LabVIEW™

Microprocessor SDK Porting Guide
Worldwide Technical Support and Product Information

ni.com

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

Worldwide Offices
Australia 1800 300 800, Austria 43 662 457990-0, Belgium 32 (0) 2 757 0020, Brazil 55 11 3262 3599, Canada 800 433 3488, China 86 21 5050 9800, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 358 (0) 9 725 72511, France 01 57 66 24 24, Germany 49 89 7413130, India 91 80 41190000, Israel 972 3 6393737, Italy 39 02 41309277, Japan 0120-527196, Korea 82 02 3451 3400, Lebanon 961 (0) 1 33 28 28, Malaysia 1800 887710, Mexico 01 800 010 0793, Netherlands 31 (0) 348 433 466, New Zealand 0800 553 322, Norway 47 (0) 66 90 76 60, Poland 48 22 3390150, Portugal 351 210 311 210, Russia 7 495 783 6851, Singapore 1800 226 5886, Slovenia 386 3 425 42 00, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 (0) 8 587 895 00, Switzerland 41 56 2005151, Taiwan 886 02 2377 2222, Thailand 662 278 6777, Turkey 90 212 279 3031, United Kingdom 44 (0) 1635 523545

For further support information, refer to the Technical Support and Professional Services appendix. To comment on National Instruments documentation, refer to the National Instruments Web site at ni.com/info and enter the info code feedback.

© 2005–2008 National Instruments Corporation. All rights reserved.
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 DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. THIS LIMITATION OF 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 THE PERFORMANCE OR COMPLETION THEREOF DUE TO REASONS REASONABLE CONTROL. 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 THE PERFORMANCE OR COMPLETION THEREOF DUE TO REASONS REASONABLE CONTROL. 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 THE PERFORMANCE OR COMPLETION THEREOF DUE TO REASONS 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. Wherever a product or software is used, the original software or other materials belonging to National Instruments or other parties may be reproduced 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.

The Bluetooth® word mark is a registered trademark owned by the Bluetooth SIG, Inc. Other product and company names mentioned herein are trademarks or trade names of their respective companies.

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 media, 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, OR 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 APPLICATION DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER 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 ELECTRONIC 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 IS 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.
Conventions

The following conventions are used in this manual:

» The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **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.

**italic**

Italic text denotes variables, emphasis, a cross-reference, or an introduction to a key concept. Italic text also denotes text that is a placeholder for a word or value that you must supply.

**monospace**

Text in this font denotes text or characters that you 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.

**Platform**

Text in this font denotes a specific platform and indicates that the text following it applies only to that platform or product.
Contents

Chapter 1  
Porting LabVIEW to a New Embedded Target

PART I  
Essential Pieces to Porting LabVIEW

Chapter 2  
Example Targets

Selecting an Appropriate Example Target................................................................. 2-1
System Requirements for Example Targets................................................................. 2-3
LabVIEW Directory Hierarchy..................................................................................... 2-4
Naming Conventions ..................................................................................................... 2-6
  Example Targets........................................................................................................... 2-6
  Plug-In Directories ..................................................................................................... 2-6
  Plug-In VIs ............................................................................................................... 2-7
Subtargeting .................................................................................................................... 2-7
Setting Up the Example Targets .................................................................................... 2-8
  Axiom CMD565 Example Target ............................................................................. 2-8
  Axiom CMD565, VxWorks Example Target ............................................................... 2-8
  Axiom CMD565, eCos Example Target .................................................................... 2-15
  Freescale ColdFire, uClinux Example Target ........................................................... 2-19
    Installing the uClinux Image for the ColdFire Example Target ......................... 2-20
    Setting the MAC Address for the ColdFire Example Target ............................... 2-21
    Configuring the ColdFire Example Target to Use DHCP or a Static IP Address .... 2-21
    Downloading and Installing the Sourcery G++ Lite Edition for the ColdFire Architecture .............................................................. 2-22
  PHYTEC Example Target ......................................................................................... 2-22
    PHYTEC Connections ......................................................................................... 2-22
    Installing the RedBoot Bootloader ...................................................................... 2-24
    Using the GNU Toolchain with the PHYTEC Example Target ......................... 2-24
  Spectrum Digital DSK6713 Example Target ........................................................... 2-25
    Using the TI Code Composer Studio Toolchain with the DSK6713 Example Target .............................................................................. 2-25
    Unix Console and Unix UI Example Targets ....................................................... 2-25
    Unicon UCN2410-CWIFI Example Target ............................................................. 2-26
Chapter 3
Important Files for Porting LabVIEW

Plug-In VIs .................................................................................................................... 3-1
Implementing Basic User Actions .............................................................................. 3-1
Target-Specific Dialog Boxes ...................................................................................... 3-2
LEP_x_ScriptCompiler.vi ............................................................................................. 3-2
   LEP_x_ScriptCompiler.vi Implementation Example ........................................... 3-3
LEP_x_ScriptLinker.vi ................................................................................................. 3-3
   LEP_x_ScriptLinker.vi Implementation Example .............................................. 3-4
LVDefs_plat.h ............................................................................................................... 3-5
lvEmbeddedMain.c ...................................................................................................... 3-5
LVSysIncludes.h .......................................................................................................... 3-6
TgtSupp.xml ................................................................................................................ 3-6
TargetSyntax.xml ....................................................................................................... 3-6

Chapter 4
LabVIEW C Code Run-Time and Analysis Libraries

LabVIEW C Code Run-Time Library ............................................................................ 4-1
Analysis Library ......................................................................................................... 4-2

Chapter 5
Defining Code Generation Options

Code Generation Attributes ......................................................................................... 5-2
   BigEndian ............................................................................................................... 5-2
   DestinationFolder ............................................................................................... 5-3
   ExpressionFolding ............................................................................................... 5-3
   GenerateFunctionCalls ...................................................................................... 5-3
   GenerateDebugInfo ............................................................................................. 5-4
   GenerateGuardCode ........................................................................................... 5-4
   GenerateInlinePrintf .......................................................................................... 5-5
   GenerateIntegerOnly .......................................................................................... 5-5
   GenerateLibraryInterface ................................................................................... 5-5
   GenerateSerialOnly ........................................................................................... 5-6
   IncrementalBuild ............................................................................................... 5-7
   interruptServiceRoutineVIs ............................................................................... 5-7
   MemoryModel ....................................................................................................... 5-8
   OCDIComments ................................................................................................... 5-8
   Silent ................................................................................................................... 5-8
   TargetAlignment ................................................................................................. 5-9
   UseStackVariables .............................................................................................. 5-9
Generating the Fastest Code ....................................................................................... 5-9
Expression Folding Patterns .................................................................5-10
  Pattern 1............................................................................................5-10
  Pattern 2............................................................................................5-11
  Pattern 3............................................................................................5-11
  Pattern 4............................................................................................5-12
  Pattern 5............................................................................................5-12
  Pattern 6............................................................................................5-12
  Pattern 7............................................................................................5-13
  Pattern 8............................................................................................5-14
Signal Naming Convention.................................................................5-14

Chapter 6
Defining Compiler Directives
  Debugging Macros..............................................................................6-1
  Feature Macros...................................................................................6-2
  Memory Mapping Macros.................................................................6-2
  General Macros..................................................................................6-3
  Operating System Macros.................................................................6-3

Chapter 7
Creating a New Embedded Target
  Copying and Renaming an Existing Target.........................................7-1
  Creating a New Target from Scratch................................................7-2
  Modifying an Existing Target.............................................................7-3
  Configuring Target Properties..........................................................7-4
    Configuring General Items...............................................................7-4
    Defining Shortcut Menu Items........................................................7-6
    Defining Target Properties Categories..........................................7-6
    Defining Timing Sources.................................................................7-7
    Defining Events................................................................................7-7
    Defining Signed Files.......................................................................7-7
    Configuring Licensing.....................................................................7-8
  Configuring Build Specification Type Properties..............................7-9
    Configuring General Build Specification Items...............................7-9
    Defining Build Specification Shortcut Menu Items........................7-10
    Defining Build Specification Categories........................................7-10
    Defining Build Specification Events...............................................7-11
  Creating New Plug-In or Subpanel VIs..............................................7-11
    Creating New Plug-In VIs.................................................................7-12
    Creating New Subpanel VIs.............................................................7-12
Contents

