS-specific header file. The OS-specific header file contains macros and definitions.
Chapter 7  Porting the LabVIEW Embedded Run-Time and Analysis Libraries Source Code

The labview\CodeGen\libsrc\blockdiagram directory contains code for the following:

- arrays
- complex numbers
- clusters
- floating-point numerics
- file support
- integers
- number to text
- strings

The labview\CodeGen\libsrc\comms directory contains code for serial and TCP/IP communications.

Note  The labview\CodeGen\libsrc\comms directory also contains code for Bluetooth and IrDA, which has been ported only to the Pocket PC and Windows Mobile platforms.

The labview\CodeGen\libsrc\frontpanel directory contains code for front panel terminals only. The directory does not contain user interface components to display the terminals. The Embedded Development Module supports the following controls:

- Array
- Boolean
- Chart
- Cluster
- Enumerated type
- Graph
- Listbox
- Numeric
- Picture
- Ring
- String
- Tab
Chapter 7  Porting the LabVIEW Embedded Run-Time and Analysis Libraries Source Code

LVEmbeddedMain.c

LVEmbeddedMain.c contains the main entry point for all applications the LabVIEW C Code Generator generates. This file contains functions that perform set up and tear down of common pieces, such as occurrences and FIFOs. LVEmbeddedMain.c uses #ifdefs rather than separate files for each OS. The main function in LVEmbeddedMain.c initializes all global variables and then calls the top-level VI in the project. After the top-level VI has completed, a shutdown routine is completed. The following two macros define the initialization and shutdown routines:

- LV_PLATFORM_INIT
- LVPLATFORMFINI

These macros are defined per target rather than per operating system. The labview\CCodeGen\libsrc\os folder hierarchy does not define LV_PLATFORM_INIT and LV_PLATFORM_FINI because a one-to-one mapping does not exist between operating systems and targets. For example, an eCos target on a Nintendo Gameboy Advanced device needs a certain LV_PLATFORM_INIT to set up the hardware, but eCos on the CMD565 does not.

If you define LV_PLATFORM_INIT as a function, LV_PLATFORM_INIT is called before any VI is called and before any other setup is performed. The target defines this function and you can implement it in any way that is appropriate for your target. If you define LV_PLATFORM_FINI as a function, LV_PLATFORM_FINI is called after all VIs have finished and any other tear down is complete.

Include Files

The LabVIEW C Code Generator run-time requires two main include files:

- LVSysIncludes.h
- LVDefs_plat.h

Each supported OS contains a version of these two include files, which are located in the following directory:

labview\CCodeGen\include\os
LVSysIncludes.h

Contains operating system includes that all or most of the source files need. LVSysIncludes.h is included everywhere in the LabVIEW C Code Generator run-time code.

LVDefs_plat.h

Contains platform defines for data types, macros that convert generic names to OS-specific names, and feature flags that define which features are implemented.

#defines

A series of #define statements map the generic function calls, such as StrCaselessCompare (x, y) to OS-specific function calls, such asstrcasecmp (x, y), for Linux and VxWorks and _stricmp (x, y) for Windows.

Examples of some of the #defines in LVDefs_plat.h include the following:

- typedef signed char int8;—In the LabVIEW C Code Generator run-time code, int8 is used everywhere and must be defined to a signed 8-bit quantity.
- #define StrCopy(x, y) strcpy(x, y)—In the LabVIEW C Code Generator run-time code, StrCopy is used everywhere and must be defined to a function that copies a string.
- #define SocketSupport 1—Defines the sockets. Therefore, TCP/IP functions are supported on this platform.
- #define BigEndian—Defines the data storage format as big endian or little endian.
Feature Flags

The following flags in \texttt{LVDefs\_plat.h} turn on and off large sets of features that correspond to pieces of hardware that might or might not be applicable for your target.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HeapListSupport</td>
<td>Controls the freeing of memory blocks associated with VIs. If this flag is TRUE, the memory blocks are not freed until the embedded application exits, which speeds up application execution but greatly increases the memory footprint.</td>
</tr>
<tr>
<td>BluetoothSupport</td>
<td>Controls Bluetooth support. Bluetooth support has not been ported to any of the embedded example targets and operating systems, but the code is present.</td>
</tr>
<tr>
<td>IrDA Support</td>
<td>Controls infrared (IrDA) support. IrDA support has not been ported to any of the embedded example targets and operating systems, but the code is present.</td>
</tr>
<tr>
<td>Serial Support</td>
<td>Controls serial communication support.</td>
</tr>
<tr>
<td>TCPUDP Support</td>
<td>Controls TCP and UDP support along with the \texttt{SocketSupport} flag in \texttt{CCGNetConnSupport.c}.</td>
</tr>
<tr>
<td>File Support</td>
<td>Controls file I/O support.</td>
</tr>
</tbody>
</table>

OS-Specific Components

The following functionality is OS-specific. The generated C code calls these OS-specific functions when various features are on the block diagram. If you do not want to port certain parts of this OS-specific code to a new platform, you must avoid using features that depend on the unported parts to avoid linker errors. Look at each file and determine if you must rewrite the file for a new platform. A copy of each file is located in each operating system folder in the following directory:

\begin{verbatim}
labview\CCodeGen\libsrc\os
\end{verbatim}
Critical Sections: \texttt{LVCritSect.c}

Occurrences, ISRs, Real-Time FIFO VIs, and Timed Loops depend on \texttt{LVCritSect.c}, which contains the following functions:

\begin{itemize}
\item \texttt{InitLVCriticalSections()} \texttt{UninitLVCriticalSections()}
  \textbf{Purpose}: Called from \texttt{LVEmbeddedMain.c} to initialize and tear down critical sections.
\item \texttt{LVCriticalSection \texttt{LVCriticalSectionCreate}()}
  \textbf{Purpose}: Allocates the critical section record and returns a pointer to it.
\item \texttt{LVCriticalSectionDelete(LVCriticalSection \texttt{critsect})}
  \textbf{Purpose}: Frees a critical section record.
\item \texttt{LVCriticalSectionEnter(LVCriticalSection \texttt{critsect})}
  \textbf{Purpose}: Called when a section of code is entered that must not be interrupted.
\item \texttt{LVCriticalSectionLeave(LVCriticalSection \texttt{critsect})}
  \textbf{Purpose}: Called when the critical section is finished.
\end{itemize}

Events: \texttt{LVEvent.c}

\textbf{Note} The event functions are real-time events. The functions are not UI events.

Occurrences and Timed Loops depend on \texttt{LVEvent.c}, which contains the following functions:

\begin{itemize}
\item \texttt{InitLVEvents()}, \texttt{UninitLVEvents()}
  \textbf{Purpose}: Called from \texttt{LVEmbeddedMain.c} to initialize and tear down events.
\item \texttt{LVEvent \texttt{LVEventCreate}()}
  \textbf{Purpose}: Allocates an event record and returns a pointer to it.
\item \texttt{LVEventDelete(LVEvent \texttt{event})}
  \textbf{Purpose}: Frees an event record.
\end{itemize}
Chapter 7 Porting the LabVIEW Embedded Run-Time and Analysis Libraries Source Code

LVEventSet(LVEvent event)

**Purpose:** Triggers an event. All threads waiting on the event proceed with execution.

LVEventReset(LVEvent event)

**Purpose:** Resets an event so that subsequent waiters must wait for another LVEventSet call.

LVEventWait(LVEvent event, int32 millisec_timeout, Boolean *timedout)

**Purpose:** Waits on an event with a timeout. Suspends the executing thread until the event is set.

**Threads: LVThreads.c**

Timed Loops use LVThreads.c, which contains the following functions:

InitLVThreads(), UninitLVThreads()

**Purpose:** Called from LVEmbeddedMain.c to initialize and tear down threads.

LVTHREAD_HANDLE LVThreadCreate(void* startAddress, LVTHREAD_PRIORITY priority, void* params)

**Purpose:** Creates a thread and returns a handle to it.

LVThreadActivate(LVTHREAD_HANDLE h)

**Purpose:** On some operating systems, threads are created in a suspended state. This function activates and runs a thread. This function might be a no-op if LVThreadCreate creates and activates threads.

LVThreadGetCurrentThreadId()

**Purpose:** Returns the ID of the thread from which the function is called.

LVThreadSetPriority(LVTHREAD_HANDLE h, LVTHREAD_PRIORITY p)

**Purpose:** Sets the thread priority.

LVThreadSetCurrentPriority(LVTHREAD_PRIORITY p)

**Purpose:** Sets the priority of the thread from which this function is called.
LVThreadGetPriority(LVTHREAD_HANDLE h)

**Purpose:** Returns the priority of the thread referred to by the handle passed in.

LVTHREAD_PRIORITY LVThreadGetCurrentPriority()

**Purpose:** Returns the priority of the thread from which this function is called.

LVTHREAD_PRIORITY LVThreadGetInitPriority()

**Purpose:** Returns the priority when the thread from which this function was called was created.

LVTHREAD_PRIORITY LVThreadGetMainPriority()

**Purpose:** Returns the priority of the main thread.

void LVThreadSleep_ms(unsigned long ms)

**Purpose:** Suspends the thread for the time interval passed in.

### Non-Blocking Operations: LVNBOps.c

Non-blocking operations consist of atomic compare-exchange operations the RT-FIFO VIs use. Use machine instructions, if available, to implement the atomic compare-exchange operations. LVNBOps.c contains the following functions:

LVNBOpsAtomicCompareExchange(uInt32 *inspectedLocation, uInt32 oldValue, uInt32 newValue)

LVNBOpsAtomicCompareExchange16(uInt16 *inspectedLocation, uInt16 oldValue, uInt16 newValue)

LVNBOpsAtomicCompareExchange8(uInt8 *inspectedLocation, uInt8 oldValue, uInt8 newValue)

### LabVIEW-Based Interrupt Service Routines: OEM_LVISR.c

OEM_LVISR.c implements LabVIEW-based ISRs. OEM_LVISR.c contains the following functions:

OEMISRBoilerPlate(int param)

**Purpose:** Finds the ISR VI in the lookup table that param indexes.
InitOEMISR(), UninitOEMISR()

**Purpose:** Called from LVEmbeddedMain.c to initialize and tear down ISR VIs.

OEMISRRegisterHandler(uInt32 isr_vector, uInt32 isr_param, ISRFunc isr_runFunc, uInt32 *register_param)

**Purpose:** Searches for an empty slot in the lookup table, places ISRFunc in the table, and returns the index of the slot in the lookup table.

OEMISRUnregisterHandler(uInt32 isr_vector, uInt32 isr_param, ISRFunc isr_runFunc, uInt32 *register_param)

**Purpose:** Removes the ISR VI from the lookup table.

**Printf: PDAStrSupport_OS.c**

The One Button Dialog function uses the printf functionality. PDAStrSupport_OS.c implements printf.

**Time: CCGTimeSupport_OS.c**

CCGTimeSupport_OS.c implements timing. It contains the following functions:

uInt32 LVGetTicks()

**Purpose:** Returns clock ticks in milliseconds. You must port this function because this function is used throughout the run-time libraries and generated C code.

Boolean DtToSecs( VoidPtr vpIn, DataType dt, uInt32 *pSecs )

**Purpose:** Converts a cluster containing date and time to seconds. This function is called directly from the generated C code to implement the Date Time to Seconds function.

Boolean SecsToDt( void* pSecs, DataType dt, VoidPtr vpOut, DataType dtOut )

**Purpose:** Converts seconds to a cluster containing date and time. This function is called directly from the generated C code to implement the Seconds to Date Time function.
Boolean LVGetDateTime(double dSecs, LVDateTime *pDateTime)

**Purpose**: Converts a time in seconds to date and time. If the seconds passed in is zero, this function returns the current time. This function is called directly from the generated C code to implement the Get Date Time function.

**Serial: PlatformSerial_OS.c**

Use PlatformSerial_OS.c for serial functions and for debugging over a serial connection. PlatformSerial_OS.c contains the following functions:

SerialInit(SerialDevice ser, SerialConfig *cfg)

**Purpose**: Initializes the serial driver.

SerialOpen(int portNumber, SerialDevice *serptr)

**Purpose**: Opens a serial port.

SerialClose(SerialDevice ser)

**Purpose**: Closes a serial port.

SerialRead(SerialDevice ser, char *buffer, int *length)

**Purpose**: Reads bytes from a serial port.

SerialWrite(SerialDevice ser, const char *buffer, int *length)

**Purpose**: Writes bytes to a serial port.

SerialBytesAvail(SerialDevice ser, int *bytes)

**Purpose**: Returns the number of bytes available from a serial port.

SerialBreak(SerialDevice ser)

**Purpose**: Sends a break on a serial line.
CAN: PlatformCAN_OS.c

Use PlatformCAN_OS.c for the embedded CAN VIs and for debugging over a controller area network (CAN) connection. PlatformCAN_OS.c contains the following functions:

MgErr LvCanOpen(uInt32 boardIdx, uInt32 controllerIdx, uInt32 *reference);

**Purpose:** Opens a session to a CAN controller.

MgErr LvCanClose(uInt32 reference);

**Purpose:** Closes an open session to a CAN controller.

MgErr LvCanGetChannel(uInt32 reference, uInt8 rx, uInt8 *pChannelNum);

**Purpose:** Returns an open channel to transmit or receive CAN message frames.

MgErr LvCanFreeChannel(uInt32 reference, uInt8 channelNum);

**Purpose:** Frees a channel you open with LvCanGetChannel.

MgErr LvCanStart(uInt32 reference);

**Purpose:** Starts communicating with the CAN controller.

MgErr LvCanStop(uInt32 reference);

**Purpose:** Stops a CAN controller from sending or transmitting messages.

MgErr LvCanRead(uInt32 reference, uInt8 channelNum, int32 *bNewData, int32 *canID, uInt8 *len, uInt8 *data);

**Purpose:** Reads a CAN message frame from the specified channel.

MgErr LvCanWrite(uInt32 reference, uInt8 channelNum, uInt32 canID, uInt8 len, uInt8 *data);

**Purpose:** Writes a CAN message frame to the specified channel.

MgErr LvCanGetGlobalFilter(uInt32 reference, int32 *filter);

**Purpose:** Retrieves the filter you set with LvCanSetGlobalFilter.
MgErr LvCanSetGlobalFilter(uInt32 reference, int32 filter);
    
    **Purpose:** Sets the global filter for CAN messages.

MgErr LvCanGetChannelFilter(uInt32 reference, uInt8 channel, int32 *filter);
    
    **Purpose:** Returns the channel filter you set with LvCanSetChannelFilter.

MgErr LvCanSetChannelFilter(uInt32 reference, uInt8 channel, int32 filter);
    
    **Purpose:** Sets the CAN message filter for the specified open channel.

MgErr LvCanGetBaudRate(uInt32 reference, uInt32 *baudRate);
    
    **Purpose:** Returns the current baud rate of the CAN controller.

MgErr LvCanSetBaudRate(uInt32 reference, uInt32 baudRate);
    
    **Purpose:** Sets the baud rate of the CAN controller.

MgErr LvCanMessageAvail(uInt32 reference, uInt8 channel, uInt8 *bAvail);
    
    **Purpose:** Polls channels for new CAN messages to read.

**TCP/UDP: CCGNetConnSupport.c**

TCP communication is implemented on top of Berkeley sockets on all platforms and is used by TCP functions, UDP functions, remote Call By Reference Nodes, and TCP-based debugging. All Embedded Development Module example targets support Berkeley sockets. The implementation file is CCGNetConnSupport.c, which is located in the labview\CodeGen\libsrc\comms directory. CCGNetConnSupport.c may be OS-specific even though the file is located in the comms folder.

SocketSupport is the flag to turn on or off TCP/UDP support. #define SocketSupport to be 1 to turn on support and 0 to turn off support.
Static Memory Allocation Support

If you want to support embedded applications that allocate no memory, you must define some constants to determine the maximum number of items associated with various features for your target. The following tables lists the constants you must use to statically declare these items.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENERATE_OCCURRENCE_MAX</td>
<td>Defines the maximum number of Generate Occurrence functions.</td>
</tr>
<tr>
<td>WAIT_ON_OCCURRENCE_MAX</td>
<td>Defines the maximum number of Wait on Occurrence functions.</td>
</tr>
<tr>
<td>ISR_HANDLE_MAX</td>
<td>Defines the maximum number of ISR handler VIs.</td>
</tr>
<tr>
<td>CRITICAL_SECTION_MAX</td>
<td>Defines the maximum number of critical section handles.</td>
</tr>
<tr>
<td>EVENT_MAX</td>
<td>Defines the maximum number of events.</td>
</tr>
<tr>
<td>NODE_MAX</td>
<td>Defines the maximum number of external time sources.</td>
</tr>
<tr>
<td>FIFO_MAX</td>
<td>Defines the maximum number of Real-Time FIFOs.</td>
</tr>
<tr>
<td>FIFO_SIZE_MAX</td>
<td>Defines the maximum size of an Real-Time FIFO in elements.</td>
</tr>
</tbody>
</table>

Analysis Library

Note Targets must support floating-point operations either natively or though an emulator library to support the LabVIEW analysis library.

To port the analysis library, look for an OS similar to your target OS and search the `#ifdefs` for that OS. Most of the porting work is done in the `platdefines.h` include file located in the following directory:

```
labview\CCodeGen\analysis\development\include
```

You can port the analysis library with or without BLAS/LAPACK, which is a set of highly optimized, open source math functions widely used in industry and academia. The Embedded Development Module does not include BLAS/LAPACK because BLAS/LAPACK makes analysis functionality very large. BLAS/LAPACK is more appropriate for desktop PCs where large amounts of memory are available. Define `COMPILE_FOR_SMALL_RT` to remove BLAS/LAPACK support.
The following directory contains the analysis source code to convert LabVIEW data types to C data type:

```
labview\CCodeGen\analysis\LV\source
```

The following directory contains the actual analysis algorithms:

```
labview\CCodeGen\analysis\development\source
```

The C++ code in the LabVIEW analysis library does not use constructors and destructors. Ignore the peak detector and pulse duration functions because these functions assume LabVIEW internals that only apply to LabVIEW for Windows. The peak detector and pulse duration functions will crash and do not work on embedded targets.

You also must set the following #defines:

```
LV_Embedded=1
NoFFTtablePersist=1
LVANALYSIS=1
ThreadsUsed=0
COMPILE_FOR_SMALL_RT=1
```

**Note** The analysis library does have some thread-safe guards that require more support functions. Thread-safe guards are not implemented in this release of the Embedded Development Module.

The analysis library relies on LabVIEW.lib, which is another library that you must build. LabVIEW.lib provides an interface between LabVIEW and the analysis library. The source for the LabVIEW.lib source is located in the following file and contains functions for memory management and resizing of arrays.

```
labview\CCodeGen\libs\source\lvanalysis\arrresize.c
```
Using FuncList.dat to Improve Performance

You can use FuncList.dat to override subVI calls with C function calls. For example, the Serial Compatibility VIs are supported on embedded targets. For Windows targets, the Serial Compatibility VIs are implemented on top of VISA, which embedded targets do not support. For embedded targets, the Serial Compatibility VIs have an alternate implementation that makes direct calls to the LabVIEW C Code Generator run-time library. Adding entries to FuncList.dat is how you provide an alternative implementation. Using FuncList.dat removes the overhead of subVI calls and can produce substantial performance improvements in embedded applications you create with LabVIEW. SubVI calls have overhead to support multithreading, reentrancy, front panel controls and indicators, and so on.

FuncList.dat is useful when you wrap high performance, target-specific C function calls into subVIs. Creating subVI wrappers provide a better experience in LabVIEW so you can have the best of both—the ease of use of LabVIEW and the performance of an optimized C function call. Refer to the LabVIEW Embedded Development Module Target Distribution Guide, available by selecting Start»All Programs»National Instruments»LabVIEW»LabVIEW Manuals and opening EMB_Distribution_Guide.pdf, for information about additional ways to wrap C functions so the functions are easier to use in LabVIEW.

⚠️ Caution ⚠️ National Instruments originally created FuncList.dat for internal use only. Entries in FuncList.dat have specific formatting requirements and contain little error checking. If the formatting is incorrect, you receive compiler errors and LabVIEW might crash when you build an embedded VI into an embedded application. Refer to the Format of FuncList.dat section in this chapter for information about the formatting requirements.
Adding Entries to FuncList.dat

Complete the following steps to add entries to FuncList.dat.

1. Open FuncList.dat, which is located in the labview\CodeGen directory, in any text editor.
2. If necessary, modify the file so FuncList.dat is not read-only.
3. Add your FuncList.dat entry to the top of the file.
4. Save FuncList.dat.
5. Restart LabVIEW to reload FuncList.dat.

You can now use the subVI you added to FuncList.dat as any other subVI.

Format of FuncList.dat

FuncList.dat contains lines that describe how to replace a subVI call in the generated C code with a fragment of C code. The fragments can be anything but are usually function calls. FuncList.dat has specific formatting requirements. Each FuncList.dat entry must contain three lines.

Note The LabVIEW C Code Generator interprets any line that begins with # as a comment.

1st Line—VI Names

The first line of a FuncList.dat entry is the VI name. If you add an entry to FuncList.dat, you must change the VI name in the following ways:
- Convert spaces to underscores.
- If the VI begins with an underscore, prepend A__ to the VI name.

2nd Line—C Code

The second line of a FuncList.dat entry contains the line of C code you want inserted in the file. If you call a function, the second line contains the function parameters. You can either pass the parameters directly or use ArgList.
Passing Parameters Directly

Passing parameters for C function calls in FuncList.dat have the following requirements:

- Place an @ character at the beginning of the line of C code.
- Use a format code in the form @xn to access the subVI inputs and outputs. x represents a character that determines if the parameter is an input or output and the way the parameter is passed. Refer to the following bullet for more information about how you indicate the way the parameter is passed. n is the number of the input or output on the connector pane. The numbering on the connector pane is not intuitive. You might have to use trial and error to find the correct number.
- You indicate how the parameter is passed with the following characters:
  - I is an input passed by reference.
  - i is an input passed by value.
  - o is an output passed by reference.
  - it is the type of input.
- Place a value in parentheses after a format code to specify a default, or unwired, value for the input. For example, if you want to pass input 3 by reference with Null as the default value, the syntax is @I3(Null).

Passing Parameters by ArgList

Using ArgList to pass parameters has the following requirements:

- Do not place an @ at the beginning of the line of C code.
- Specify %s for the ArgList input and a second %s for the ArgList output.
- The function you call must unpack the input and output values from ArgList.

The ArgList struct declaration is located in the following header file:

```
labview\CCodeGen\Include\blockdiagram\LVCCG.h
```

3rd Line—Feature Section

The third line of a FuncList.dat entry contains a string. You can use any string for the third line as long as the string is a valid C identifier and the string is unique.
Chapter 8  Using FuncList.dat to Improve Performance

Debugging

If you change the implementation of a subVI to use FuncList.dat, you must debug the subVI in C using your toolchain. You cannot debug the FuncList.dat subVIs in LabVIEW.

FuncList.dat contains very little error reporting.

Troubleshooting

If you create a FuncList.dat entry and the generated C code is still generating the subVI call instead of the C function call, verify you restarted LabVIEW to reload FuncList.dat. After you restart LabVIEW, verify the subVI name in FuncList.dat. Refer to the 1st Line—VI Names section in this chapter for information about how to specify VI names in FuncList.dat.

If you receive compile errors when you build an embedded VI into an embedded application, verify the format of the C function call and the order of the parameters in FuncList.dat. Refer to the 2nd Line—C Code section in this chapter for information about the required formatting for passing parameters in a C function call.

Example

National Instruments has implemented many subVIs for the Embedded Development Module using FuncList.dat, which is located in the following directory:

labview\CCodeGen
Elemental I/O

Overview

Elemental I/O is a flexible, user-defined way to create single-point I/O for embedded LabVIEW targets. Elemental I/O Nodes are portable across many targets. Although the I/O implementation might change among targets, you do not need to redevelop the application for another target. The block diagram can stay the same because the Elemental I/O Node itself does not change. Use Elemental I/O Nodes to create block diagrams that represent algorithms that you can reuse on many platforms.

Compare Elemental I/O Nodes to Call Library Function Nodes or register-level programming where driver differences between embedded targets are fully exposed on the block diagram. Elemental I/O requires more work to implement and set up, but provides greater reuse than Call Library Function Nodes or register-level programming.

Elemental I/O consists of the following:
- I/O devices
- Elemental I/O classes
- Pins
- Resources
- Plug-in VIs

I/O Devices

An I/O device is the software representation of hardware that performs I/O on an embedded target. An I/O device might be the CPU on a target or a separate piece of hardware on the same board. You can reuse an I/O device on multiple targets so several similar targets can share a single I/O device. For example, a PowerPC 565 ROM target and a PowerPC 565 RAM target might share a single I/O device.
Elemental I/O Classes

Elemental I/O supports the following I/O classes:

- Analog In
- Analog Out
- Digital In
- Digital Out
- Digital Bank In
- Digital Bank Out
- Pulse Width Modulation Out

You can implement some or all of the supported I/O classes for an I/O device.

Pins

A pin is the software equivalent of a physical pin that connects an I/O device to the outside world. Resources can share pins because typically, there are fewer pins than resources.

Resources

An Elemental I/O resource is the software equivalent of a circuit in the hardware that performs I/O. Resources use pins to connect the circuits to the outside world. An I/O device can have many resources.

Elemental I/O resources bind the pins to the I/O classes. For example, the hardware an Elemental I/O device uses might have eight general purpose I/O pins that you can use for digital input, digital output, analog input, and analog output. You can create four resources for each pin and bind the pin to the respective class of I/O.

Plug-In VIs

The Elemental I/O plug-in VIs enforce how users can use pins and resources when configuring and using Elemental I/O Nodes and Elemental I/O Property Nodes. Refer to the Implementing the Elemental I/O Plug-In VIs section of this chapter for more information about plug-in VIs.
How Users Use Elemental I/O Nodes

Users create Elemental I/O items in the Project Explorer window by right-clicking a VI and selecting New » Elemental I/O from the shortcut menu. Users can then drag the Elemental I/O item from the Project Explorer window to the block diagram.

Users configure an Elemental I/O Node by right-clicking the node on the block diagram and selecting Properties from the shortcut menu. Users select configuration parameters you implement in the Elemental I/O implementation VIs.

When the Elemental I/O Node executes on the embedded target, it executes LabVIEW code the Elemental I/O device defines. The Elemental I/O device has plug-in VIs for each type of I/O resource, which replace the Elemental I/O Node when executing on the embedded target. Most of the plug-in VIs have a resource name or channel number as an input, and compute the I/O result using an Inline C Node that accesses the I/O registers.

Creating Elemental I/O

To create the Elemental I/O for a target, you must create the plug-in VIs and then run the Elemental I/O Device Wizard.

Creating Elemental I/O Plug-In VIs

Use the plug-in VIs to define the behavior of I/O operations. You must create the plug-in VIs before you run the Elemental I/O Device Wizard because the wizard uses information from these VIs. If the plug-in VIs are modified in any way, you must run the wizard again to ensure that the information in the target matches the VIs.

Use the template VIs in the following directory to create plug-in VIs.

labview\examples\lvemb\EDM\EIO\EIO User Plugin Templates

The template VIs include all possible controls and indicators for the plug-in VIs. Copy the template VIs for the target, replace the Project Attributes and Node Attributes cluster controls with type definitions or strict type definitions, and remove other controls that you do not need.
The **Project Attributes** cluster control is what appears on the **I/O Node Project Panel** page of the **Elemental I/O Properties** dialog box when the user right-clicks an I/O item in the **Project Explorer** window and selects **Properties** from the shortcut menu. The values set in the **I/O Node Project Panel** page apply to all nodes that refer to this I/O item. The project attributes are passed to the I/O implementation VIs and the I/O Property Node Implementation VIs for this I/O item.

The **Node Attributes** control is what appears on the **I/O Node Panel** page of the **Elemental I/O Node Properties** dialog box when the user right-clicks an Elemental I/O Node on the block diagram and selects **Properties** from the shortcut menu. The values set in the **I/O Node Panel** page apply to this particular Elemental I/O Node only. Node attributes are passed to the I/O Implementation VIs.

**Note** All plug-in VIs use the standard 12 terminal connector pane. You must wire controls and indicators as specified by the connector pane of each template VI.

Refer to the following directory for example plug-in VIs:

`labview\examples\lvemb\EDM\EIO\EIOSimulationExample`

**I/O Node Implementation Plug-In VI**

The I/O Node Implementation VI is the only required plug-in VI. You must have at least one I/O Node Implementation VI for each I/O resource you define. If you have multiple I/O Node Implementation VIs, the I/O Specify Node VI determines which I/O Node Implementation VI to call. The I/O Node Implementation VI runs on the embedded target, so anything that is valid for the target is allowed in this VI.

Logically, the I/O Node Implementation VI behaves as if the Elemental I/O Node has been replaced with a call to this VI with constants wired for the attributes and properties. The project attributes and the node attributes are passed to the I/O Implementation VIs for the I/O item.

**I/O Property Node Implementation Plug-In VIs**

If you want to support setting properties dynamically, you must include a I/O Property Node Implementation VI for each property. The I/O Property Node Implementation VIs are called where the user drops an Elemental I/O Property Node on the block diagram. These VIs can communicate with hardware, call device drivers, set global variables used by the I/O
Implementation VI, and so on. These VIs run on the embedded target, so anything that is valid for the target is allowed in these VI. The I/O Property Node Implementation VI is passed the project attributes for the selected I/O item.

**I/O Specify Node Plug-In VI**

Use the I/O Specify Node VI if you want to have Elemental I/O Nodes whose type is determined at edit time, as opposed to when you create the I/O. The VI returns the type of the I/O Node, which is either the return type for Elemental I/O Nodes configured to read or the input type for Elemental I/O Nodes configured to write. If you use this VI, you must include an I/O Node Implementation VI for each type returned.

If you have multiple I/O Node Implementation VIs, you must include an I/O Specify Node VI to determine which one to use. If you have an I/O Implementation VI for reading and an I/O Implementation VI for writing with different data types, you must include an I/O Specify Node VI. These VIs run at edit time on the host.