Chapter 8
Configuring the Target Syntax
Adding Target Syntax to an Embedded Target ............................................................. 8-1
Modifying the Target Syntax for an Embedded Target ................................................. 8-4

Chapter 9
Implementing Serial or TCP Instrumented Debugging
Serial Communication ................................................................................................... 9-1
TCP Communication ..................................................................................................... 9-2
Log Files for Troubleshooting ....................................................................................... 9-2

Chapter 10
Testing Your Target Implementation
LabVIEW C Generator Test Framework ...................................................................... 10-1
Default Tests ................................................................................................... 10-2
Adding Your Target to the LabVIEW C Generator Test Framework ............ 10-3
Emailing Failed Test Results ..................................................................................... 10-4
Creating Configuration Files ........................................................................... 10-5
Default Configuration Files ............................................................................ 10-6
Autotest Implementation Example ........................................................................... 10-6
C Generator-Specific VI Analyzer Tests ....................................................................... 10-7
Log Files for Troubleshooting ....................................................................................... 10-8

Part II
Advanced Porting Implementations

Chapter 11
Implementing User Interface Support
Limitations in Designing User Interfaces ................................................................... 11-2
Front Panel Controls and Indicators ............................................................... 11-3
Native Controls and Indicators ........................................................................... 11-3
Array Controls and Indicators ............................................................................ 11-4
Boolean Controls ..................................................................................................... 11-4
Color in Boolean Controls ................................................................................. 11-4
Checkboxes .............................................................................................................. 11-4
Boolean Text ........................................................................................................... 11-5
Charts ......................................................................................................................... 11-5
Color ......................................................................................................................... 11-5
Cluster Controls and Indicators ............................................................................. 11-6
Combo Boxes .......................................................................................................... 11-6
Decorations .............................................................................................................. 11-6
## Enumerated Type Controls
- Page 11-7

## Fonts
- Page 11-7

## Graphs
- Page 11-8
  - Plots on Waveform Graphs
  - Plots on XY Graphs

## Labels
- Page 11-9

## Listboxes
- Page 11-9
  - Multicolumn Listboxes

## Menus
- Page 11-9

## Numeric Controls and Indicators
- Page 11-9

## Path Controls
- Page 11-10

## Picture Controls
- Page 11-10
  - Events
  - Showing Ampersands (&) in Picture Controls

## Refnums
- Page 11-11

## Scaling Front Panel Objects
- Page 11-11

## String Controls and Indicators
- Page 11-12
  - Using Password Display

## Tab Controls
- Page 11-12

## Time Stamp Controls and Indicators
- Page 11-13
  - Minimum and Maximum Time Values
  - Editing Modes

## Variant Controls
- Page 11-13

## Supported Properties
- Page 11-14
  - Generic Properties
  - Array Controls and Indicators
  - Boolean Controls and Indicators
  - Cluster Controls
  - Enumerated Type Controls
  - Graphs and Charts
  - Listbox Controls
  - Numeric Controls and Indicators
  - Ring Controls
  - String Controls and Indicators
  - Time Stamp Controls and Indicators
  - Waveform Charts
  - Waveform Graphs

## Supported Methods
- Page 11-18
  - Generic Methods
  - Graphs and Charts
  - String Controls and Indicators
  - Waveform Graphs
  - VIs

## Unsupported Controls and Indicators
- Page 11-19
Chapter 12
Elemental I/O

I/O Devices.................................................................................................................... 12-1
I/O Classes ................................................................................................................... 12-2
I/O Pins ....................................................................................................................... 12-2
I/O Resources .............................................................................................................. 12-2
Elemental I/O Plug-In VIs .......................................................................................... 12-2
How Users Use Elemental I/O Nodes ........................................................................... 12-3
Creating Elemental I/O Plug-In VIs .............................................................................. 12-3
I/O Node Implementation Plug-In VI ........................................................................... 12-4
I/O Property Node Implementation Plug-In VI ............................................................ 12-5
I/O Specify Node Plug-In VI ....................................................................................... 12-5
I/O Project and Node Attribute Validation Plug-In VI ................................................ 12-5
Using the Elemental I/O Device Wizard to Create Elemental I/O Devices, Classes, Pins, and Resources .................................................................................................................... 12-6
Adding Elemental I/O Devices to an Embedded Target .............................................. 12-8
Elemental I/O Implementation Example ...................................................................... 12-8

Chapter 13
Porting the LabVIEW C Code Run-Time Library to a New Platform

#define in LVDefs_plat.h ............................................................................................. 13-1
Feature Flags ................................................................................................................. 13-2
OS-Specific Components ............................................................................................. 13-3
Critical Sections: LVCritSect.c ................................................................................... 13-3
Events: LVEvent.c ....................................................................................................... 13-3
Threads: LVThreads.c .................................................................................................. 13-4
Non-Blocking Operations: LVNBOps.c ...................................................................... 13-5
LabVIEW-Based Interrupt Service Routines: OEM_LVISR.c ...................................... 13-6
Printf: PDAStrSupport_os.c ....................................................................................... 13-6
Time: CCGTimeSupport_os.c ..................................................................................... 13-7
Serial: PlatformSerial_os.c ......................................................................................... 13-7
CAN: PlatformCAN_os.c ............................................................................................ 13-8
TCP/UDP: CCGNetConnSupport.c ............................................................................... 13-10
Static Memory Allocation Support ................................................................................ 13-10
RT FIFO Size Limits .................................................................................................... 13-11
## Chapter 14
### Implementing Interrupt Service Routine Support

<table>
<thead>
<tr>
<th>ISR Functions</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitOEMISRs</td>
<td>14-1</td>
</tr>
<tr>
<td>UninitOEMISRs</td>
<td>14-1</td>
</tr>
<tr>
<td>OEMISRRegisterHandler</td>
<td>14-2</td>
</tr>
<tr>
<td>OEMISRUnregisterHandler</td>
<td>14-2</td>
</tr>
<tr>
<td>Tips for Supporting ISRs on a New Target</td>
<td>14-2</td>
</tr>
<tr>
<td>ISR Implementation Example</td>
<td>14-3</td>
</tr>
</tbody>
</table>

## Chapter 15
### Timing

<table>
<thead>
<tr>
<th>Using the Target Editor to Define Timing Sources</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateInternalTSource()</td>
<td>15-1</td>
</tr>
<tr>
<td>FireInternalTSource()</td>
<td>15-2</td>
</tr>
<tr>
<td>DeleteInternalTSource()</td>
<td>15-2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Using the Timed Loop VIs to Define Timing Sources</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15-3</td>
</tr>
</tbody>
</table>

## Chapter 16
### Memory Mapping

<table>
<thead>
<tr>
<th>Implementing Memory Mapping</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Mapping Plug-In VIs</td>
<td>16-1</td>
</tr>
<tr>
<td>LEP_x_SectionNames.vi</td>
<td>16-2</td>
</tr>
<tr>
<td>LEP_x_MemoryMap_Default.vi</td>
<td>16-2</td>
</tr>
<tr>
<td>LEP_x_MemoryMap_Cmp.vi</td>
<td>16-2</td>
</tr>
<tr>
<td>LEP_x_MemoryMap.vi</td>
<td>16-2</td>
</tr>
<tr>
<td>LEP_x_MemoryMap_Query.vi</td>
<td>16-3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How Users Use Memory Mapping</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16-3</td>
</tr>
</tbody>
</table>