**I/O Project and Node Attribute Validation Plug-In VIs**

Use these VIs to validate the attributes of the I/O item in the Project Explorer window and the Elemental I/O Nodes on the block diagram. These VIs run at edit time on the host.

**Creating Elemental I/O Devices, Classes, Resources, and Pins**

After you create the plug-in VIs, use the Elemental I/O Device Wizard to define Elemental I/O devices, classes, resources, and pins for an embedded target. The wizard determines which I/O resources are available for an embedded target, what attributes and I/O properties those resources have, and which VIs the Elemental I/O Node and Elemental I/O Property Node call to perform I/O operations.

Complete the following steps to run the Elemental I/O Device Wizard.

1. Select **Tools»Embedded Tools»Elemental I/O Device Wizard** to display the Elemental I/O Device Wizard.
2. On the Name the Device page of the wizard, enter the name of the new device.

*Note* You also can click the **Load existing I/O target** button to select an existing device.
3. Click the **Next** button to display the **Select the Elemental I/O Classes** page.

4. Select the I/O classes the device supports. The I/O classes only affect the icon that appears on the I/O Node and the name of the I/O folder in which the resources show up in the project. The I/O classes have no functional effect on the I/O.

5. Click the **Next** button to display the **Add I/O Resources** page.

6. Add resources. Use this page of the wizard to name the I/O resources. These are the default names that appear when a user right-clicks the embedded target in the **Project Explorer** window and selects New» **Elemental I/O** from the shortcut menu.

7. Click the **Next** button to display the **Add Pins** page.

8. Define the physical pins on the device. Use pins to reserve shared resources. You might have physical pins on the device that can be used for analog or digital input but only one can be configured. In this case, you can name these pins and assign them to the I/O resources. Once a user adds an I/O resource that uses a pin, the user cannot add any other I/O resource that uses that same pin. If you do not need to reserve shared resources, do not create any pins.

9. Click the **Next** button to display the **Assign the Pins to Resources** page.

10. Define the connection between the pins and the resources you created. You can assign multiple pins to each resource, and you can assign the same pin to multiple resources. If you do not need to reserve shared resources, do not assign any pins to the I/O resources.

11. Click the **Next** button to display the **Add Implementation VIs to Resources** page.

12. Associate the plug-in VIs with the resources. For each I/O resource, you must have at least one read or write I/O Implementation VI. All other VIs are optional.

13. Click the **Next** button to display the **Generating Elemental I/O VIs and Resource File** page.

14. Click the **Generate** button. The wizard creates a directory in the location you specify in the **Output Directory** and adds several VIs and an `eio.xml` file in the new directory.

15. Click the **Finish** button to exit the wizard.
Adding Elemental I/O Devices to an Embedded Target

Complete the following steps to add an Elemental I/O device to an embedded target.

1. Select Tools » Embedded Tools » Target Editor to display the Target Editor window.

2. Select File » Open to browse to an existing target or select File » New to create a new target.

3. Right-click a target and select Properties from the shortcut menu to display the Build Specification Type Properties dialog box.

4. On the General tab, enter the path to the eio.xml file you created above in the Path to Elemental I/O configuration file field.

5. Click the OK button.

6. Select File » Save in the Target Editor window to save the target.

7. Restart LabVIEW to start using the Elemental I/O in the target.

Example

The PHYTEC and Simulated I/O targets provide Elemental I/O implementation examples.
10

Implementing Interrupt Service Routine (ISR) Support

If you want to support interrupt service routines (ISRs) for your target, you must implement a set of functions in OEM_LVISR.c, which is located in the following directory:

```
labview\CCodeGen\libsrc\os\<OS>
```

where `<OS>` is the target operating system. The function prototypes are defined in LVISR.h, which is located in the following directory:

```
labview\CCodeGen\Include\blockdiagram
```

### ISR Functions

You must implement the following functions:

#### InitOEMISR

```
Boolean InitOEMISR();
```

**Purpose:** This function is called only by InitISR. It must do any required ISR subsystem initialization.

- Returns TRUE on success.
- Returns FALSE on failure.

#### UninitOEMISR

```
Boolean UninitOEMISR();
```

**Purpose:** This function is called only by UninitISR. It must do any necessary ISR subsystem cleanup.

- Returns TRUE on success.
- Returns FALSE on failure.
**OEMISRRegisterHandler**

Boolean OEMISRRegisterHandler (uInt32 isr_vector, uInt32 isr_param, ISRFunc isr_runFunc, uInt32 *register_param);

**Purpose:** This function is called only by ISRRegisterHandler. It performs the actual registration of the interrupt with the OS or with the hardware in the case of bare metal. \texttt{isr\_vector} is the intended vector, \texttt{isr\_param} is the parameter. Both \texttt{isr\_vector} and \texttt{isr\_param} are to be passed to the \texttt{ISRFunc isr\_runFunc}. \texttt{register\_param} is an output parameter that is unique to this ISR registration instance. The \texttt{register\_param} value is passed to the OEMISRUnregisterHandler function when the ISR is being unregistered.

Returns TRUE on success.
Returns FALSE on failure.

**OEMISRUnregisterHandler**

Boolean OEMISRUnregisterHandler (uInt32 isr_vector, uInt32 isr_param, ISRFunc isr_runFunc, uInt32 register_param);

**Purpose:** This function is called only by ISRUnregisterHandler. This function unregisters an ISR with the OS or with the hardware in the case of bare metal. \texttt{register\_param} is the register parameter that is returned by the OEMISRRegisterHandler routine.

Returns TRUE on success.
Returns FALSE on failure.

**Tips for Supporting ISRs on a New Platform**

The OS and/or hardware have specific requirements for the prototype of the ISR routine that likely does not match the requirements for an ISR VI. Write a function that is the actual ISR that calls the ISR VI with the correct parameters. You might need to create a global data structure to hold the parameters to pass to the ISR VI.

The ISR VI parameters are meant to be general enough to handle most hardware and OS requirements. Depending on the platform, you might not need these parameters. If you do not need these parameters, you can optimize the ISR by passing zeros for the required parameters and have the ISR VI ignore them.
Refer to the VxWorks example implementation, located in the following directory, for an example:

```
labview\CCodeGen\libsrc\os\vxworks\OEM_LVISR.c
```
11

Timing

A Timed Loop executes a subdiagram, or frame, for each iteration of the loop at the period you specify. Use the Timed Loop when you want to develop VIs with multi-rate timing capabilities, precise timing, feedback on loop execution, timing characteristics that change dynamically, or several levels of execution priority.

Note Embedded targets do not support timing source hierarchies in Timed Loops.

Using the Target Editor to Define Timing Sources

In version 1.0 of the Embedded Development Module, users are limited to selecting the default 1 kHz clock in the Configure Timed Loop dialog box. You now can expose additional timing sources directly in the Configure Timed Loop dialog box, which means users do not have to use the Timed Loop VIs to create and generate timing sources. Refer to the Using the Timed Loop VIs to Define Timing Sources section for more information about the Timed Loop VIs. Adding additional timing sources makes it easier for users to use your target because no additional programming is required to create and generate timing sources.

Note You cannot delete or modify the default 1 kHz clock.

You must define and implement your target-specific timing sources. Refer to the Defining Timing Sources section of Chapter 3, Using the Target Editor to Manage Embedded Targets, for information about the Timing Sources tab in the Target Editor, which is where you define your timing sources.

You must call the CreateInternalTSource() function for each timing source you define in the Target Editor before you call the top-level VI and after you initialize the run-time library. Define the LV_PLATFORM_RTL_INIT macro in LV_Decls_plat.h as the name of the function to call before the top-level VI is called and after the run-time library is initialized.
The external timing source is fired from the driver code by calling the
FireInternalTSource() function. You must call
FireInternalTSource() with the same index as
CreateInternalTSource() creates.

**Note** FireInternalTSource() calls the LVEventSet() function, which cannot be
called from ISRs on some targets. If you cannot call LVEventSet() from an ISR, call
FireInternalTSource() from a deferred interrupt handler.

Release all external timing sources when the embedded application finishes
executing. Define the LV_PLATFORM_RTL_FINI macro in
LVDefs_plat.h as the name of the function to call after the top-level VI
finishes executing and before the run-time library is released.

LabVIEW adds the timing sources you define this way as options in the
Configure Timed Loop dialog box.

### CreateInternalTSource()

CreateInternalTSource(TextPtr sourceName, uInt32 iIntTSource)

**Purpose**: Creates an internal timing source with name *sourceName* and
index *iIntTSource*.

*sourceName* is the name of the timing source. Each timing source name
must be unique.

*iIntTSource* is a zero-based index parameter from
<0, INT_TSOURCE_SIZE-1>. INT_TSOURCE_SIZE is a macro in
LVDefs_plat.h that defines the total number of internal timing sources.

### FireInternalTSource()

FireInternalTSource(uInt32 iCount, uInt32 iIntTSource)

**Purpose**: Generates the internal timing source that *iIntTSource* specifies.

*iCount* is the number of iterations.

*iIntTSource* is a zero-based index parameter from
<0, INT_TSOURCE_SIZE-1>. INT_TSOURCE_SIZE is a macro in
LVDefs_plat.h that defines the total number of internal timing sources.
DeleteInternalTSource()

DeleteInternalTSource(uInt32 iIntTSource)

**Purpose**: Deletes the internal timing source that `iHwTSource` specifies.

`iIntTSource` is a zero-based index parameter from `<0, INT_TSOURCE_SIZE-1>`. `INT_TSOURCE_SIZE` is a macro in `LVDefs_plat.h` that defines the total number of internal timing sources.

**Example**

The Windows Console example target implements three test timing sources. All three example timing sources are initialized and released from the `LVWinInit.c` file, which is located in the following directory:

```
labview\CCodeGen\libsrc\os\win
```

The `LV_win_init()` function is called on application start up immediately before the top-level VI executes.

**Using the Timed Loop VIs to Define Timing Sources**

Use the Timed Loop VIs to create and manipulate timing sources for a Timed Loop. The VIs are polymorphic. The instance you use depends on the memory model of the embedded application.

LabVIEW does not add the timing sources you create with the Create External Timing Source VI to the **Source Type** list in the **Configure Timed Loop** dialog box.

**Note** For static memory models, you cannot use the timing source name as a terminal in the Timed Loop because the source name is a string and static memory models do not support strings.

**Create External Timing Source VI**

Registers an external timing source you can use in a Timed Loop and in the Fire External Timing Source VI. Create an external timing source to control a Timed Loop with external events, such as Interrupts, rather than a clock timing source. You can use this polymorphic VI to register an external timing source for small memory models or static memory models.
Chapter 11  Timings

**Fire External Timing Source VI**

Generates an event for an external timing source you create with the Create External Timing Source VI. You can call the Fire External Timing Source VI whenever and as many times as you need to generate the event, such as Interrupts, that controls the rate a Timed Loop executes its subdiagram. You can use this polymorphic VI to generate an event for an external timing source for small memory models or static memory models.

**Delete External Timing Source VI**

Unregisters an external timing source, such as Interrupts, you registered with the Create External Timing Source VI. You can use this polymorphic VI to unregister an external timing source for small memory models or static memory models.
Memory Mapping

Memory mapping maps particular code segments to particular memory addresses on the processor corresponding to RAM or ROM. The LabVIEW Embedded Development Module provides a framework to define memory layout and map input sections of generated C files into output sections of built embedded applications.

How Users Use Memory Mapping

Users can right-click the build specification and select Map Memory from the shortcut menu to open the Map Memory dialog box to change the implicit mapping, if necessary. For example, if the user's application uses a large, read-only global variable that the user wants to download to ROM memory, the user can change the memory mapping for the variable.

Implementing Memory Mapping

Implementing memory mapping is optional. The LabVIEW C Code Generator prepends the _DATA_SECTION macro to all array constants. Your C compiler puts arrays into the default input section (.data) if you define _DATA_SECTION as empty, for example. You can define this macro as section (sram_section) to notify the compiler that the array goes to the sram_section input section. The definition of _DATA_SECTION is target-specific. Different C compilers provide different ways to explicitly place global variables and static variables into the different input sections. Refer to the documentation for your C compiler for information about how to explicitly place variables.

Use LEP_x_ScriptCompiler.vi to add compiler-specific implementations for your target. VIs are the smallest unit for which you can define _DATA_SECTION. It is not possible to place individual arrays of a single VI into multiple input sections. The _TEXT_SECTION macro serves the same purpose as _DATA_SECTION, but for functions. You can use this macro to place the generated C code of a VI in a non-default code section.
The memory mapping configuration persists in the LabVIEW project (.lvproj) file.

To implement memory mapping, you must implement the following VIs:

**LEP\_x\_SectionNames.vi**
Defines the name of all memory sections users can map, such as RAM, ROM, and flash memory. Use descriptive names that users understand, because users select the memory mapping from the names you implement in LEP\_x\_SectionNames.vi. This VI returns an array of strings that contain the list of section names.

**LEP\_x\_MemoryMap\_Default.vi**
Defines the default mapping of VIs to memory sections on the target. This VI returns an array of configuration clusters. Each cluster corresponds to a VI type and contains a list of code sections and data sections that may be used for that VI type. The connector pane of this VI must match EMB\_Utility\_MemMap\_VI\_Default.ctl, which is located in the following directory:

```
labview\vi.lib\LabVIEW Targets\Embedded\Utilities\MemMap
```

**LEP\_x\_MemoryMap\_Cmp.vi**
Implements the comparison operator for the configuration database. LEP\_x\_MemoryMap\_Cmp.vi returns TRUE if the VI in the path control matches the VI in the VI Config cluster. You typically do not need to change this VI. The connector pane of this VI must match EMB\_Utility\_MemMap\_VI\_Compare.ctl, which is located in the following directory:

```
labview\vi.lib\LabVIEW Targets\Embedded\Utilities\MemMap
```

**LEP\_x\_MemoryMap.vi**
Reads the memory mapping configuration data from the .lvproj file, initializes the memory mapping framework, and displays the Map Memory dialog box to the user. You must change the paths to LEP\_x\_MemoryMapDefault.vi, LEP\_x\_MemoryMap\_Cmp.vi, and the subVI calls to these two VIs.
Use EMB_Utility_MemMap_Init.vi to initialize the memory mapping framework. Call EMB_Utility_MemMap.vi to display the Map Memory dialog box to the user.

**LEP_x_MemoryMap_Query.vi**

Queries the memory mapping database for the memory section assigned to a given VI. LEP_x_ScriptCompiler.vi can call LEP_x_MemoryMap_Query.vi for each compiled source file to assign data structures and functions to explicit input memory sections. LEP_x_ScriptLinker.vi can call LEP_x_MemoryMap_Query.vi to generate or modify the linker definition file. You must change the paths to LEP_x_MemoryMapDefault.vi, LEP_x_MemoryMap_Cmp.vi, and the subVI calls to these two VIs.

Refer to LEP_vxworks_cmd565_rom_MemoryMap_Query.vi, which is located in the following directory, for an example implementation:

```
labview\Targets\NI\Embedded\vxworks\cmd565\rom\vxworks_cmd565_rom_LEP_TargetPlugin\
```

Refer to LEP_vxworks_cmd565_rom_ScriptCompiler.vi, which is located in the following directory, for an example usage:

```
labview\Targets\NI\Embedded\vxworks\cmd565\rom\vxworks_cmd565_rom_LEP_TargetPlugin\
```

The Get Mem Config subVI on the block diagram of LEP_vxworks_cmd565_rom_ScriptCompiler.vi picks the compiler script (.bat) based on the memory mapper configuration data.
Embedded Automated Tests

The LabVIEW Embedded Development Module includes an automated test framework. Use the embedded automated tests to verify successful target implementations. You might need to implement target-specific sequence files to support the execution of tests on actual hardware. Refer to the example files listed in the Test Sequences section for examples of how to implement target-specific sequence files. The tests are located in the following directory:

labview\autotest\tests\Large semi-auto tests

Select Tools►Embedded Tools►Run All Embedded Tests to run the embedded automated tests. Most tests run on all platforms.

Each test contains separate configuration files for each target. Add configuration files for all applicable tests when you create your new target.

The automated test framework includes a default set of tests to help you verify that porting LabVIEW to your target is successful. Most of the tests are generic and do not interact with any specific hardware. National Instruments strongly recommends that you add automated tests for your device drivers. The following table lists the default set of tests.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
<th>Porting Prerequisite</th>
</tr>
</thead>
<tbody>
<tr>
<td>analysis</td>
<td>Advanced analysis</td>
<td>Analysis library</td>
</tr>
<tr>
<td>arr-*</td>
<td>Array data types and functions</td>
<td>—</td>
</tr>
<tr>
<td>basic</td>
<td>Basic data types (mostly scalar) and operations that use these types</td>
<td>—</td>
</tr>
<tr>
<td>cluster</td>
<td>Cluster data types and functions</td>
<td>—</td>
</tr>
<tr>
<td>cmplx*</td>
<td>Complex data types and functions</td>
<td>—</td>
</tr>
<tr>
<td>compare*</td>
<td>Comparison operators</td>
<td>—</td>
</tr>
<tr>
<td>const</td>
<td>Front panel control constants</td>
<td>—</td>
</tr>
<tr>
<td>elementalI/O</td>
<td>Elemental I/O framework</td>
<td>Elemental I/O</td>
</tr>
</tbody>
</table>
Configuration Files

LEP_AutoTest.ini is the main automated test framework configuration file. This file lists the targets to test and notification email addresses for failed tests. Refer to the following directory for an example LEP_AutoTest.ini file:

labview\autotest

Test Sequences

LEP_AutoTest_TestSequence_1.vi

This is the default test sequence. It expects test results in a text file. This sequence is for all targets that can redirect standard output to a file. Refer to LEP_win32con_Run.vi for more details about how to redirect standard output to a test text file. The Windows Console and VxWorks Simulation example targets use this sequence. A win32con_Target.ini example test file is located in the following directory:

labview\autotest\tests\Large semi-auto tests\arr-driv\
**LEP_AutoTest_TestSequence_2.vi**

Supports target hardware reset and serial port capture capability. This sequence expects that the **Run** button can download and execute an embedded application, and that the standard output is redirected on a serial port. An `ecos_cmd565_Target.ini` example test file is located in the following directory:

```
labview\autotest\tests\Large semi-auto tests\arr-drv\
```

**Example**

The `ecos_cmd565_Target.ini` test file is located in the following directory:

```
labview\autotest\tests\Large semi-auto tests\arr-drv
```

```
[arr_drv_Test_Debug]
_Test_TopLevelVI_=arr-drv-con.vi
_Test_SequenceVI_=LEP_AutoTest_TestSequence_2.vi
i_Test_BaudRate_=57600
s_Test_ResourceName_=COM6
_Test_Timeout_=300000
iBuildConfiguration=1
bGenerateDebugInfo=FALSE
iMemoryModel=0
bGenerateSerialOnly=FALSE
bGenerateGuardCode=TRUE
bUseStackVariables=FALSE
```

The following table lists the INI tokens.

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[arr_drv_Test_Debug]</td>
<td>The name of the test. Test names must be enclosed in brackets and cannot contain spaces.</td>
</tr>
<tr>
<td><em>Test_TopLevelVI</em></td>
<td>The name of the top-level embedded VI to build into an embedded application.</td>
</tr>
<tr>
<td><em>Test_SequenceVI</em></td>
<td>The name of the test sequence VI. The default value is <code>LEP_AutoTest_TestSequence_1.vi</code>.</td>
</tr>
<tr>
<td>i_Test_BaudRate_</td>
<td>The baud rate of the capturing serial port. This key is used only by test sequences that capture output on the serial port.</td>
</tr>
<tr>
<td>Key</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td>s_Test_ResourceName_</td>
<td>The name of the capturing serial port. This key is used only by test sequences that capture output on the serial port.</td>
</tr>
<tr>
<td><em>Test_Timeout</em></td>
<td>The time interval in milliseconds to get the result before the test fails.</td>
</tr>
<tr>
<td>iBuildConfiguration bGenerateDebugInfo iMemoryModel bGenerateSerialOnly bGenerateGuardCode bUseStackVariables</td>
<td>Target-specific configuration tokens. Each target-specific configuration token is prepended with a letter that indicates the data type of the token. b = Boolean value i = integer value p = path value s = string value</td>
</tr>
</tbody>
</table>
Static Memory Allocation

The LabVIEW Embedded Development Module supports building embedded applications that do not allocate memory when running. Embedded applications that you build using a static memory model might run faster and are more deterministic than applications using a dynamic memory model because static memory models do not have to make malloc() and free() calls.

If a user builds an embedded application using the static memory model, the generated code and the LabVIEW embedded run-time code do not allocate dynamic memory during execution. Instead, the LabVIEW C Code Generator analyzes the block diagram to determine the maximum amount of memory the application requires and declares global variables to pre-allocate the required memory chunks.

However, it is not possible to determine the maximum amount of memory required for a block diagram in all cases. For example, the LabVIEW C Code Generator cannot estimate the maximum amount of memory required if a block diagram contains either of the following:

- Variable size array constants.
- Building arrays using the indexing tunnel of a loop with an unknown number of iterations.

If the LabVIEW C Code Generator cannot estimate the amount of memory required when static memory allocation is requested, the block diagram breaks and an error message indicates the problem.

Supporting a Static Memory Model

To support the static memory model, all of the C files must be compiled with the CStatic compiler directive. You must set the Memory model parameter of the LabVIEW C Code Generator to 1.

The CStatic compiler directive must be defined for all source files in the embedded project and all run-time library files. When CStatic is defined, dynamic memory management calls, such as malloc, realloc, and free, are defined to be no-ops. Additional function arguments are produced that
Chapter 14  Static Memory Allocation

are pointers to global variables the LabVIEW C Code Generator declares. The functions then use the global data in these extra function arguments instead of calling malloc. This is done with macros.

If you want to have both static and dynamic memory allocation on the same target hardware, you must compile both static memory and dynamic memory versions of the pre-built run-time libraries and link with the correct one when a user builds an embedded VI into an embedded application. An embedded application must be either completely static or completely dynamic. A single LabVIEW target cannot support both memory models. You must create two separate targets—one for static memory and one for dynamic memory. You must create a separate TargetSyntax.xml file for the static memory and dynamic memory target. Use the TargetSyntax.xml files located in the following directory as a template for dynamic memory targets:

```
labview\Targets\NI\Embedded\common
```

Use the TargetSyntax.xml files located in the following directory as a template for static memory targets:

```
labview\Targets\NI\Embedded\common\static
```

The static memory version of TargetSyntax.xml restricts the target syntax to support only fixed size arrays. Variable size arrays and strings are not supported. All scalar data types are supported without any limitations. You must enable the preallocate user interface for static memory targets. Enable the preallocate user interface by placing a checkmark in the Preallocate UI Enable (Static Memory Allocation) checkbox in the Target Editor, which enables a Set Dimension Size shortcut menu option for block diagram array constants. Refer to Chapter 3, Using the Target Editor to Manage Embedded Targets, for more information about the Target Editor.

**Limitations of the Static Memory Model**

**Limitations for End-Users**

- The String data type is not supported.
- Only fixed size array constants are supported. To specify the type and size of an array constant, right-click the array constant and select Set Dimension Size from the shortcut menu.
- You cannot access the String element of the error cluster.
The Insert Into Array function is not supported.

You can index tunnels in a For Loop, but you must define the number of iterations. The target syntax checker does not support constant folding, so you must wire a constant to the count (N) terminal of the For Loop.

You cannot wire the name of the Timed Loop external source to the Source input because the string data type is unsupported. Enter the name of the external timing source in the Source name text box in the Loop Configuration dialog box. Open the Loop Configuration dialog box by right-clicking the Timed Loop on the block diagram and selecting Configure Timed Loop from the shortcut menu. Remove any checkmarks from the Use terminal checkbox. Enter 1 in the Period box.

The number of external timing sources is limited to an OEM-defined limit.

You cannot wire string constants to the Create External Source VI, Fire External Source VI, or Delete External Source VI. You must use a fixed size array constant to specify the name of the external timing source. For example, if you want to create an external timing source named foo, you must create a fixed size array constant of 3 with values 102, 111, and 111. Verify the representation is 8-byte unsigned integer (U8) or 8-byte signed integer (I8).

The RTFIFOCreaté VI uses a string input to specify the name of the RTFIFO instance. You specify the name using a fixed size array of bytes, which is the same way you specify the external timing source names. Refer to the previous bullet for more information.

The static version of the RT-FIFO does not support arrays.

The number of RT-FIFO instances is limited to an OEM-defined limit.

The number of elements in an RT-FIFO is limited to an OEM-defined limit.

The number of Generate Occurrence functions and Wait on Occurrence functions are limited to an OEM-defined limit.

The maximum number of interrupt handles you can use is limited to an OEM-defined limit.

TCP/UDP functions are not supported.

The LabVIEW Analysis Library is not supported.
Limitations for OEMs

- The NODE_MAX macro limits the number of external timing sources.
- The FIFO_MAX macro defines the number of RT FIFO instances you can use and does not support array elements. The FIFO_SIZE_MAX macro defines the maximum number of RT FIFO elements.
- The GENERATE_OCCURRENCE_MAX macro and the WAIT_ON_OCCURRENCE_MAX macro limits the number of Generate Occurrence functions and Wait on Occurrence functions, respectively.
- The ISR_HANDLE_MAX macro defines the number of interrupt handles you can use.
Instrumented Debugging

You can use instrumented debugging with LabVIEW to communicate with an embedded application running on an embedded target, and provide front panel connectivity, probes, and block diagram breakpoints.

Instrumented debugging occurs through synchronization and data-transfer routines in the generated C code of an embedded VI. These routines use an underlying communication layer, which you must provide for your embedded targets and LabVIEW. You use plug-in VIs to implement the instrumented debugging communication layer for LabVIEW. You must implement instrumented debugging on the target with C functions. The following sections describe how to implement the host side and target side debugging plug-in VIs.

Implementing the Instrumented Debugging Communication Layer for LabVIEW

How you implement instrumented debugging depends on if you are using standard serial or TCP communication or another communication layer, such as CAN or RTDX. If you are using another communication layer, you must provide four plug-in VIs for an embedded target: open, close, read, and write. You must modify the debugging plug-in VIs to implement support for initializing the debugging connection. The debugging plug-in VIs only implement a communication layer. No parsing or interpretation of the data is necessary because LabVIEW interprets the debugging data on the host computer and the run-time code interprets the debugging data on the embedded target.

LEP_x_Debug.vi

Depending on the communication layer, LEP_x_Debug.vi must call one of the following VIs:

- nitargetStartSerialDebug.vi—Use with serial communication.
- nitargetStartTCPDebug.vi—Use with TCP communication.
• nitarGetStartDebug.vi — Use with other types of communication, such as CAN or RTDX. If you are not using serial or TCP communication, you must implement the four host plug-in VIs described in the Implementing the Host Plug-In VIs section.

These VIs are located in the following directory:

labview\vi.lib\LabVIEW Targets\TargetShared\

Implementing the Host Plug-In VIs

To implement the host side for instrumented debugging, you must create an instance of the following four VIs:

• Open VI
• Close VI
• Read VI
• Write VI

Refer to the Spectrum Digital DSK6713 example target as an implementation example. The implementation of these VIs are located in the following directory:

labview\Targets\NI\Embedded\dsp_bios\DSK6713\TMS_TargetPlugin\debug_plugin

Place these VIs in your target directory, which is the following directory.

labview\LabVIEW Targets\x

where x is your top-level target directory.

Note The labels of the controls and indicators must match your implementation exactly.

Open VI

Establishes a connection with the embedded target.

The Open VI must have the following items on the connector pane:

• error out—An error cluster indicator.
• connection data—A variant indicator. Place data for the debugging connection in this variant. This variant is passed in to all of the other debugging plug-in VIs. You can modify this variant so you can communicate between plug-in VIs you call at different times.
Close VI

Closes the connection with the embedded target.

The Close VI must have the following items on the connector pane:

- **error out**—An error cluster indicator
- **connection data**—A variant control. The contents of this variant are user-defined, and can be modified so you can communicate between plug-in VIs you call at different times.

Read VI

Reads data from the embedded target.

The Read VI must have the following items on the connector pane:

- **error out**—An error cluster indicator.
- **connection data in**—A variant control. The contents of this variant come from the last invocation of an instrumented debugger plug-in VI.
- **bytes to read**—An I32 numeric control that is the number of bytes read from the embedded target.
- **connection ID**—An I32 numeric control.

After the read has finished, your Read VI must call nitargetReadData.vi, which is located in the following directory:

```markdown
labview\vi.lib\LabVIEW Targets\TargetShared
```

nitargetReadData.vi sends the data that you read to LabVIEW. Wire the data read to the **Data Read** control on the connector pane of nitargetReadData.vi. Wire the **connection ID** passed into the Read VI to the **Connection ID** input on the nitargetReadData.vi connector pane.

Write VI

Writes data from LabVIEW to the embedded target. The write must be blocking—when the VI finishes, LabVIEW expects the data to be sent to the embedded application.
Write VI must have the following items on the connector pane:

- **error out** — An error cluster indicator
- **connection data in** — A variant control. The contents of this variant come from the last invocation of an instrumented debugger plug-in VI.
- **data to send** — A string control. The contents of this string are sent to the embedded target.

### Implementing the Instrumented Debugging Communication Layer for Your Embedded Target

To implement the target side of instrumented debugging, you must create an instance of the following C functions in the LabVIEW Embedded run-time library:

- Connect
- Disconnect
- Write
- Read
- Bytes Available

Also, you must assign global function pointers to the functions you create so that the LabVIEW Embedded run-time library can use the functions for debugger connections.

**Tip** Use new, unique names for your functions to avoid linker errors.

The Embedded Development Module provides examples of serial and TCP/IP implementations of target side communication.

The serial implementation is located in the following file:

```text
labview\CCodeGen\libsrc\comms\SerialMessaging.c
```

The TPC/IP implementation is located in the following file:

```text
labview\CCodeGen\libsrc\comms\TCPMessaging.c
```

To use these C functions, you must still implement the Connect, Disconnect, Write, Read, and Bytes Available functions, but the implementation of these functions can call the TCP and serial implementations.
Note  If a Linux target and host are different machines, you must build the run-time
libraries with TCP_USE_LOCALHOST undefined and pass the IP address of the debugger as
a command line argument to the embedded application. If a Linux target and host are the
same machine, define TCP_USE_LOCALHOST as 1.