## Chapter 17
### Implementing Instrumented Debugging for a Custom Communication Layer

<table>
<thead>
<tr>
<th>Implementing the Instrumented Debugging Communication Layer for LabVIEW</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open VI</td>
<td>17-1</td>
</tr>
<tr>
<td>Close VI</td>
<td>17-2</td>
</tr>
<tr>
<td>Read VI</td>
<td>17-2</td>
</tr>
<tr>
<td>Write VI</td>
<td>17-3</td>
</tr>
</tbody>
</table>
Implementing the Instrumented Debugging Communication Layer for Your Embedded Target

Chapter 18
On Chip Debugging

Debug Database ................................................................. 18-3
C Parser ................................................................. 18-3
C Compiler .............................................................. 18-5
List Parser ............................................................... 18-6
Linker ........................................................................ 18-8
Map Parser ..................................................................... 18-8
Debug Daemon ......................................................... 18-9
Comparing On Chip and On Chip (Release) Modes .............. 18-9
Adding Support for an Emulator that Uses the GCC Toolchain ........................................ 18-10
Adding Support for an Emulator that Uses a Non-GCC Toolchain ......................................... 18-12
  EMB_Debug_Map_x_UpdateDD.vi ............................................................ 18-12
  EMB_Debug_Map_x.ResolveSymbols.vi .............................................. 18-12
Resolving variable.member (type_code) Signals.......................... 18-12
Adding Support for a New Emulator Interface ....................... 18-14
OCDI Debugging Example ............................................. 18-14
List Parser Plug-In VIs ...................................................... 18-15
  EMB_Debug_Map_RelativeAddress.vi ........................................... 18-15
  EMB_Debug_Map_SignalOffset.vi ............................................... 18-15
OCDI Debugging Plug-In and Utility VIs................................. 18-16
  IInit.vi ....................................................................... 18-16
  IClearBreakpoints.vi .................................................. 18-18
  IConfig.vi .................................................................. 18-19
  IContExecution.vi ..................................................... 18-20
  IDownload.vi .......................................................... 18-22
  IExit.vi .............................................................. 18-23
  IGetBreakpoint.vi .................................................... 18-24
  IGo.vi .................................................................. 18-26
  IMemoryRead.vi ........................................................ 18-27
  IMemoryWrite.vi ......................................................... 18-28
  IPauseExecution.vi ..................................................... 18-30
Chapter 19
Understanding the LabVIEW Embedded LVM File

Format of the LVM File ................................................................. 19-1
Example LVM File ...................................................................... 19-2
VI Section .................................................................................. 19-4
Offset_Table Section ................................................................. 19-8
Generating the LVM File ............................................................. 19-18
Plug-In VIs for the Debug Database ............................................. 19-19
  EMB_Debug_DD_GetBP.vi ...................................................... 19-20
  EMB_Debug_DD_GetMem.vi .................................................... 19-22
  EMB_Debug_DD_Init.vi ......................................................... 19-24
  EMB_Debug_DD_ParseCFile.vi .............................................. 19-25
  EMB_Debug_DD_Save.vi ......................................................... 19-26
  EMB_Debug_DD_SetAbsoluteBPAddress.vi .......................... 19-27
  EMB_Debug_DD_SetAbsoluteMemAddress.vi ......................... 19-29

Chapter 20
Integrating LabVIEW with Another IDE

Additional Debugging Features with IDE Integration .................... 20-1
Example of Code Comments ...................................................... 20-2
Example IDE Integration Implementation .................................. 20-3
  Installing the Necessary Tools for LabVIEW and Eclipse Integration ........ 20-3
    Eclipse ............................................................................... 20-3
    C/C++ Development Tools for Eclipse (CDT) SDK .................. 20-3
    LabVIEW Embedded Eclipse Plug-In .................................. 20-4
    Cygwin 1.5.23-2 or later ...................................................... 20-4
  Implementation Example ....................................................... 20-5
    Debugging Backend .......................................................... 20-5
    Target Properties Dialog Box .......................................... 20-5
IDE Integration Plug-In VIs .......................................................... 20-6
  IInitIndication.vi ................................................................. 20-6
  IGetIndication.vi ................................................................. 20-7
  IShowCrash.vi ................................................................... 20-9
  IResolveAddress.vi .............................................................. 20-10
Chapter 21
Overriding SubVI Calls with C Function Calls to Improve Performance
Format of Funclist Entries ................................................................. 21-2
<VI Name> .................................................................................. 21-2
<Function>..................................................................................... 21-2
    Passing Parameters Directly .................................................. 21-2
    Passing Parameters by ArgList ............................................. 21-3
<Feature> ...................................................................................... 21-3
Implementing Target-Specific Funclist Files ...................................... 21-4
Debugging and Troubleshooting Funclist Entries .............................. 21-4
Implementation Example................................................................. 21-5

Chapter 22
Static Memory Allocation
Supporting a Static Memory Model .................................................. 22-1
Limitations of the Static Memory Model ......................................... 22-3
    OEM-Defined Limitations .................................................... 22-3
    User Limitations ................................................................. 22-3

Chapter 23
Using the Target Server
Target Server Window ..................................................................... 23-1
Target Server Daemon ..................................................................... 23-2
Telnet Client .................................................................................. 23-2
Implementation Example................................................................. 23-2

Appendix A
Technical Support and Professional Services
Porting LabVIEW to a New Embedded Target

Use the LabVIEW Microprocessor SDK to port LabVIEW to any 32-bit microprocessor. By using a single development tool from concept to finished product, you can ease the development process and increase end quality while reducing time to market.

The porting process includes several steps; some steps are required and some steps are optional depending on your target and the features you want to implement and support.

The main steps to porting LabVIEW include the following:

1. Obtaining the necessary toolchain and board support package (BSP) for your target.
2. Compiling, downloading, running, and debugging a “hello world” application using the toolchain and BSP. This step verifies that the necessary toolchain is installed on the host computer and the board support package is installed and configured correctly.
3. Porting the LabVIEW Run-Time Library to the target operating system.
4. Creating and/or modifying the plug-in VIs for basic user actions and target-specific dialog boxes.
5. Adding the target to the LabVIEW development environment.

The actual amount of implementing versus reusing of example targets depends on how closely your target, operating system, and toolchain match one of the example targets, operating systems, and toolchains. While the example targets include common processor architectures, operating systems, and toolchains, the examples targets cannot cover everything.
The Microprocessor SDK includes the LabVIEW C Code Generator, which generates C code based on the block diagram when you build an embedded VI into an embedded application. Next, the C code is passed with any external C code and the LabVIEW Run-Time Library through your third-party cross-compiler to create an executable file. This executable file is saved on the host computer.

When you download, or deploy, an embedded application, your toolchain downloads the application, usually over serial, TCP, or JTAG. If you run the embedded application, the go command is sent for that application. A basic embedded application runs headless, which means it runs without a user interface, keyboard, mouse, and so on. If your target has an LCD and you implement user interface support, your embedded application might have a user interface. When you debug an embedded application, you create an interactive debug connection back to the host PC, usually over serial, TCP, or JTAG.

Part I, Essential Pieces to Porting LabVIEW, contains information about the essential pieces to porting LabVIEW to a new embedded target. Part II, Advanced Porting Implementations, contains information about some advanced porting functionality that might be necessary depending on your target.
Part I

Essential Pieces to Porting LabVIEW

Exactly how you port LabVIEW to a new embedded target depends on your target and functionality. The essential pieces to porting LabVIEW include the following:

- Selecting one or more example targets, if possible, to use as a starting point to implementing your own target
- Becoming familiar with important files related to porting, the LabVIEW C Code Run-Time Library, and the LabVIEW Analysis Library
- Understanding the LabVIEW C Code Generator, code generation attributes, and compiler options
- Using the Target Editor to create a new target
- Configuring the target syntax
- Implementing serial or TCP instrumented debugging
- Testing your target implementation

Some targets might require additional, advanced porting implementations. Refer to Part II, Advanced Porting Implementations, for more information about advanced functionality.
Example Targets

The LabVIEW Microprocessor SDK 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

Selecting an appropriate example target is a good place to start when you port LabVIEW to a new embedded target. Use the target that is closest to your target and toolchain. For example, 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 that uses the same operating system.