Connect

eRunStatus DebugConnect(void)

Opens a connection to LabVIEW and establishes a connection to the Open
debugging plug-in VI. Return eFail if there is an error. Otherwise, return
eFinished.

Disconnect

eRunStatus DebugDisconnect(void)

Closes the connection to LabVIEW. Return eFail if there is an error.
Otherwise, return eFinished.

Write

eRunStatus DebugWrite(const char* pData, uInt32 size)

Writes the data that pData points to. size indicates the number of bytes to
write. Do not delete pData. Return eFail if there is an error. Otherwise,
return eFinished.

Read

eRunStatus DebugRead(char** pData, uInt32 reqBytes,
uInt32* actualBytes)

Reads the number of bytes reqBytes indicates. Returns the number of
actual bytes read in *actualBytes. *actualBytes should always equal
reqBytes when complete. Allocate a buffer in *pData to return the data.
Return eFail if there is an error. Otherwise, return eFinished.

Bytes Available

eRunStatus DebugBytes(uInt32* bytes)

Returns the number of bytes that *bytes can read. Store 0 in *bytes if no
bytes are available. Return eFail if there is an error. Otherwise, return
eFinished.
Using Instrumented Debugging

To build an embedded VI into an embedded application that uses instrumented debugging, you must give the appropriate arguments to the LabVIEW C Code Generator. Refer to the *Defining Code Generation Options* chapter for more information.

Assigning to Function Pointers

Assign the target side instrumented debugging functions to the global LabVIEW Embedded run-time function pointers in the `DebugComm.c` run-time source file in the `PDADebugInitializeComm()` function. `DebugComm.c` is located in the following directory:

```
labview\CCodeGen\libsrc\comms
```

Add a new C preprocessor identifier that describes the debugger connection mechanism. Pre-existing identifiers include `UsesTCPDebugger`, `UsesSerialDebugger`, and `UsesCANDebugger`. When you script the compiler to build an embedded application, verify you are defining the new identifier if you are building a debuggable application. Make a new `#elif` case in `PDADebugInitializeComm()`, and assign the follow function pointers:

- `PDADebugConnect`: Assign this pointer to the connection function.
- `PDADebugDisconnect`: Assign this pointer to the disconnect function.
- `PDADebugRead`: Assign this pointer to the read function.
- `PDADebugWrite`: Assign this pointer to the write function.
- `PDADebugBytes`: Assign this pointer to the bytes available function.
- `PDADebugWaitOnMessage`: Assign this pointer to `PDADebugGenericWaitOnMessage`.
- `PDADebugGetMsg`: Assign this pointer to `PDADebugGenericGetMsg`.
- `PDADebugSendMsg`: Assign this pointer to `PDADebugGenericSendMsg`. 
Troubleshooting

You can configure LabVIEW to create an embedded debugger diagnostic file. To create this diagnostic file, add the line LogPDAMessages=True to the LabVIEW.ini file, located in the labview directory. LabVIEW creates a file called labview\pdamsglog.txt, which contains various debugging diagnostic and error messages that occur as a result of instrumented debugging connections. Use this file to help you implement and troubleshoot the connection implementation.
You can implement on chip debugging (OCDI) that is compatible with the STAB (Symbol Table) as shown in Figure 16-1.

Figure 16-1. On Chip Debugging of an Embedded Application
This chapter uses simple\_math.vi, shown in Figure 16-2, as an example throughout this chapter.

![simple\_math.vi](image)

**Figure 16-2.** simple\_math.vi

You can use a JTAG, BDM, or NEXUS connection to perform on chip debugging of an embedded application.

Users can enable on chip debugging in the **Build Specification Properties** dialog box by selecting **On Chip** or **On Chip (Release)** from the **Debug mode** drop-down list. Some debugging interfaces might need to stop the CPU for a significant amount of time to access target memory.

**Note**  Some example targets use non-instrumented instead of On Chip.

The On Chip debugging mode does not affect the execution of the built embedded application because this mode does not modify the generated C code.

The On Chip (Release) debugging mode affects the execution of the embedded application because this mode adds extra checkpoint code for all possible LabVIEW breakpoints. Use the On Chip (Release) debugging mode if your C compiler cannot generate lists files to map lines of generated C code to the relative symbol offsets, if you need to debug an optimized application, or if your emulator has limited breakpoints.

**Note**  Not all targets support on chip debugging. On chip debugging might require special debugging hardware, such as JTAG, BDM, or a Nexus emulator.

### .lvm File

The .lvm file is an ASCII text file the debug database generates. Refer to the **Debug Database** section for more information about the debug database, which contains a map of all possible signals, breakpoints, controls, and indicators in the physical memory of the target device. The debug database also contains line numbers and information for each node.
on the block diagram, which LabVIEW uses to map the current debugging position in a .c file to a node on the block diagram. LabVIEW assigns each object an object ID. The .lvm file allows each object ID to be resolved to a physical memory address on the target. Also, you can map physical memory addresses back to the associated LabVIEW object IDs for real-time interaction between LabVIEW and the embedded application.

When a user selects On Chip debugging in the **Build Specification Properties** dialog box, the LabVIEW C Code Generator adds a set of comments to the generated C code to note where potential probes, breakpoints, controls, and indicators might be placed. For example, the extra comments generated for simple_math.vi is as follows:

```c
hA8EA7A0->sCA8/* n: MultiplyConstant */ = 5.0000000000000000000E+2;
hA8EA7A0->sA9C/* n: DivideConstant */ = 1.0000000000000000000E+2;
hA8EA7A0->sBEC/* n: NumberIn */ = NumberIn__1364;
OCDI_BEGIN_NODE(profileInfo_simple_math, 0, 2) /*
OCDI::Begin::Node(1, 2, 3) */
OCDI::Begin::Node(1, 2, 3) */
 /**
 */ Multiply */
 /**
 OCDI_CHECK_POINT(simple_math_OCCDIFlag[0] & 0xl,
 &simple_math_OCDIFlag, 0)/* OCDI::BreakPoint::Node(1, 2, 3) */
hA8EA7A0->sB58/* n: Multiply: x*y */ = hA8EA7A0->sBEC/* n: NumberIn */
 hA8EA7A0->sCA8/* n: MultiplyConstant */;
OCDI_END_NODE(profileInfo_simple_math, 0, 2) /* OCDI::End::Node(1, 2,
3) */
OCDI::Begin::Node(profileInfo_simple_math, 1, 3) /*
OCDI::Begin::Node(profileInfo_simple_math, 1, 3) */
OCDI::Begin::Node(2, 3, 3) */
/**
 */ Divide */
 /**
 OCDI_CHECK_POINT(simple_math_OCCDIFlag[0] & 0x2,
 &simple_math_OCDIFlag, 1)/* OCDI::BreakPoint::Node(2, 3, 3) */
hA8EA7A0->sA08/* n: Divide: x/y */ = (float64)hA8EA7A0->sB58/* n:
 Multiply: x*y */(float64)hA8EA7A0->sA9C/* n:DivideConstant */;
OCDI_END_NODE(profileInfo_simple_math, 1, 3) /* OCDI::End::Node(2, 3,
3) */
if (NumberOut__616) {*
 *NumberOut__616 =hA8EA7A0->sA08/* n: Divide: x/y */;
}
return eFinished;
```
**C Parser**

The C parser searches the generated C code for special comments and maps the comments to line numbers. This information is stored in the debug database, which translates control and signal IDs into relative symbol addresses. The database records for all breakpoints only map breakpoint IDs into line numbers. The list parser must resolve relative offsets of all breakpoints. You do not need to port the C parser because the parser is target independent. The C parser generates the bold portion of the following `.lvm` file for `simple_math.vi`:

```
[simple_math.vi]
Node(1,2,3)@92=0x4011BF
Node(2,3,3)@99=0x40122B
Location.Node(1,2,3)@[88..94]
Location.Node(2,3,3)@[95..101]
SID_2716=0x4727A0
SID_2904=0x4727A8
SID_3052=0x4727B0
SID_3240=0x4727B8
SID_2568=0x4727C0
```

For nodes, the syntax for this portion of the `.lvm` file is as follows:

```
Node(a, b, c)x=
```

where \((a, b, c)\) represents the unique LabVIEW-generated coordinates that identify the node on the block diagram, and \(x\) represents the line number where the comment occurs.

For node location information, the syntax for this portion of the `.lvm` file is as follows:

```
Location.Node(1,2,3)@[123...456]
Location.SNode(1,2,3)@[123...456]
Location.SRM(1,2,3)@[123...456]
```

where `Location.Node` represents nodes, `Location.SNode` represents structures, and `Location.SRM` represents self-referencing nodes. \([123...456]\) represents the line numbers in the generated C file.

For controls and signals, the syntax for this portion of the `.lvm` file is as follows:

```
CID_i=...
SID_i=...
```
where $\text{CID}$ represents Control ID, $\text{SID}$ represents Signal ID, and $i$ represents the unique LabVIEW-generated object ID.

**C Compiler**

A third-party target-specific compiler is called for each generated run-time library and external source file. The generated C code contains special comments that help the C parser find the symbols of all controls, signals, and positions of all possible breakpoints in the VI. The C compiler outputs an intermediate object module `.o` and a list file. The list file must contain enough information to enable the list parser to translate line numbers of the generated C files into relative symbol offsets. It is important to execute the compiler with the correct debugging and list options to generate a list file that the list parser can understand.

**Note**  
Parsing the list and map files is one way of extracting the necessary information to populate the `.lvm` file. Your toolchain might provide a different way to extract the necessary information.

**List Parser**

The list (`.lst`) parser is a compiler-specific component. The target examples use a GCC-specific parser located in the following directory:

```
labview\vi.lib\LabVIEW Targets\Embedded\Debug\Map\GCC
```

You must implement your own list parser if your compiler cannot generate the list file in a compatible stabs format. If your compiler cannot generate the list file to map lines of compiled source code into relative symbol addresses, you cannot support breakpoints in the pure On Chip debugging mode. You can still build embedded applications in the On Chip (Release) mode. The On Chip (Release) mode implements the debug checkpoint macro, so the running embedded application must evaluate simple checkpoint expressions every time the application approaches a possible LabVIEW breakpoint. Note that the debug checkpoint macro is left blank for the pure On Chip debugging mode, which eliminates the need for the list parser, but slightly changes the timing of the built application.

The list parser serves two primary purposes in populating the debug database and generating the `.lvm` file. First, the list parser helps to resolve the lines of C code to series of relative offsets based on one or two variables. This map does not contain entries for lines for which the C compiler does not generate assembly code. Second, it populates data type alignment
information that is specific to the manner in which the compiler handles the alignment of the LabVIEW data type structures.

The list parser determines which nodes and signals to locate by querying the debug database, which the C parser populates. The following example shows the .lvm file for simple_math.vi after the signals and nodes have been resolved to a series of hexadecimal offsets. The memory locations are still only known relative to yet unresolved symbols. Absolute memory address information is not available at this point in the process.

```
[simple_math.vi]
Node(1, 2, 3)@64=simple_math_Run+0x58
Node(2, 3, 3)@68=simple_math_Run+0x6C
SID_2360=simple_math_heap+0x0
SID_2212=simple_math_heap+0x8
SID_3104=simple_math_heap+0x10
SID_1988=simple_math_heap+0x18
SID_1332=simple_math_heap+0x20
```

The list parser also populates the debug database with information about the compiler byte packing. These descriptions are based on compiler-specific information, and are necessary so that the debug daemon knows what bytes correspond to the actual binary data value of the control or signal. If you are sure that the compiler alignment options are fixed and you are not providing that level of control to users, this information can be generated as a static string. The bold portion in the following example defines the Boolean data type.

```
[Offset_Table]
BigEndian=1
BooleanData=4
  BooleanData.bInput=16, 8
  BooleanData.bStatic=0, 8
  BooleanData.bVal=8, 8
  BooleanData.padding=1=24, 8
NumericData=12
  NumericData.bInput=48,8
```

LVBoolean.h defines this data type in the following directory:

```
labview\CCodeGen\include\frontpanel
```
The members of the following C data structure mirrors those listed in the .lvm file.

```c
typedef struct {
    UInt8 bStatic;     ///< 0000 000?
    UInt8 bVal;        ///< 0000 00?0
    UInt8 bInput;      ///< 0000 0?00
    UInt8 padding1;
} BooleanData;
```

The syntax for the portion of the .lvm file is as follows:

```text
DataType=L
DataType.dataMember=j, k
```

where

- \( j \) represents the offset from the beginning of the data structure in bits
- \( k \) represents the length of the data member in bits
- \( L \) represents the length of the entire data type in bytes

Repeat this process for each of the following data types:

- ArrayControlData
- BooleanData
- ClusterControlData
- EnumCtlData
- NumericData
- PictInfo
- StringData
- _ControlData
- _LVGraphData
- _OCDI_Alignment
- _PDAArr
- _PDAStr

In addition, the .lvm file needs information from the LabVIEW project to completely decouple the LabVIEW development environment and the debugging interface.

```text
BigEndian=x
ReleaseDebugMode=x
```

where \( x \) represents 1 or 0 based on the information from the LabVIEW project.
Linker

The linker links all intermediate object files, external files, and operating system libraries to produce the application image. The On Chip debugging modes require the linker to generate the map file that is used to resolve symbols of the built image. You can execute the linker with the correct command line options to generate the map file in a format that the map parser can understand. The linker can be executed with command line options to remove all debugging information and all symbols unless you want to debug an embedded application at the source level. LabVIEW does not use the debugging information for the built application. The map file does not have to be generated for targets with downloadable module support, the VxWorks downloadable module for example. All symbols must be resolved at the module load time.

.map Parser

In the first stage, the map parser runs through the map file generated by the linker and creates a structure that maps symbol names to absolute addresses of the built image. The map parser is a linker-specific component that you must implement if your linker does not produce the map file compatible with the example map parser. In the second stage, the map parser resolves all symbols in the debug database. The map parser is not executed for targets that build a downloadable module. All symbols must be resolved during the module download.

Map file parsing resolves the previously unresolved symbols and converts the relative addresses the list parser generates to physical address for the embedded target. The resolved addresses are shown in the following .lvm file:

```
[simple_math.vi]
Node(1, 2, 3)@64=0x1476C
Node(2, 3, 3)@68=0x14780
SID_2360=0xB5C50
SID_2212=0xB5C58
SID_3104=0xB5C60
SID_1988=0xB5C68
SID_1332=0xB5C70
```

When all of the symbols are resolved, the .lvm file is ready for use with LabVIEW and the debug daemon.
**Debug Database**

The debug database is a collection of records for all controls, signals, and breakpoints in the VI hierarchy for the embedded application. The debug database also contains .c line number information for nodes on the block diagram. The database is initialized before the first .c file is compiled, usually in the setup stage of the build VI. The C parser inserts records for all signals, controls, and breakpoints. The list parser maps the breakpoint line numbers into relative symbol offsets and updates all breakpoint records. Finally, the map parser resolves all symbols and updates all database records so they contain absolute addresses of the built image. The database snapshot is then saved in the .lvm file located in the project build folder. The debug daemon automatically loads the .lvm file every time you debug a built application. The debug database also contains additional index tables that improve the performance of the database so the debug daemon does not have to sequentially search object records. Index tables are not stored in the .lvm file. Instead, the debug daemon re-indexes the database every time the .lvm file is loaded.

**Debug Daemon**

The debug database implements a high-level interface LabVIEW uses to debug an application running on an embedded target. The debug daemon uses data from the debug database to translate LabVIEW object IDs into absolute addresses and absolute addresses of pending breakpoints back to LabVIEW IDs so LabVIEW can highlight the corresponding nodes. The debug daemon uses emulator-specific plug-ins to instrument actual emulator hardware. All emulator plug-ins must implement a common set of methods the debug daemon can call to set a breakpoint, clear a breakpoint, get a breakpoint, read target memory, write target memory, and so on. The debug daemon completely isolates LabVIEW and the emulator from each other. The LabVIEW debugger does not know anything about absolute object addresses and the user emulator interface. The emulator plug-in does not have to deal with internal LabVIEW object IDs.

**Emulator Plug-in**

The emulator plug-in implements a low-level emulator interface the debug daemon uses. In most cases, the emulator plug-in translates calls of the debug daemon interface into the API of the emulator vendor.
Adding Support for an Existing Emulator to a New Target

Complete the following steps to add support for an existing emulator to a new target.

1. Call `EMB_Debug_DD_Init.vi` from the Build VI before the build script executes to initialize the debug database. See `LEP_vxworks_Build.vi` for an example of how to use `EMB_Debug_DD_Init.vi`.

2. Call `EMB_Debug_Map_Gcc_UpdateDD.vi` for every compiled generated source file. `EMB_Debug_Map_Gcc_UpdateDD.vi` parses the source file and the list file the compiler generates and updates the debug database. This VI should be called only if the On Chip debugging mode is enabled in the Build Specification Properties dialog box. You must configure the compiler to generate the list file that the list file parser `EMB_Debug_Map_Gcc_UpdateDD.vi` calls can understand. Refer to the following example batch file, which executes a GCC compiler to compile the C file and generate the list file in stabs format.

```
labview\Targets\NI\Embedded\vxworks\cmd565\utils\compiled.bat
```

The following compiler options instruct the compiler to generate the list file in stabs format and change the maximum length of lines to 4096 characters to prevent truncation.

```
-gstabs -Wa
--listing-rhs-width=4096
-als=.\prjStabs.lst
```

All example targets use a GCC parser located in the following directory:

```
labview\vi.lib\LabVIEW Targets\Embedded\Debug\Map\GCC
```

You must implement your new parser for a non-GCC compiler. Refer to `LEP_vxworks_cmd565_ScriptCompiler.vi` for information about retrieving configuration data, defining the release debug macro, and calling the update debug database VI to parse source and list files.

3. Call `EMB_Debug_Map_Gcc_ResolveSymbols.vi` and `EMB_Debug_Map_Gcc_ResolveExtraSymbols.vi` after the build script executes to parse the map file and resolve all symbols in the debug database. Refer to `LEP_vxworks_cmd565_ram_ScriptLinker.vi` for information about how to set the release debug only attributes, call the resolve...
symbols, resolve extra VIs to parse the map file the linker generates, and resolve all symbols in the debug database.

4. Call EMB_Debug_DD_Save.vi after the build script executes to save the contents of the debug database to the .lvm file. You must save the .lvm file in the project directory. The .lvm file must have the same name as the project.

Adding the List Parser for a Non-GCC Compiler

You must implement the list parser if your target uses a compiler that is not compliant with GCC. The purpose of the list parser is to map lines of special debugging comments emitted into the generated code to relative offsets.

You must implement the following VIs:

**EMB_Debug_Map_x_UpdateDD.vi**

Calls the C file parser to initialize the debug database with all breakpoints, signal, and control IDs. The object addresses are left blank at this stage. Then the list parser is called to parse the list file generated by the compiler to assign relative symbol offsets to all objects listed in the debug database.

**EMB_Debug_Map_x_ResolveSymbols.vi**

 Parses the file the linker generates and resolves all objects in the debug database. You must resolve the following two types of relative addresses:

- Relative addresses in the form of variable + offset. For example, __ocdi1_heap + 0x345.
- Signals in the form of variable.member (type_code). You must find the offset of the member relative to the beginning of the variable and transform the offsets into a relative address of variable + member_offset. For example, __ocdi1_heap.c_error_in__no_error__((33,3,7)).

Resolving variable.member (type_code) Signals

The LabVIEW C Code Generator adds type information to the OCDI comments for every symbol in the signal table, which you can use to find the offset of the structure members if your toolchain does not provide a way to find the offset. Depending on the data type, the type information in the OCDI comments is in the form of (type_code), where type_code is a
numeric value for non-cluster data types or is in the form of

\((\text{type} \_\text{code1}, \text{type} \_\text{code2}, \text{and so on})\), where \text{type} \_\text{code1},
\text{type} \_\text{code2} is a numeric value for cluster data types as shown in the
following code sample.

```c
struct _ocdi1_heap {
    cl_00000 c_error_in__no_error_; /* c: error in (no error) */
    int32 l_Constant; /* l: Constant */
    int32 l_For_Loop_N; /* l: For Loop: N */
    int32 l_sum; /* l: sum */
    Boolean c_error_in__no_error__CS; /* c: error in (no error): CS */
} __DATA_SECTION __ocdi1_heap; /* hB3A9B04 */
```

```c
struct _ocdi1_heap __DATA_SECTION *hB3A9B04 = &__ocdi1_heap; /* hB3A9B04 */
/* OCDI::Memory::Heap = __ocdi1_heap */
/* OCDI::Memory::Signal::0x20FC = __ocdi1_heap.c_error_in__no_error_((33,3,7)) */
/* OCDI::Memory::Signal::0x77C = __ocdi1_heap.l_Constant (3) */
/* OCDI::Memory::Signal::0x460 = __ocdi1_heap.l_For_Loop_N (3) */
/* OCDI::Memory::Signal::0x1218 = __ocdi1_heap.l_sum (3) */
/* OCDI::Memory::Signal::0x14CC = __ocdi1_heap.c_error_in__no_error__CS (33) */
```

The C parser uses the extra comments to generate the symbols for the
signals in the form of \text{variable}.\text{member} \ (\text{type} \_\text{code}). For example,
\_\_ocdi1_heap\.l_Constant(3).

Use the following VIs to compute the member offset from signal type
information and target-specific alignment:

- Emb_Debug_Map_SignalOffset.vi
- Emb_Debug_Map_RelativeAddress.vi

These VIs are located in the following directory:

```
labview\vi.lib\LabVIEW Targets\Embedded\Debug\Map\GCC\Emb_Debug_Map_SignalOffset.vi
```
**EMB_Debug_Map_SignalOffset.vi**

Calculates the offset of a member in a structure based on the member type information and target alignment rules.

<table>
<thead>
<tr>
<th>initial offset</th>
<th>symbol</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*initial offset* is the ending offset of the previous member of the structure. *initial offset* is 0 for the first member in the structure.

*symbol* is the symbol name in the form of `variable.member(type_code)`.

*offset* is the offset of the member in the structure.

*offset + length* is the ending offset of the member in the structure.

**EMB_Debug_Map_RelativeAddress.vi**

Calculates the relative address of a symbol in the format `variable + offset` or `variable.member(type_code)`. This VI uses `EMB_Debug_Map_SignalOffset.vi` internally.

<table>
<thead>
<tr>
<th>relative address</th>
<th>base variable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*relative address* is the relative address to process.

*last base variable* is the last base variable this VI generates.

*last offset* is the last offset this VI generates.

*base variable* is the base variable of the relative address.

*offset* is the offset of the relative address.

*offset+size* is the end offset of the symbol.
Refer to the following VI for an implementation example:

```
labview\vi.lib\LabVIEW Targets\Embedded\Debug\Map\GCC\EMB_Debug_Map_Gcc.ResolveSymbols.vi
```

**Note** If your toolchain provides an easier way to obtain the offset of structure members, use your toolchain and disregard the type information in the OCDI comments.

## Adding Support for a New Emulator Interface

Complete the following steps to add support for a new emulator interface:

1. Navigate to the following directory and create a new subdirectory.

```
labview\vi.lib\LabVIEW Targets\Embedded\Debug\OCDI
```

2. Copy the contents of the `EST` directory, which contains implementation of the plug-in VIs for all WindRiver emulators.

3. Rename all of the VIs in the emulator support directory to match the name of folder. Do not rename `EMB_Debug_OCDI.ini`.

4. Edit `EMB_Debug_OCDI.ini` so all tokens point to the VI you renamed in step 3.

5. Change the implementation of all VIs in the root directory. Implementation of some methods is optional.

## OCDI Debugging Plug-In VIs

You must implement the following OCDI debugging plug-in VIs:

### IInit

Establishes the debug connection to the hardware and returns the breakpoint occurrence by initializing a low-level emulator driver, connecting to the emulator, and returning a reference to the LabVIEW occurrence that is signaled by the low-level driver every time the emulator encounters a breakpoint event on the target. You can spawn a background VI that polls a low-level emulator API for events and sets breakpoint occurrences as needed. How you implement this VI depends on the behavior of your emulator. You can use the **build ID** input to prevent users from downloading and debugging an embedded application from two different build specifications.
Note Call the `Occur()` function to signal an occurrence from the driver DLL. You must include the driver DLL in the `extcode.h` header file, which is located in the `labview\cintools` directory, and link with `labview.lib`, which also is located in the `labview\cintools` directory.

**build ID** is the ID of the build specification project item from which this VI is called.

**target ID** is the ID of the target project item from which this VI is called. Use **target ID** to distinguish between concurrent debug connections to different targets over the same plug-in interface.

**error in** describes error conditions that occur before this VI or function runs. The default is **no error**. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This VI or function runs normally only if no error occurred before this VI or function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error Handler VI to display the description of the error code. Use **error in** and **error out** to check errors and to specify execution order by wiring **error out** from one node to **error in** of the next node.

**status** is **TRUE (X)** if an error occurred before this VI or function ran or **FALSE (checkmark)** to indicate a warning or that no error occurred before this VI or function ran. The default is **FALSE**.

**code** is the error or warning code. The default is 0. If status is **TRUE**, code is a nonzero error code. If status is **FALSE**, code is 0 or a warning code.

**source** specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.

**occurrence** indicates a breakpoint occurrence, which is signaled by the low-level emulator driver when a hardware or software breakpoint is encountered. You can set an occurrence from a driver VI or DLL.
error out contains error information. If error in indicates that an error occurred before this VI or function ran, error out contains the same error information. Otherwise, it describes the error status that this VI or function produces.

status is TRUE (X) if an error occurred or FALSE.

code is the error or warning code. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.

IRelease

Closes all background VIs and the connection to the emulator API. How you implement this VI depends on the behavior of your emulator. The LabVIEW download, run, and debug commands call the IInit and IRelease plug-in VIs before and after each command. You must ensure that the emulator does not reset the target every time the IInit plug-in VI is called, which would invalidate the target memory before the debug or run commands can be called. National Instruments recommends connecting to the emulator only when the IInit plug-in VI is called for the first time or when the emulator is in a state that requires initialization. Do not disconnect from the emulator when the IRelease plug-in VI is called. The drawback of this recommendation is that LabVIEW keeps an open and active connection to the emulator. You must then release the debug connection and all associated resources from the IExit plug-in VI that is called when a target item is deleted from the project or the project is closed.

target ID is the ID of the target project item from which this VI is called. Use target ID to distinguish between concurrent debug connections to different targets over the same plug-in interface.

error in describes error conditions that occur before this VI or function runs. The default is no error. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This
VI or function runs normally only if no error occurred before this VI or function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error Handler VI to display the description of the error code. Use error in and error out to check errors and to specify execution order by wiring error out from one node to error in of the next node.

status is TRUE (X) if an error occurred before this VI or function ran or FALSE (checkmark) to indicate a warning or that no error occurred before this VI or function ran. The default is FALSE.

code is the error or warning code. The default is 0. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.

disconnection is TRUE if a users clicks the Abort Execution button when debugging a VI.

error out contains error information. If error in indicates that an error occurred before this VI or function ran, error out contains the same error information. Otherwise, it describes the error status that this VI or function produces.

status is TRUE (X) if an error occurred or FALSE.

code is the error or warning code. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.

ISetBreakpoints

Sets one or more hardware or software breakpoints on specified absolute addresses.

Note  You must access the emulator interface in a loop for each address in address list if the emulator API does not expose how to set more than one breakpoint at a time.
target ID is the ID of the target project item from which this VI is called. Use target ID to distinguish between concurrent debug connections to different targets over the same plug-in interface.

address list is an array of absolute breakpoint addresses.

error in describes error conditions that occur before this VI or function runs. The default is no error. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This VI or function runs normally only if no error occurred before this VI or function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error Handler VI to display the description of the error code. Use error in and error out to check errors and to specify execution order by wiring error out from one node to error in of the next node.

status is TRUE (X) if an error occurred before this VI or function ran or FALSE (checkmark) to indicate a warning or that no error occurred before this VI or function ran. The default is FALSE.

code is the error or warning code. The default is 0. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.

error out contains error information. If error in indicates that an error occurred before this VI or function ran, error out contains the same error information. Otherwise, it describes the error status that this VI or function produces.

status is TRUE (X) if an error occurred or FALSE.

code is the error or warning code. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.
source describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.

IClearBreakpoints

Clears breakpoints on given absolute addresses.

You must access the emulator interface in a loop for each address in address list if the emulator API does not expose how to set more than one breakpoint at a time.

- target ID is the ID of the target project item from which this VI is called. Use target ID to distinguish between concurrent debug connections to different targets over the same plug-in interface.
- address list is an array of absolute breakpoint addresses.
- error in describes error conditions that occur before this VI or function runs. The default is no error. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This VI or function runs normally only if no error occurred before this VI or function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error Handler VI to display the description of the error code. Use error in and error out to check errors and to specify execution order by wiring error out from one node to error in of the next node.
- status is TRUE (X) if an error occurred before this VI or function ran or FALSE (checkmark) to indicate a warning or that no error occurred before this VI or function ran. The default is FALSE.
- code is the error or warning code. The default is 0. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.
- source specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.
error out contains error information. If error in indicates that an error occurred before this VI or function ran, error out contains the same error information. Otherwise, it describes the error status that this VI or function produces.

status is TRUE (X) if an error occurred or FALSE.

code is the error or warning code. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.

IGetBreakpoint

Retrieves the current breakpoint when a breakpoint occurrence is signaled by the low-level emulator driver.

target ID is the ID of the target project item from which this VI is called. Use target ID to distinguish between concurrent debug connections to different targets over the same plug-in interface.

error in describes error conditions that occur before this VI or function runs. The default is no error. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This VI or function runs normally only if no error occurred before this VI or function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error Handler VI to display the description of the error code. Use error in and error out to check errors and to specify execution order by wiring error out from one node to error in of the next node.

status is TRUE (X) if an error occurred before this VI or function ran or FALSE (checkmark) to indicate a warning or that no error occurred before this VI or function ran. The default is FALSE.
code is the error or warning code. The default is 0. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.

address is the absolute breakpoint address.

breakpoint indicates the active breakpoint. breakpoint must be TRUE only the first time a breakpoint occurs when this plug-in VI is called. breakpoint must be FALSE for all subsequent calls to this plug-in VI until the next breakpoint is encountered.

error out contains error information. If error in indicates that an error occurred before this VI or function ran, error out contains the same error information. Otherwise, it describes the error status that this VI or function produces.

status is TRUE (X) if an error occurred or FALSE.

code is the error or warning code. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.