The Microprocessor SDK 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 example target that is similar to your target. Depending on the feature, you also might want to look for an existing implementation that uses the same operating system as your target.
The following table lists which example targets contain example implementations of different features. Use this table to find an example of a feature you are implementing for your target.

<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>
<th>UI</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>
<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>
<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>
<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>
<td>No</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>
<td>No</td>
</tr>
<tr>
<td>Axiom CMD565, VxWorks ROM Image</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</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>
<td>No</td>
</tr>
<tr>
<td>Freescale ColdFire M5329EVB, uClinux</td>
<td>TCP</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PHYTEC LPC229x, eCos</td>
<td>Serial</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</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>
<td>No</td>
</tr>
<tr>
<td>Unicon UCN2410-CW IFI, Linux</td>
<td>TCP</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### System Requirements for Example Targets

The Microprocessor SDK 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 RAM Image</td>
<td>Axiom CMD-565 Development Board</td>
<td>Cygwin 1.5.x, eCos 2.0 PowerPC toolchain</td>
</tr>
<tr>
<td>Axiom CMD565, eCos ROM Image</td>
<td>Axiom CMD-565 Development Board</td>
<td>Cygwin 1.5.x, eCos 2.0 PowerPC toolchain</td>
</tr>
<tr>
<td>Axiom CMD565, VxWorks Module</td>
<td>Axiom CMD-565 Development Board</td>
<td>Wind River Tornado 2.2.1, VxWorks 5.5.1 BSP for CMD565</td>
</tr>
<tr>
<td>Axiom CMD565, VxWorks RAM Image</td>
<td>Axiom CMD-565 Development Board, iSYSTEM iC3000ActiveEmulator</td>
<td>Wind River Tornado 2.2.1, VxWorks 5.5.1 BSP for CMD565, 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, VxWorks 5.5.1 BSP for CMD565</td>
</tr>
<tr>
<td>VxWorks Simulation</td>
<td>None</td>
<td>Wind River Tornado 2.2.1, VxWorks 5.5.1, (Optional) ULIP Ethernet driver</td>
</tr>
<tr>
<td>Freescale ColdFire M5329EVB, uClinux</td>
<td>ColdFire M5329EVB Development Kit</td>
<td>CodeSourcery ColdFire uClinux toolchain release 4.1-11 m68k, CF Flasher 3.1, P&amp;E Device Drivers</td>
</tr>
<tr>
<td>PHYTEC LPC229 x, eCos</td>
<td>phyCORE-ARM7/LPC229x Rapid Development Kit, GPIO Expansion Board</td>
<td>Cygwin 1.5.x, eCos 2.0 ARM toolchain</td>
</tr>
</tbody>
</table>
Chapter 2  Example Targets

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

Refer to ecos.sourceforge.net/getstart.html for information about downloading and installing eCos.

Refer to gcc.gnu.org for information about downloading and installing gcc.

The VxWorks for LabVIEW Embedded Development Module Evaluation Kit 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 Microprocessor SDK. Refer to windriver.com/alliances/eval-cd, click National Instruments Evaluation CD Program, and follow the instructions to receive the VxWorks for LabVIEW Embedded Development Module Evaluation 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.

LabVIEW Directory Hierarchy

You do not need to be familiar with the entire labview directory structure, but the following directories are important to porting LabVIEW:

- **autotest**—Contains the LabVIEW C Generator Autotest Framework and test VIs. The VIs are in the tests\Large semi-auto tests subdirectory.
• **CCodeGen**—Contains any Funclist files and the following subdirectories:
  - **analysis**—Contains the LabVIEW Analysis Library source code.
  - **include**—Contains the include files for the LabVIEW C Code Run-Time Library. This directory contains several subdirectories, but the **os** subdirectory is the directory you work with the most.
    - **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 work with the most.
    - **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 examples for working with interrupts and the Inline C Node.
• **help**—Contains help files.
• **manuals**—Contains PDF documentation for LabVIEW and any additional modules or toolkits.
• **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 Microprocessor SDK 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. The top-level targets are self-contained and do not rely on any subtarget implementations below the top-level target. In contrast, the subtargets reuse and 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 create an Embedded subdirectory under your company name directory when you create your target.

- vi.lib—Contains libraries of built-in VIs. The following subdirectories are important when porting LabVIEW:
  - 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 on the Functions palette or used by your users.

Naming Conventions

Targets, plug-in directories, and plug-in VIs have the following naming conventions.

Example Targets

National Instruments recommends, but does not require, you follow the same naming convention as the Microprocessor SDK 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.

Plug-In Directories

Each example operating system contains OS-specific plug-in VIs. The top-level plug-in VI directories have the following naming convention:

labview\Targets\Company Name\Embedded\OS\OS_LEP_TargetPlugin

For example, labview\Targets\NI\Embedded\ecos\ecos_LEP_TargetPlugin
Plug-In VIs

The plug-in VIs have the following naming convention:

LEP_OS_VIName.vi

For example, LEP_ecos_LoadProjectInfo.vi

Subtargeting

Some example operating systems contain multiple example targets. Example targets using these operating systems use the top-level plug-in VIs plus target-specific plug-in VIs. For example, the PHYTEC and Axiom CMD565 example targets use the eCos operating system. Those example targets use the plug-in VIs in the labview\Targets\NI\Embedded\ecos\ecos_LEP_TargetPlugin directory plus target-specific plug-in VIs, which are in a target-specific subdirectory. These target-specific plug-in VIs might implement target-specific functionality or override OS functionality the top-level plug-in VIs implement. The top-level targets are self-contained and do not rely on any subtarget implementations below the top-level target. In contrast, the subtargets rely on the top-level target implementation.

For example, the PHYTEC-specific plug-in VIs are in the following directory:

labview\Targets\NI\Embedded\ecos\phytec_lpc229x\ecos_Phytec_LPC229x_LEP_TargetPlugin

The PHYTEC-specific plug-in VIs use the following naming convention:

LEP_ecos_Phytec_LPC229x_Build.vi

The Axiom CMD565-specific plug-in VIs are in the following directory:

labview\Targets\NI\Embedded\ecos\cmd565\ecos_cmd565_LEP_TargetPlugin

The Axiom CMD565-specific plug-in VIs use the following naming convention:

LEP_ecos_cmd565_Build.vi
Setting Up the Example Targets

How you set up an example target to work with the Microprocessor SDK depends on the target. You must set up the example targets as described for the targets to work.

**Axiom CMD565 Example Target**

The Microprocessor SDK includes an Axiom CMD565 example target for VxWorks and eCos.

> **Note**  The Axiom CMD565 is a socketed CPU.

**Axiom CMD565, VxWorks Example Target**

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.

**Axiom CMD565, VxWorks Module Example Target**

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 the following figure. This configuration is the default 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. 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 the following figure.
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.
Axiom CMD565, VxWorks ROM Example Target

This target configuration allows you to download an application you build with the Microprocessor SDK 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 the following figure. 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 as shown in the following figure. Change the communication speed of the terminal program to 57,600 bps.
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 receive the VxWorks logo on one serial port. Verify you receive WDB READY on the other serial port (COM 2) communications at speed 57,600.

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.

Axiom CMD565, eCos Example Target

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.

Axiom CMD565, eCos RAM Example Target

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 the following figure. 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. Change the communication speed of the terminal program to 57,600 bps.


11. Run an example application to verify the setup.

**Axiom CMD565, eCos ROM Example Target**

This target configuration allows you to download an application you build with the Microprocessor SDK 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 the following figure. 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.