IContExecution

Resumes execution of the target until the next breakpoint is encountered. This plug-in VI is called when an application a user is debugging stops because of a breakpoint event and the user clicks the Pause button or one of the single-stepping buttons to continue execution from the current position of the program counter.

target ID is the ID of the target project item from which this VI is called. Use target ID to distinguish between concurrent debug connections to different targets over the same plug-in interface.
**error in** describes error conditions that occur before this VI or function runs. The default is **no error**. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This VI or function runs normally only if no error occurred before this VI or function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error Handler VI to display the description of the error code. Use **error in** and **error out** to check errors and to specify execution order by wiring **error out** from one node to **error in** of the next node.

**status** is **TRUE** (X) if an error occurred before this VI or function ran or **FALSE** (checkmark) to indicate a warning or that no error occurred before this VI or function ran. The default is **FALSE**.

**code** is the error or warning code. The default is 0. If status is **TRUE**, code is a nonzero error code. If status is **FALSE**, code is 0 or a warning code.

**source** specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.

**error out** contains error information. If **error in** indicates that an error occurred before this VI or function ran, **error out** contains the same error information. Otherwise, it describes the error status that this VI or function produces.

**status** is **TRUE** (X) if an error occurred or **FALSE**.

**code** is the error or warning code. If status is **TRUE**, code is a nonzero error code. If status is **FALSE**, code is 0 or a warning code.

**source** describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.

**IResolveSymbols**

Resolves relative breakpoint, signal, and control addresses on targets that use loaders or have operating systems that support dynamic linking. The loader or OS must expose the ability to return the absolute address of the passed symbol. LabVIEW uses the information this plug-in VI returns to resolve all symbols in the debug database.
target ID is the ID of the target project item from which this VI is called. Use target ID to distinguish between concurrent debug connections to different targets over the same plug-in interface.

symbol list is an array of symbol names to resolve.

target ID is the ID of the target project item from which this VI is called. Use target ID to distinguish between concurrent debug connections to different targets over the same plug-in interface.

target ID is the ID of the target project item from which this VI is called. Use target ID to distinguish between concurrent debug connections to different targets over the same plug-in interface.
source describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.

**IMemoryRead**

Reads one or more memory blocks from the target. LabVIEW combines front panel reads and probe reads into as few IMemoryRead plug-in calls as possible without further optimizations. This plug-in VI can join passed block requests into fewer, but longer, blocks with some overhead to improve overall emulator performance. The optimization algorithm depends on the type of emulator connection.

**target ID** is the ID of the target project item from which this VI is called. Use **target ID** to distinguish between concurrent debug connections to different targets over the same plug-in interface.

**address** is an array of memory block base addresses.

**block size** is an array of memory block sizes.

**error in** describes error conditions that occur before this VI or function runs. The default is **no error**. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This VI or function runs normally only if no error occurred before this VI or function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error Handler VI to display the description of the error code. Use **error in** and **error out** to check errors and to specify execution order by wiring **error out** from one node to **error in** of the next node.

**status** is TRUE (X) if an error occurred before this VI or function ran or FALSE (checkmark) to indicate a warning or that no error occurred before this VI or function ran. The default is FALSE.

**code** is the error or warning code. The default is 0. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.
source specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.

data is the target data.

error out contains error information. If error in indicates that an error occurred before this VI or function ran, error out contains the same error information. Otherwise, it describes the error status that this VI or function produces.

status is TRUE (X) if an error occurred or FALSE.

code is the error or warning code. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.

**IMemoryWrite**

Writes memory blocks to the target when a user changes the value of a front panel control while debugging.

target ID is the ID of the target project item from which this VI is called. Use target ID to distinguish between concurrent debug connections to different targets over the same plug-in interface.

address is an array of memory block base addresses.

block size is an array of memory block sizes.

data is the target data.

error in describes error conditions that occur before this VI or function runs. The default is no error. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This VI or function runs normally only if no error occurred before this VI or function runs.
function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error Handler VI to display the description of the error code. Use error in and error out to check errors and to specify execution order by wiring error out from one node to error in of the next node.

status is TRUE (X) if an error occurred before this VI or function ran or FALSE (checkmark) to indicate a warning or that no error occurred before this VI or function ran. The default is FALSE.

code is the error or warning code. The default is 0. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.

error out contains error information. If error in indicates that an error occurred before this VI or function ran, error out contains the same error information. Otherwise, it describes the error status that this VI or function produces.

status is TRUE (X) if an error occurred or FALSE.

code is the error or warning code. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.

**IGo**

Executes the downloaded image on the target. This plug-in VI is called from the build specification shortcut menu and instruments the emulator to set the program counter initial value and continue execution.
build ID is the ID of the build specification project item from which this VI is called.

target ID is the ID of the target project item from which this VI is called. Use target ID to distinguish between concurrent debug connections to different targets over the same plug-in interface.

error in describes error conditions that occur before this VI or function runs. The default is no error. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This VI or function runs normally only if no error occurred before this VI or function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error Handler VI to display the description of the error code. Use error in and error out to check errors and to specify execution order by wiring error out from one node to error in of the next node.

status is TRUE (X) if an error occurred before this VI or function ran or FALSE (checkmark) to indicate a warning or that no error occurred before this VI or function ran. The default is FALSE.

code is the error or warning code. The default is 0. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.

error out contains error information. If error in indicates that an error occurred before this VI or function ran, error out contains the same error information. Otherwise, it describes the error status that this VI or function produces.

status is TRUE (X) if an error occurred or FALSE.

code is the error or warning code. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.
IStop

Aborts the execution of a debugged application when a user clicks the **Abort Execution** button.

**target ID** is the ID of the target project item from which this VI is called. Use **target ID** to distinguish between concurrent debug connections to different targets over the same plug-in interface.

**error in** describes error conditions that occur before this VI or function runs. The default is **no error**. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This VI or function runs normally only if no error occurred before this VI or function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error Handler VI to display the description of the error code. Use **error in** and **error out** to check errors and to specify execution order by wiring **error out** from one node to **error in** of the next node.

**status** is TRUE (X) if an error occurred before this VI or function ran or FALSE (checkmark) to indicate a warning or that no error occurred before this VI or function ran. The default is FALSE.

**code** is the error or warning code. The default is 0. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

**source** specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.

**error out** contains error information. If **error in** indicates that an error occurred before this VI or function ran, **error out** contains the same error information. Otherwise, it describes the error status that this VI or function produces.

**status** is TRUE (X) if an error occurred or FALSE.

**code** is the error or warning code. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.
source describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.

IDownload

Downloads the embedded application to target memory. This plug-in VI is called from the build specification shortcut menu to download an embedded application to the target and instruments the emulator to download the application using dedicated or just memory write functions.

Note The format of the application image is not defined.

build ID is the ID of the build specification project item from which this VI is called.

target ID is the ID of the target project item from which this VI is called. Use target ID to distinguish between concurrent debug connections to different targets over the same plug-in interface.

path is the absolute local path application image.

error in describes error conditions that occur before this VI or function runs. The default is no error. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This VI or function runs normally only if no error occurred before this VI or function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error Handler VI to display the description of the error code. Use error in and error out to check errors and to specify execution order by wiring error out from one node to error in of the next node.

status is TRUE (X) if an error occurred before this VI or function ran or FALSE (checkmark) to indicate a warning or that no error occurred before this VI or function ran. The default is FALSE.
code is the error or warning code. The default is 0. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.

error out contains error information. If error in indicates that an error occurred before this VI or function ran, error out contains the same error information. Otherwise, it describes the error status that this VI or function produces.

status is TRUE (X) if an error occurred or FALSE.

code is the error or warning code. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.

IConfig

Invokes an emulator-specific configuration dialog box. This VI is called from the build specification shortcut menu to configure the emulator.

build ID is the ID of the build specification project item from which this VI is called.

target ID is the ID of the target project item from which this VI is called. Use target ID to distinguish between concurrent debug connections to different targets over the same plug-in interface.

error in describes error conditions that occur before this VI or function runs. The default is no error. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This VI or function runs normally only if no error occurred before this VI or
function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error Handler VI to display the description of the error code. Use error in and error out to check errors and to specify execution order by wiring error out from one node to error in of the next node.

**status** is TRUE (X) if an error occurred before this VI or function ran or FALSE (checkmark) to indicate a warning or that no error occurred before this VI or function ran. The default is FALSE.

**code** is the error or warning code. The default is 0. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

**source** specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.

**error out** contains error information. If error in indicates that an error occurred before this VI or function ran, error out contains the same error information. Otherwise, it describes the error status that this VI or function produces.

**status** is TRUE (X) if an error occurred or FALSE.

**code** is the error or warning code. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

**source** describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.

**IExit**

Closes all connections and releases all resources associated with a specific target. The IRelease plug-in VI calls this plug-in VI when a user removes a target from a project or the project is closed.
**target ID** is the ID of the target project item from which this VI is called. Use **target ID** to distinguish between concurrent debug connections to different targets over the same plug-in interface.

**error in** describes error conditions that occur before this VI or function runs. The default is no error. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This VI or function runs normally only if no error occurred before this VI or function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error Handler VI to display the description of the error code. Use **error in** and **error out** to check errors and to specify execution order by wiring error out from one node to **error in** of the next node.

**status** is TRUE (X) if an error occurred before this VI or function ran or FALSE (checkmark) to indicate a warning or that no error occurred before this VI or function ran. The default is FALSE.

**code** is the error or warning code. The default is 0. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

**source** specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.

**error out** contains error information. If **error in** indicates that an error occurred before this VI or function ran, **error out** contains the same error information. Otherwise, it describes the error status that this VI or function produces.

**status** is TRUE (X) if an error occurred or FALSE.

**code** is the error or warning code. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

**source** describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.
Example

The following targets support OCDI debugging:

- Axiom CMD565, VxWorks RAM Image
- Axiom CMD565, VxWorks Module
- Axiom CMD566, VxWorks
- Unix Console

An example list file parser for GCC-compatible compilers that can generate list files in stabs format is located in the following directory:

labview\vi.lib\LabVIEW Targets\Embedded\Debug\Map

Refer to the following directory for examples of support for various emulators:

labview\vi.lib\LabVIEW Targets\Embedded\Debug\OCDI
You can integrate LabVIEW with another Integrated Development Environment (IDE) if you have fully implemented OCDI debugging for your target. Refer to Chapter 16, *On Chip Debugging*, for information about implementing OCDI debugging. You can have side-by-side debugging sessions. Traditional embedded developers can use their C experience to debug LabVIEW embedded applications by using a familiar IDE in parallel with LabVIEW. LabVIEW developers can use the side-by-side debugging sessions to understand what C code the LabVIEW C Code Generator generates for each block diagram object in an embedded VI.

IDE integration provides users with the following debugging features:

- **Synchronized stepping**—Stepping through the generated C code in an IDE updates the node in LabVIEW.
- **Synchronized breakpoints**—Setting and clearing breakpoints in an IDE updates the corresponding breakpoints in LabVIEW.
- **Application crash point detection**—When LabVIEW receives a crash message from the IDE, the problem node highlights in LabVIEW and a dialog box displays the reason.
- **Show source code**—Adds a *Show source code* shortcut menu option to each node during an OCDI debugging session. When a user selects this shortcut menu option, the corresponding C code is displayed in the IDE.

When you implement IDE integration, the LabVIEW C Code Generator adds a set of comments to the generated code to mark the beginning and the end of a node. These comments are parsed and added to the debug database.
Chapter 17 Integrating LabVIEW with Another IDE