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 of 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.

Freescale ColdFire, uClunix Example Target

You must configure the Freescale MCF5329EVB or MCF5329EVE board before you can use it as a target in LabVIEW. The Microprocessor SDK includes a uClunix image that works with M5329EVB and M5329EVE evaluation boards. Configuring the target includes the following:
1. Installing the uClunix image.
2. Setting the Media Access Control (MAC) address.
3. Configuring the board to use DHCP or a static IP address.
4. Downloading and installing the ColdFire uClunix toolchain.
Installing the uClinux Image for the ColdFire Example Target

Complete the following steps to install the uClinux image.

1. Install the P&E BDM interface drivers from the CD that comes with the evaluation board or download and install the drivers from the P&E Microcomputer Systems Web site at www.pemicro.com. National Instruments recommends downloading and installing the drivers. If the P&E BDM interface drivers are already installed, you do not need to reinstall the drivers.


3. Use a USB cable to connect the USB-ML-CF emulator to the M5329EVB board and the host computer. The emulator is included with the M5329EVB board. Windows should recognize the emulator and install the necessary drivers.

4. Power on the evaluation board.

5. Select Programs>CF Flasher 3.1 to launch CF Flasher. Refer to the CF Flasher documentation for more information about using CF Flasher.

6. Click the Target Config button in the CF Flasher main dialog box to open the Target Configuration dialog box.

7. Select M5329EVB from the Target Configuration list.

8. Select PE_USB_ML from the BDM Communication list.

9. Click the OK button to close the Target Configuration dialog box.

10. Click the Program button.

11. Unzip m5329evb.zip, located in the labview\Targets\NI\Embedded\unix\m5329evb\bin\directory, to a temporary location.

12. Navigate to and select m5329evb.bin.

13. When you are prompted for the start address, enter 0 and click the OK button.

14. Click the Program button.

Note Installing the image can take a few minutes.
Setting the MAC Address for the ColdFire Example Target

Complete the following steps to set the MAC address, which also is called an Ethernet address or an IEEE MAC address.

1. Connect the M5329EVB board to the host computer using the serial cable, which is included with the M5329EVB board.
2. Launch your terminal program and set the communication speed to 115200.
3. Reset the board.
4. Press any key to prevent uClinux from booting and to open the dBUG monitor.
6. Type `show` and verify the MAC address.

Configuring the ColdFire Example Target to Use DHCP or a Static IP Address

You can configure the M5329EVB board to use the Dynamic Host Configuration Protocol (DHCP) or a static IP address. You have to configure only one or the other.

Complete the following steps to configure the board to use DHCP or a static IP address.

1. Reset the M5329EVB board.
2. Press any key to open the dBUG monitor.
3. To use DHCP, type `set kcl rootfstype=romfs ip=:::eth0:dhcp`.
   
   To use a static IP address, type `set kcl rootfstype=romfs ip=<ip>::<gateway>::<netmask>::eth0:off`.
4. Type `show` to check the configuration.
Chapter 2  Example Targets

Downloading and Installing the Sourcery G++ Lite Edition for the ColdFire Architecture

You must download and install the ColdFire uClinux version of the Sourcery G++ Lite Edition, which is the CodeSourcery version of the GNU toolchain, from the CodeSourcery Web site at www.codesourcery.com/gnu_toolchains/coldfire.

PHYTEC Example Target

You must do the following to use the PHYTEC example target in LabVIEW:

1. Establish the connections on the expansion board.
2. Install the RedBoot bootloader.
3. Install the GNU arm-elf toolchain.

The PHYTEC example target uses the following hardware:

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

Refer to the PHYTEC Web site at www.phytec.com for more information about PHYTEC hardware.

PHYTEC 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 Microprocessor SDK.

Note  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 NXP Web site at www.nxp.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>
<tr>
<td>LED 7</td>
<td>GPIO P1.22 (0)</td>
<td>14A</td>
<td>LED IN 7</td>
</tr>
<tr>
<td>LED 8</td>
<td>GPIO P1.23 (0)</td>
<td>14E</td>
<td>LED IN 8</td>
</tr>
<tr>
<td>DIP switch 0</td>
<td>GPIO P0.10 (0)</td>
<td>4F</td>
<td>Switch OUT 1</td>
</tr>
<tr>
<td>DIP switch 1</td>
<td>GPIO P0.11 (0)</td>
<td>5A</td>
<td>Switch OUT 2</td>
</tr>
<tr>
<td>DIP switch 2</td>
<td>GPIO P0.12 (0)</td>
<td>5C</td>
<td>Switch OUT 3</td>
</tr>
<tr>
<td>DIP switch 3</td>
<td>GPIO P0.13 (0)</td>
<td>5E</td>
<td>Switch OUT 4</td>
</tr>
<tr>
<td>KEY 4 (red)</td>
<td>EINT1 (2)</td>
<td>28B</td>
<td>/KEY OUT 4</td>
</tr>
<tr>
<td>KEY 5 (blue)</td>
<td>EINT2 (2)</td>
<td>28F</td>
<td>/KEY OUT 5</td>
</tr>
<tr>
<td>KEY 6 (black)</td>
<td>EINT3 (3)</td>
<td>29A</td>
<td>/KEY OUT 6</td>
</tr>
<tr>
<td>Motor A</td>
<td>PWM5 (1)</td>
<td>8B</td>
<td>MOT IN A (1)</td>
</tr>
<tr>
<td>Motor count</td>
<td>CAP1.2 (1)</td>
<td>8D</td>
<td>Count Out (3)</td>
</tr>
<tr>
<td>AIN 0</td>
<td>AIN0 (1)</td>
<td>17A</td>
<td>Poti OUT 1</td>
</tr>
<tr>
<td>AIN 1</td>
<td>AIN1 (1)</td>
<td>16F</td>
<td>Poti OUT 2</td>
</tr>
<tr>
<td>AIN 2</td>
<td>AIN2 (1)</td>
<td>16B</td>
<td>Poti OUT 3</td>
</tr>
<tr>
<td>VDD 3.3 V</td>
<td>—</td>
<td>1A</td>
<td>VCC</td>
</tr>
<tr>
<td>VDD 3.3 V</td>
<td>—</td>
<td>1C</td>
<td>VPoti</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 install 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. [kHX]: 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 with the PHYTEC Example Target

The PHYTEC example target uses the GNU arm-elf toolchain. Refer to the eCos Web site at ecos.sourceforge.org 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 Microprocessor SDK:

- 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 SYSTEM path variable.
Spectrum Digital DSK6713 Example Target


Using the TI Code Composer Studio Toolchain with the DSK6713 Example Target

The Spectrum Digital DSK6713 uses the TI 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 (16 Mb) Quick Start Installation Guide on the CD for hardware installation and set up instructions. The quick start guide is available with the DSP Starter Kit. If you install Code Composer Studio in the default location, refer to c:\CCstudio\docs\pdf\manuals_ccs_full_c6000.html for a complete list of related documentation, including how to set-up Code Composer Studio.

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.

Unix Console and Unix UI Example Targets

You must install Cygwin 1.5.23-2 or later to use the Unix Console or Unix UI example target. Install the following Cygwin GNU tools:

- binutils 20060817-1 or later
- gcc 3.4.4 or later
- gdb 20060706-2 or later
- make 3.81 or later

To use the Unix UI example target, you also must install the following GNU libraries:

- libjpeg-devel 6b-12
- libjpeg62 6b-12
- libpng12-devel version 1.2.12-x
- libpng12 1.2.12-x
Refer to the Cygwin Web site at www.cygwin.com for download and installation instructions.

**Unicon UCN2410-CWIFI Example Target**

Complete the following steps to set up the Unicon UCN2410-CWIFI example target.

1. Install Cygwin 1.5.23-2 or later. Refer to the Cygwin Web site at www.cygwin.com for download and installation instructions. Install the following Cygwin GNU tools:
   - binutils 20060817-1 or later
   - gcc 3.4.4 or later
   - gdb 20060706-2 or later
   - make 3.81 or later
2. Download and install the gcc toolchain.
   a. Unzip the file to cygwin\opt\ where cygwin is where you installed Cygwin. This step creates an arm-linux-tools folder.
   b. Edit the Windows system PATH variable to include cygwin\opt\arm-linux-tools\bin.
3. Launch your terminal program and set the following serial port settings:
   - Bits per second = 115200
   - Data bits = 8
   - No parity
   - Stop bits = 1
   - No flow control
4. Plug in the hardware. The login prompt ucn-2410 login: appears after the hardware boots.
5. Type root and press the <Enter> key.
6. Leave the password blank and press the <Enter> key.
7. At the ~# prompt, type cd /tmp and press the <Enter> key.
8. At the /var/tmp # prompt, type rx lvprep.sh and press the <Enter> key.
9. Using your terminal application, send the labview\Targets\NI\Embedded\linux\UCN2410\drivers\lvprep.sh file using the Xmodem protocol.
10. Type `rx lvprep.tgz` and press the <Enter> key.
11. Using your terminal application send the `labview\Targets\NI\Embedded\linux\UCN2410\drivers\lvprep.tgz` file using the Xmodem protocol.
12. Type `chmod +x lvprep.sh` and press the <Enter> key.
13. Type `lvprep.sh` and press the <Enter> key. `lvprep.sh` unpacks `lvprep.tgz` and makes the necessary symbolic links and file changes.
14. Type `reboot` and press the <Enter> key to reboot the board. The `ucn-2410 login:` prompt appears when the board completes rebooting.
15. Unplug and replug the USB cable so Windows recognizes the USB device.
16. Follow the instructions in the Microsoft Windows Hardware Update Wizard to install the driver located in the `labview\Targets\NI\Embedded\linux\UCN2410\drivers` directory.

   **Note** Ignore the warning that the driver is not signed by Microsoft.

17. The USB connection to the board appears as a local network connection in Windows, so you must set an IP address for the network connection. Use the following IP addresses:
   - Host IP address = 192.168.1.100
   - Subnet mask = 255.255.255.0
   - Default gateway = 192.168.1.100 (same as Host IP address)

   By default, LabVIEW sets the board IP address, which is 192.168.1.1, in the **Target Properties** dialog box.

   **Note** If you want to set a password for the board, set the password on the board first and then add it to LabVIEW in your **Target Properties** dialog box.
Important Files for Porting LabVIEW

How to port LabVIEW to a target is very target-specific, but you must be aware of and modify some important files that are fundamental to porting LabVIEW to a new target.

Plug-In VIs

Plug-in VIs customize the compiling and linking process, which include implementing basic user actions and target-specific dialog boxes.

Refer to the target plug-in directories for different plug-in VI implementation examples.

Implementing Basic User Actions

The basic user actions you must implement include the following:

- Build (includes linking and compiling)
- Download or deploy
- Run
- Debug

These basic user actions build on one another. For example, if you download an application that is not built, LabVIEW builds the application before downloading it. If you run an embedded application that has not been downloaded, LabVIEW downloads the application before running it, and so on.

LEP_x_Build.vi is the main plug-in VI that implements the build user action. This plug-in VI performs the following:

- Loads project configuration information.
- Creates the build folder on the host computer.
- Prepares the list of libraries, include directories, and source files for LEP_x_ScriptCompiler.vi and LEP_x_ScriptLinker.vi.
Chapter 3  Important Files for Porting LabVIEW

- Calls LEP_x_ScriptCompiler.vi to compile the source files into an object file.
- Calls LEP_x_ScriptLinker.vi to link the project libraries and the object files into an embedded application.

LEP_x_Run.vi is the main plug-in VI that implements the run user action. It runs the application on the target.

LEP_x_Download.vi is the main plug-in VI that implements the download user action.

Note Some example targets use different filenames for the plug-in VIs and some might contain additional or different plug-in VIs.

Target-Specific Dialog Boxes

Different targets require different target-specific dialog boxes. The two most common target-specific dialog boxes include the following:

- LEP_x_BuildSettingsDlg_x.vi is the Build Specification Properties dialog box for your target. Each category, or page, is a separate plug-in VI.
- LEP_x_TargetConfigDialog.vi is the Target Properties dialog box for your target.

Note Some example targets use a different filename for the Target Properties dialog box and some do not contain any target properties at all.

LEP_x_ScriptCompiler.vi

LEP_x_ScriptCompiler.vi, where x is a subtarget and/or an OS, scripts your toolchain compiler and passes the compiler directives to the compiler from the command line. LEP_x_ScriptCompiler.vi is located in the following directory:

labview\Targets\company_name\Embedded\os\os_LEP_TargetPlugin

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

You must implement LEP_x_ScriptCompiler.vi if you require behavior that differs from the default implementation. Use the base target default implementation of LEP_x_ScriptCompiler.vi for your OS and override only the necessary features. To override the default
implementation, wire an 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.

LEP_x_ScriptCompiler.vi can execute the C compiler directly by calling the System Exec VI. You also can use a batch file to execute the compiler directly. Use a batch file if you need to set compiler-specific system variables before the compiler 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.

**LEP_x_ScriptCompiler.vi Implementation Example**

The VxWorks CMD565 example target implements a subtarget-specific LEP_vxworks_cmd565_ScriptCompiler.vi, 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_vxworks_ScriptCompiler.vi, LEP_vxworks_ScriptLinker.vi, and LEP_vxworks_Config.ctl. 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.

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_ScriptLinker.vi**

LEP_x_ScriptLinker.vi, where x is a subtarget and/or an OS, links all intermediate object files, external libraries, run-time libraries, and operating system libraries into an embedded application. LEP_x_ScriptLinker.vi is located in the following directory:

```
labview\Targets\company_name\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. Use the base example target default implementation for your OS of `LEP_x_ScriptLinker.vi` and override only the necessary features. To override the default implementation, wire an `LEP_x_ScriptLinker.vi` VI reference to the `LinkerScript` input of `LEP_x_Build.vi`. Your custom implementation of `LEP_x_ScriptLinker.vi` must have the same connector pane as the default implementation.

Note Some example targets avoid long command lines by using `prjObjs.lst` to obtain a list of all modules. `LEP_x_Build.vi` generates `prjObjs.lst` and lists all binary modules and libraries.

`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.

**LEP_x_ScriptLinker.vi Implementation Example**

The VxWorks CMD565 example target implements a subtarget-specific `LEP_vxworks_cmd565_ScriptLinker.vi`, which is located in the following file:

```
labview\Targets\NI\Embedded\vworks\cmd565\vworks_cmd565_LEP_TargetPlugin\LEP_vxworks_cmd565_Build.vi
```

This custom implementation overrides the default implementation of the `LEP_vxworks_ScriptCompiler.vi`, `LEP_vxworks_ScriptLinker.vi`, and `LEP_vxworks_Config.ctl`. 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.

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_vxworks_cmd565_ScriptLinker.vi` also must link with the run-time library intermediate files.
LVDefs_plat.h

LVDefs_plat.h contains basic constants and macros the LabVIEW C Code Generator needs to generate C code. This file provides the mapping between generic function calls the LabVIEW C Generator generates to OS-specific run-time functions. It also contains platform defines for data types and feature flags that define which features are implemented for an OS.

Each OS has a separate LVDefs_plat.h file, so if your OS matches one of the example operating systems, you do not need to create this file. LVDefs_plat.h is located in OS-specific folders in the following directory:

```
labview\CCodeGen\include\os\``

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
- LV_PLATFORM_FINI

You define these macros 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 an eCos target on the CMD565 does not.