The following code shows an example of these extra comments:

```plaintext
hA8EA7A0->sCA8/* n: MultiplyConstant */ = 5.0000000000000000000E+2;

hA8EA7A0->sA9C/* n: DivideConstant */ = 1.0000000000000000000E+2;

hA8EA7A0->sBEC/* n: NumberIn */ = NumberIn__1364;

OCDI_BEGIN_NODE(profileInfo_simple_math, 0, 2) /*
OCDI::Begin::Node(1, 2, 3) */

/* Multiply */

OCDI_CHECK_POINT(simple_math_OCDIFlag[0] & 0x1,
&simple_math_OCDIFlag, 0)/* OCDI::BreakPoint::Node(1, 2, 3) */

hA8EA7A0->sB58/* n: Multiply: x*y */ = hA8EA7A0->sBEC/* n: NumberIn */
* hA8EA7A0->sCA8/* n: MultiplyConstant */;

OCDI_END_NODE(profileInfo_simple_math, 0, 2) /* OCDI::End::Node(1, 2, 3) */

OCDI_BEGIN_NODE(profileInfo_simple_math, 1, 3) /*
OCDI::Begin::Node(2, 3, 3) */

/* Divide */

OCDI_CHECK_POINT(simple_math_OCDIFlag[0] & 0x2,
&simple_math_OCDIFlag, 1)/* OCDI::BreakPoint::Node(2, 3, 3) */

hA8EA7A0->sA08/* n: Divide: x/y */ = (float64)hA8EA7A0->sB58/* n: Multiply: x*y */;

OCDI_END_NODE(profileInfo_simple_math, 1, 3) /* OCDI::End::Node(2, 3, 3) */

if (NumberOut__616) {
    *NumberOut__616 = hA8EA7A0->sA08/* n: Divide: x/y */;
}

return eFinished;

Plug-In VIs

To implement IDE integration with LabVIEW, you must modify and implement the following OCDI debugging plug-in VIs:

- IInitIndication
- IGetIndication
- IShowSource
- IShowCrash
You can use the Eclipse implementation as a starting point, but you must reimplement the plug-in VIs for each target and IDE combination. Refer to the Example Implementation section in this chapter for more information about the LabVIEW and Eclipse IDE example integration.

Tip If you use the Eclipse example implementation as a starting point, change the icon banner for the plug-in VIs to indicate the IDE you are implementing.

**InitIndication**

Initializes the IDE indication occurrence. You can use this VI to spawn a background VI to listen for events from the IDE.

- **build ID** is the ID of the build specification project item from which this VI is called.
- **target ID** is the ID of the target project item from which this VI is called. Use target ID to distinguish between concurrent debug connections to different targets over the same plug-in interface.
- **error in** describes error conditions that occur before this VI or function runs. The default is no error. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This VI or function runs normally only if no error occurred before this VI or function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error Handler VI to display the description of the error code. Use error in and error out to check errors and to specify execution order by wiring error out from one node to error in of the next node.
- **status** is TRUE (X) if an error occurred before this VI or function ran or FALSE (checkmark) to indicate a warning or that no error occurred before this VI or function ran. The default is FALSE.
- **code** is the error or warning code. The default is 0. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.
**source** specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.

**indication occurrence** is the occurrence used for indicating IDE events.

**error out** contains error information. If **error in** indicates that an error occurred before this VI or function ran, **error out** contains the same error information. Otherwise, it describes the error status that this VI or function produces.

**status** is TRUE (X) if an error occurred or FALSE.

**code** is the error or warning code. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

**source** describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.

**IGetIndication**

Retrieves the last event from the IDE.

**target ID** is the ID of the target project item from which this VI is called. Use **target ID** to distinguish between concurrent debug connections to different targets over the same plug-in interface.

**error in** describes error conditions that occur before this VI or function runs. The default is no error. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This VI or function runs normally only if no error occurred before this VI or function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error Handler VI to display the description of the error code. Use **error in** and **error out** to check errors and to specify execution order by wiring **error out** from one node to **error in** of the next node.
**status** is TRUE (X) if an error occurred before this VI or function ran or FALSE (checkmark) to indicate a warning or that no error occurred before this VI or function ran. The default is FALSE.

**code** is the error or warning code. The default is 0. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

**source** specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.

**indication type** is the type of event.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>indStart</td>
<td>Indicates that the IDE has finished the start-up procedure.</td>
</tr>
<tr>
<td>indBreakpointHit</td>
<td>Signals a stop on a breakpoint in the application being debugged.</td>
</tr>
<tr>
<td>indBreakpointAdd</td>
<td>Generated when a user adds a breakpoint in the IDE.</td>
</tr>
<tr>
<td>indBreakpointRem</td>
<td>Generated when a user clears a breakpoint in the IDE.</td>
</tr>
<tr>
<td>indSuspend</td>
<td>Generated when the application being debugged suspends execution in the IDE, for example at a step end.</td>
</tr>
<tr>
<td>indSignal</td>
<td>Indicates that a fatal signal occurred in the IDE. This type of event usually is caused when an embedded application crashes.</td>
</tr>
<tr>
<td>indStop</td>
<td>Indicates that the IDE is about to exit. LabVIEW stops the corresponding VI when this event is received.</td>
</tr>
</tbody>
</table>

**c file name** indicates the C file where the event occurred. This output is required for the following event types:

- indBreakpointHit
- indBreakpointAdd
- indBreakpointRem
- indSuspend

**line number** is the line number at which the event occurred. This output is required for the following event types:

- indBreakpointHit
- indBreakpointAdd
- indBreakpointRem
- indSuspend
error out contains error information. If error in indicates that an error occurred before this VI or function ran, error out contains the same error information. Otherwise, it describes the error status that this VI or function produces.

status is TRUE (X) if an error occurred or FALSE.

code is the error or warning code. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.

data is a generic, IDE-specific output that contain different data. For example, it might contain the stack trace when an indSignal event is received.

IShowSource

Called when a user right-clicks a block diagram object and select Show source code from the shortcut menu.

target ID is the ID of the target project item from which this VI is called. Use target ID to distinguish between concurrent debug connections to different targets over the same plug-in interface.

line is the line number in the generated C code that corresponds to a block diagram object.

source code file is the C filename where the generated C code for the object resides.

error in describes error conditions that occur before this VI or function runs. The default is no error. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This VI or function runs normally only if no error occurred before this VI or function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error
Handler VI to display the description of the error code. Use **error in** and **error out** to check errors and to specify execution order by wiring **error out** from one node to **error in** of the next node.

**status** is TRUE (X) if an error occurred before this VI or function ran or FALSE (checkmark) to indicate a warning or that no error occurred before this VI or function ran. The default is FALSE.

**code** is the error or warning code. The default is 0. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

**source** specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.

**error out** contains error information. If **error in** indicates that an error occurred before this VI or function ran, **error out** contains the same error information. Otherwise, it describes the error status that this VI or function produces.

**status** is TRUE (X) if an error occurred or FALSE.

**code** is the error or warning code. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

**source** describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.

**IShowCrash**

Called when LabVIEW receives an indSignal event.

**target ID** is the ID of the target project item from which this VI is called. Use **target ID** to distinguish between concurrent debug connections to different targets over the same plug-in interface.
VI Name is the name of the VI that caused the crash. This string is empty if LabVIEW cannot obtain the mapping.

index is the index of the LabVIEW object that caused the crash.

data is a generic, IDE-specific output that contain different data. For example, it might contain the stack trace when an indSignal event is received.

error in describes error conditions that occur before this VI or function runs. The default is no error. If an error occurred before this VI or function runs, the VI or function passes the error in value to error out. This VI or function runs normally only if no error occurred before this VI or function runs. If an error occurs while this VI or function runs, it runs normally and sets its own error status in error out. Use the CCG Error Handler VI to display the description of the error code. Use error in and error out to check errors and to specify execution order by wiring error out from one node to error in of the next node.

status is TRUE (X) if an error occurred before this VI or function ran or FALSE (checkmark) to indicate a warning or that no error occurred before this VI or function ran. The default is FALSE.

code is the error or warning code. The default is 0. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source specifies the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning. The default is an empty string.

error out contains error information. If error in indicates that an error occurred before this VI or function ran, error out contains the same error information. Otherwise, it describes the error status that this VI or function produces.

status is TRUE (X) if an error occurred or FALSE.

code is the error or warning code. If status is TRUE, code is a nonzero error code. If status is FALSE, code is 0 or a warning code.

source describes the origin of the error or warning and is, in most cases, the name of the VI or function that produced the error or warning.
Example Implementation

The Unix Console example target implements IDE integration with Eclipse, which is an open source framework. Refer to the following directory for the Unix Console and Eclipse implementation:

labview\vi.lib\LabVIEW Targets\Embedded\Debug\OCDI\Eclipse

Installing the Necessary Tools for LabVIEW and Eclipse Integration

For LabVIEW and Eclipse integration, you must install the following tools in addition to the Embedded Development Module:

- Eclipse SDK 3.1.2
- CDT SDK 3.0.2 plug-in
- LabVIEW Embedded Eclipse Plug-In
- Cygwin and GNU tools

Eclipse


CDT SDK

After you install Eclipse, you must install the CDT (C/C++ Development Tools for Eclipse) SDK plug-in. Complete the following steps to install the CDT SDK plug-in.

1. Launch Eclipse and select Help»Software Updates»Find and Install.
2. Select Search for new features to install and click the Next button.
3. Click New Remote Site to add an update site.
4. Enter CDT for the name and http://download3.eclipse.org/tools/cdt/releases/eclipse3.1 for the URL.

Note This URL was valid at the time this document was published.

5. Select Eclipse C/C++ Development Tool SDK 3.0.2 and click the Next button.
6. Review the license agreement and click the Next button.
7. Click the Finish button.
8. Click the Install All button.
9. Click the Yes button to restart the workbench.

**LabVIEW Embedded Eclipse Plug-In**

Complete the following steps to install the LabVIEW Embedded Eclipse Plug-in.

1. Launch Eclipse and select Help»Software Updates»Find and Install.
2. Select Search for new features to install and click the Next button.
3. Click New Local Site to add an update site.
4. Navigate to and select the labview\Targets\NI\Embedded\unix\eclipse directory.
5. Select the update site you just created and click the Finish button.
6. Select LabVIEW Embedded Eclipse plug-in 1.0.x and click the Next button.
7. Click the Finish button.
8. Click the Install All button.
9. Click the Yes button to restart the workbench.

**Cygwin 1.15.18-1 or 1.15.21**

Refer to the Cygwin Web site at www.cygwin.com for download and installation instructions.

Install the following GNU tools in Cygwin:

- binutils 20050608-2
- gcc 3.3.3-3
- gdb 20041228-2
- make 3.79.1-7

**The Implementation**

TCP/IP is used to communicate between LabVIEW and Eclipse in the Unix Console example target. If the communication fails, you see a TCP Write or TCP Read error. National Instruments recommends restarting Eclipse and removing the Console_Application project LabVIEW creates in Eclipse before launching a new debugging session from LabVIEW.
Removing the Console_Application project allows Eclipse to properly recover.

**Debugging Backend**

Eclipse uses gdb as a debugging backend. Eclipse and gdb are designed for user interaction. Memory access occurs only when the application being debugged is paused by a user. Because of how Eclipse and gdb are designed, the update rate for controls, indicators, and probes is slow and might fail when you debug a running embedded application using IDE integration with Eclipse. LabVIEW must interrupt the application before reading from or writing to the target memory. This is not an issue while single-stepping or if the application is paused.

The Unix Console example target includes a **Show source code** shortcut menu item that appears on each block diagram object if you are in an active debugging session. Select **Show source code** to see the C code the LabVIEW C Code Generator generates for that object. The IShowSource plug-in VI implements this functionality.

**Target Properties Dialog Box**

The Unix Console example target **Target Properties** dialog box includes Eclipse-specific IDE integration settings. The **General Properties** page implementation for the Unix Console example target is located in the following file:

```plaintext
labview\Targets\NI\Embedded\unix\unix_LEP_TargetPlugin\LEP_unix_TargetConfigDialog.vi
```

Right-click the Unix Console example target in the **Project Explorer** window and select **Properties** from the shortcut menu to open the **Target Properties** dialog box. You can use `LEP_unix_TargetConfigDialog.vi` as a starting point to implement your own version. If you use the Unix Console implementation as a starting point, change the context help for each control in the **VI Properties** dialog box.
18

Configuring the Target Syntax

You can customize the LabVIEW editing environment so only features a target supports are available. Refer to the *LabVIEW Embedded Development Module Target Distribution Guide*, available by selecting **Start»All Programs»National Instruments»LabVIEW»LabVIEW Manuals** and opening **EMB_Distribution_Guide.pdf**, for more information about customizing the editing environment, which you must do if you plan on reselling your target.

Target syntax determines the supported and unsupported features for a specific target. Configuring the target syntax limits features users can use when developing an embedded application to the features the target can support. It also prevents you or other users from importing VIs from other targets that are unsupported for another target.

Select **Tools»Embedded Tools»Target Syntax Editor** to open the **Configure Target Syntax** dialog box. Use the **Configure Target Syntax** dialog box to create the XML file LabVIEW uses to check the target syntax.

**Note** This process overwrites **TargetSyntax.xml**, which LabVIEW uses to determine target syntax. National Instruments recommends that you back up **TargetSyntax.xml** before modifying it.

Adding Target Syntax to an Embedded Target

Complete the following steps to configure the target syntax for a new target.

1. Copy **labview\examples\TargetSyntax** to your target directory. This directory contains the VIs that determine the target syntax. Features that the LabVIEW C Generator does not support are already included.
   a. **LVEDefaultNumericType.ctl** determines the default data type of front panel controls. For example, if you are creating an integer-only target and want the default data type for numeric controls to be int32, create a custom int32 control.
b. LVEUnsupportedNodes.vi determines which nodes the target does not support.
c. LVEUnsupportedFunctions.vi determines which functions the target does not support.
d. LVEUnsupportedTypes.vi determines which data types the target does not support.
e. LVEUnsupportedVIs.vi determines which VIs the target does not support.

2. Edit the VIs described in step 1 by placing unsupported features on the block diagram of the appropriate VI. Save the VI.

Note For targets with very limited functionality, you can create a VI that contains only the features that the target can support.

3. Select Tools»Embedded Tools»Target Syntax Editor to open the Configure Target Syntax dialog box.

4. On the Nodes tab, add the unsupported nodes.
   a. Navigate to and select LVEUnsupportedNodes.vi, which is the VI that contains the unsupported nodes.
   b. (Optional) Navigate to and select the VI that contains the supported nodes.
   c. Place a checkmark in the Unspecified nodes are unsupported checkbox if you are using both unsupported and supported VIs and you want nodes to be unsupported if the nodes are not in either VI.

5. On the Functions tab, add the unsupported functions.
   a. Navigate to and select LVEUnsupportedFunctions.vi, which is the VI that contains the unsupported functions.
   b. (Optional) Navigate to and select the VI that contains the supported functions.
   c. Place a checkmark in the Unspecified nodes are unsupported checkbox if you are using both unsupported and supported VIs and you want functions to be unsupported if the functions are not in either VI.

6. On the Data Types tab, add the unsupported data types.
   a. Navigate to and select LVEUnsupportedTypes.vi, which is the VI that contains the unsupported data types.
b. (Optional) Navigate to and select the VI that contains the supported data types.

c. Place a checkmark in the **Unspecified nodes are unsupported** checkbox if you are using both unsupported and supported VIs and you want data types to be unsupported if the data types are not in either VI.

7. On the **SubVIs** tab, add the unsupported subVIs. Unsupported subVIs must be `vi.lib`.

8. On the **Default Type** tab, add the default data type.

   a. Navigate to and select `LVEDefaultNumericType.ctl`, which is the custom control that contains the default data type.

9. (Optional) On the **Misc.** tab, adjust the settings for maximum recursion depth, variable support, and so on. You usually can keep the default miscellaneous settings.

   - **Max Type Depth** specifies the maximum recursion depth of supported data types. For example, a cluster of an array of integers has a type depth of 3. A cluster of an array of a cluster of strings has a type depth of 4. A value of -1 ignores the maximum type depth.

   - **Max Array Dimensions** specifies the maximum number of dimensions arrays allow. A value of -1 ignores the maximum array dimension.

   - **Support global variables** specifies if the target supports global variables.

   - **Support local variables** specifies if the target supports local variables.

   - **Errors break the VI** specifies if target syntax errors break the VI.

   - **Errors are considered warnings and do not break the VI** specifies if target syntax errors are considered warnings.

A few situations where you might want to change the Misc. settings include:

- You can set the **Max Type Depth** to prevent nested types from being recursive, which is not permitted by MISRA (The Motor Industry Software Reliability Association) automotive coding standards.

- You can set the **Max Array Dimensions** to 1 to restrict users to only optimized 1D array operations to achieve higher performance.
10. Enter the directory path to the target in the **Location of target** field.

11. Click the **Generate** button to create `TargetSyntax.xml`, which is the file LabVIEW uses to check target syntax, in the target directory you specified in step 10.

### Modifying the Target Syntax for an Embedded Target

Use the **Configure Target Syntax** dialog box to modify the XML file LabVIEW uses to check the target syntax.

Complete the following steps to modify the target syntax for a target.

1. Modify the unsupported and/or supported target syntax VIs. For example, if you want to add support for a VI, remove that VI from the unsupported VI.

2. Select **Tools»Embedded Tools»Target Syntax Editor** to open the **Configure Target Syntax** dialog box.

3. Click the **Open** button.

4. Navigate to and select the `TargetSyntax.xml` file for the target you are modifying.

5. Modify as necessary.

6. Click the **Generate** button to regenerate the `TargetSyntax.xml` file, which is the file LabVIEW uses to determine target syntax.
Technical Support and Professional Services

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

- **Support**—Online technical support resources at ni.com/support include the following:
  - **Self-Help Resources**—For answers and solutions, visit the award-winning National Instruments Web site 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.
  - **Free Technical Support**—All registered users receive free Basic Service, which includes access to hundreds of Application Engineers worldwide in the NI Discussion Forums at ni.com/forums. National Instruments Application Engineers make sure every question receives an answer.

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

- **Training and Certification**—Visit ni.com/training for self-paced training, eLearning virtual classrooms, interactive CDs, and Certification program information. You also can register for instructor-led, hands-on courses at locations around the world.

- **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.

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