If you define LV_PLATFORM_INIT as a function, LV_PLATFORM_INIT is called before calling any VI and before performing any other setup. 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.
lvEmbeddedMain.c is located in the following directory:

labview\CCodeGen\libsrc\main

LVSysIncludes.h

LVSysIncludes.h is a list of header files for a target OS. Depending on the target OS, LVSysIncludes.h contains OS-specific header files, toolchain-specific header files, and standard C header files. All but a few files in the labview\CCodeGen directory include this file.

Each OS has a separate LVSysIncludes.h file, so if your OS matches one of the example operating systems, you do not need to create this file. LVSysIncludes.h is located in OS-specific folders in the following directory:

labview\CCodeGen\include\os\n
TgtSupp.xml

TgtSupp.xml is the file LabVIEW uses to integrate your target into the LabVIEW development environment. This target-specific file defines the target name, icons, and which plug-in VIs implement the basic user actions and menu items. Use the Target Editor to generate and modify the TgtSupp.xml file.

Refer to Chapter 7, Creating a New Embedded Target, for more information about using the Target Editor.

TargetSyntax.xml

TargetSyntax.xml is the file LabVIEW uses to configure target syntax, which determines the supported and unsupported features for a target. Use the Configure Target Syntax dialog box to create and modify the TargetSyntax.xml file.

Refer to Chapter 8, Configuring the Target Syntax, for more information about configuring the target syntax for your target.
LabVIEW C Code Run-Time and Analysis Libraries

The Microprocessor SDK includes the source code for the LabVIEW C Code Run-Time Library and the Analysis Library.

Your target directory must include the LabVIEW C Code Run-Time and Analysis Libraries. Most of the example targets use a makefile to build the libraries. The Windows Console example target uses a Visual Studio project. Some example targets have prebuilt libraries while some targets build the libraries when you build an embedded VI into an embedded target.

If you copy and rename an existing embedded target, LabVIEW relinks the LabVIEW C Code Run-Time and Analysis Libraries for your target.

LabVIEW C Code Run-Time Library

Each LabVIEW VI and function that does not map directly to a C primitive is implemented using a function call. The LabVIEW C Code Run-Time Library implements these functions on top of a lower layer of generic functions that handle memory management, string manipulation, timing, and so on.

One of the major components to porting the LabVIEW C Code Run-Time Library is the mapping of these generic, low-level function calls to functions that your target run-time environment supports. Your toolchain needs this mapping, which you provide in LVDefs_plat.h, to compile the generated C code for your target.

Depending on your target, the operating system, and the example target, you might need to modify LVDefs_plat.h with #if #else statements to account for your target. Exactly what you change is very target-specific, but alignment and endianess are the two most common #define changes.

The Microprocessor SDK includes the source code for the LabVIEW C Code Run-Time Library, which is located in the following directory:

labview\CCodeGen\libsrc
Chapter 4  LabVIEW C Code Run-Time and Analysis Libraries

If you are porting LabVIEW to a new operating system, you must implement the necessary OS-specific components and be familiar with the LabVIEW C Code Run-Time Library #defines and feature flags.

Refer to Chapter 13, Porting the LabVIEW C Code Run-Time Library to a New Platform, for more information about porting the LabVIEW C Code Run-Time Library to a new platform.

Analysis Library

To port the LabVIEW Analysis Library, you must have a C++ compiler that supports the C++ Standard Template Library (STL). Also, your target must support floating-point operations either natively or though an emulator library to support the LabVIEW 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 LabVIEW 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 Microprocessor SDK 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. Parts of the LabVIEW Analysis Library depend on BLAS/LAPACK, so define COMPILE_FOR_SMALL_RT to remove BLAS/LAPACK support. Most of the example targets define COMPILE_FOR_SMALL_RT in the makefile used to build the libraries. The Windows Console example target uses a Visual Studio project.

The following directory contains the LabVIEW Analysis Library source code to convert LabVIEW data types to C data types:

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:

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

**Note** The LabVIEW Analysis Library has some thread-safe guards that require more support functions, but thread-safe guards are not implemented.

The LabVIEW Analysis Library relies on LabVIEW.lib, which is another library that you must build. LabVIEW.lib provides an interface between the LabVIEW C Code Run-Time Library and the LabVIEW 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.

```c
labview\CCodeGen\libsrc\lvanalysis\arrresize.c
```
Defining Code Generation Options

The LabVIEW C Code Generator, which is a component of the Microprocessor SDK, 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 Microprocessor SDK 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.

Code generation options define how the LabVIEW C Code Generator generates C code. 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 when a user builds an embedded VI into an embedded application.

- **LVForms.h** contains declarations for the entry point functions for each VI.
- **LVGlobs.h** contains declarations for global initialization functions and data type information used throughout the embedded application.
- **LVISRList.h** contains a list of Interrupt Service Routines (ISRs).
- **LVFuncsUsed.h** contains usage information to support source-level dead stripping.
- **LVDebugTable.h** contains lookup information for serial, TCP/IP, and CAN debugging.

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 value.

Many attributes determine how LabVIEW generates the C code. Most of the attributes are optimizations that make the generated C code smaller and run faster. The generated C code might differ in behavior from VIs you create under the My Computer target, which runs on Windows.

Refer to the *Code Generation Attributes* section for information about the different attributes.

**Note** This VI Server call is licensed. The user must have a valid, activated Microprocessor SDK 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 `LEP_x_CGen.vi`, determine how the LabVIEW C Code Generator generates the C code. `LEP_x_CGen.vi` is located in your target plug-in directory.

**BigEndian**

**Type:** Boolean

**Default:** FALSE
Chapter 5  Defining Code Generation Options

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 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. If the **IncrementalBuild** attribute is FALSE, LabVIEW overwrites any C files in the destination directory with the same name as VIs in the embedded VI.

**ExpressionFolding**

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

Generates better performing and more efficient code by collapsing groups of nodes into single expressions that most C compilers recognize. You cannot debug an embedded VI while using expression folding because expression folding eliminates some of the wires in the generated code. If you overrun the bounds of an array or divide by zero, the embedded application might crash.

Refer to the **Expression Folding Patterns** section for a list of the patterns the LabVIEW C Code Generator recognizes.

**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 of 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 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 necessary extra code to debug over serial, Ethernet, or CAN protocols if you set this attribute to TRUE. The extra code contains function calls that update the application 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 for your target. For example, the Format Into String function in LabVIEW is implemented completely in the LabVIEW C Code Run-Time Library to support LabVIEW data types. However, if you use only integer, floating-point, or Boolean data, you can use the C library function `printf` instead. Set this attribute to TRUE to use `printf` when possible. The LabVIEW C Code Generator uses the LabVIEW C Code Run-Time library implementation if you cannot use `printf`. The `printf` function is usually smaller and faster than the LabVIEW C Code Run-Time Library 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 you compile with the _Integer_Only flag set to produce applications that run without hardware or software floating-point support.

**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, which are clusters that contain only scalars. For example, you cannot use a cluster that contains a string.

The LabVIEW C Code Generator generates a C 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, which runs on Windows, by
using default data if any input or output is null. 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, which runs on 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 you set the priority of an individual VI to subroutine on the Execution page on the VI Properties dialog box, the LabVIEW C Code Generator generates C code 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
Chapter 5  Defining Code Generation Options

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 all files the LabVIEW C Code Generator generates 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 LabVIEW C Code Generator rebuilds and overwrites the C file.

Use incremental builds if you want to write and combine large applications with a makefile build process that does not rebuild everything every time a minor change is made.

**Note** If the VIs are in a .llb the LabVIEW C Code Generator regenerates all of the VIs in the .llb 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 Microprocessor SDK does not support the ability to set a flag in a VI indicating that the VI is an ISR VI. You must implement support for ISRs.

Refer to Chapter 14, Implementing Interrupt Service Routine Support, for more information about implementing ISR support.
MemoryModel

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

Default: 0

Defines whether you are using a dynamic or static memory model.

- **Dynamic**—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 dynamic memory model have the smallest footprint possible.

- **Static**—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 static memory models might require more memory than other memory models.

Refer to Chapter 22, *Static Memory Allocation*, for more information about supporting a static memory model.

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.

Refer to Chapter 18, *On Chip Debugging*, for more information about implementing OCDI debugging.

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 can result 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

Code generation options define how the LabVIEW C Code Generator generates C code. Many attributes determine how LabVIEW generates the C code. Most of the attributes are optimizations that make the generated C code smaller and run faster. Use the following code generation options to generate the fastest code:

- GenerateGuardCode=FALSE
- GenerateSerialOnly=TRUE
- UseStackVariables=TRUE
- GenerateCFunctionCalls=TRUE
Expression Folding Patterns

The following is a list of patterns that LabVIEW recognizes when you build an embedded VI into an application.

Pattern 1

Takes the pattern `reference count-array index-primitive-array-replace` and replaces it with a single array expression.
Pattern 2

Takes the pattern \textit{left shift register-reference count-array index-right shift register} and eliminates the reference count.

Pattern 3

Takes the pattern \textit{global-reference count-array index-primitive-array replace-global} and replaces it with an array expression directly on the global variable.
Pattern 4

Takes the pattern `global-reference count-array index-global` and replaces it with an array index directly on the global variable.

Pattern 5

Takes the pattern `global-reference count-array replace-global` and replaces it with an array replace directly on the global variable.

Pattern 6

Folds function expressions for unary and binary operations into a single line of code and eliminates heap or stack variables. LabVIEW treats Quotient & Remainder functions with integer inputs and only one output wired as a binary operation for this pattern. If the output wire of a function branches, LabVIEW cannot include the function in the folded expression because the intermediate value is needed.
You must turn off the guard code generation option for this pattern to work.

**Pattern 7**

Folds expressions into a select statement and eliminates heap or stack variables.
Pattern 8

Takes the pattern `reference count-unbundle-bundle` and eliminates reference counts and copies. This pattern works only if the elements unbundled and bundled are exactly the same.

![Pattern Diagram](image)

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 `VarNames.vi`, located in the `labview\CCodeGen` 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 Example</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><img src="image" alt="Motor On" /></td>
<td>Motor_On</td>
<td>/* Motor On */</td>
</tr>
<tr>
<td>Constant</td>
<td>Label</td>
<td><img src="image" alt="Trigger Level" /></td>
<td>Trigger_Level</td>
<td>/* Trigger Level */</td>
</tr>
<tr>
<td>Function</td>
<td>Label, terminal name</td>
<td><img src="image" alt="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><img src="image" alt="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><img src="image" alt="Relay_CS" /></td>
<td>Relay_CS</td>
<td>/* Relay: Case Selector */</td>
</tr>
<tr>
<td>Case tunnel</td>
<td>Structure Label, “CT”</td>
<td><img src="image" alt="My_Structure_CT" /></td>
<td>My_Structure_CT</td>
<td>/* My Structure: CT */</td>
</tr>
</tbody>
</table>
### LabVIEW Signal Source

| Shift register (initialized) | Source name, “SR” | Init_SR, Init_SR_1 | /* Init: SR */, /* Init: SR 1 */ |
| Shift register (uninitialized) | Source name, “SR” | Foo_Loop_SR, Foo_Loop_SR_1, Foo_Loop_SR_2 | /* Foo Loop: SR */, /* Foo Loop: SR 1 */, /* Foo Loop: SR 2 */ |
| Sequence Local | Source name, “SL” | Init_SL | /* Init: Sequence local */ |
| Sequence tunnel | Source name, “ST” | Level_Y_ST | /* Level Y: ST */ |
| Structure terminal | Label, terminal name | Wave_Loop_i | /* Wave Loop: i */ |
### 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.
Defining Compiler Directives

Macros in LVDefs_plat.h define how to compile generated and external C code. LEP_x_ScriptCompiler.vi scripts the compiler and passes the directives to the compiler from the command line.

This section lists some of the more common macros. Refer to the comments in LVDefs_plat.h for your OS for additional macros.

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

Debugging Macros

The following list describes the debugging macros.

- **UsesSerialDebugger**—Define to enable debugging over a serial line.
- **DBGBAUD**—Specify the baud rate of the serial line for serial debugging.
- **UsesTCPDebugger**—Define to enable debugging over a TCP/IP connection.
- **UsesCANDebugger**—Define to enable debugging over a CAN bus.
- **OCDI_RDM**—Define to enable the Release Debug Mode for on chip debugging.
- **BDTUseThread**—Set to 1 to enable the Background Debug Thread (BDT). The default is 1, which enables BDT, for targets that support threading and 0 for targets that do not support threading.
- **BDTSleepTime**—Specify the time in milliseconds that the BTD sleeps after updating a front panel indicator or a probe. The default is 50. Increase the time for targets with very slow communication channels and decrease the time if you want better interactivity with the front panel updating during a debugging session.
- **BDTReconnectTryTime**—Specify the time interval in milliseconds that the target tries to reconnect to the host computer if disconnected. The default value is 5000.
• **BDTHostConnectTime**—Specify the time in milliseconds to wait before the host connects to the target if you build the application with instrumented debugging. If the host computer does not connect to the target in the time you specify, the embedded application starts running. The default is 5000.

### Feature Macros

The following list describes the feature macros.

- **CANSupport**—Set to 1 to enable VxWorks CAN support. To use the Embedded CAN VIs with your target, you must obtain a driver and libraries for the board from Wind River and compile it into your VxWorks image.
- **SocketSupport**—Set to 1 to enable BSD (Berkeley Software Distribution) socket support.
- **_Include_OSScheduler**—Set to 1 to enable multithreading support. The LabVIEW C Code Generator usually sets this macro.
- **HeapListSupport**—Set to 1 to enable heap support.
- **BluetoothSupport**—Set to 1 to enable Bluetooth support.
- **FileSupport**—Set to 1 to enable file I/O support.
- **IrDA_Support**—Set to 1 to enable IrDA support.
- **SerialSupport**—Set to 1 to enable serial support.
- **DatalogSupport**—Set to 1 to enable datalogging support.
- **TCPUDPSupport**—Set to 1 to enable TCP/UDP support.

### Memory Mapping Macros

The following list describes the memory mapping macros.

- **_DATA_SECTION**—Defines the input memory section for array structures.
- **_TEXT_SECTION**—Defines the input memory section for generated code.
General Macros

The following list describes the general macros.

- CHeadless—Define this macro unless you implement a user interface for your target. The Unicon UCN2410-CWIFI and the Unix UI example targets are the only example targets that implement a user interface.
- _Integer_Only—Define for integer-only targets.
- CStatic—Define for the static memory model.

Operating System Macros

The following list describes the operating system macros.

- Win32—Define for all Windows targets.
- _vxworks—Define for all VxWorks targets except VxWorks Simulation targets.
- _vxworksim—Define for all VxWorks Simulation targets.
- _ECOS—Define for all eCos targets.
- linux—Define for all Linux targets.
- dsp_bios—Define for all DSP Bios (TMS320) targets.
Creating a New Embedded Target

Use the Target Editor to create new embedded targets, modify existing targets, delete targets, and so on. You use the Target Editor to create and modify TgtSupp.xml, which is the file LabVIEW uses to integrate your target into the LabVIEW development environment.

You can create new embedded targets from scratch or by copying an existing example target.

Tip National Instruments recommends copying an existing target. Look for an example target that is similar to your target. If you cannot find a similar example target, use an example target with a similar OS, if possible. Use the Target Editor to explore Microprocessor SDK example target implementations.

Copying and Renaming an Existing Target

Complete the following steps to create a new embedded target by copying and renaming an existing embedded target. You also can create a new embedded target from scratch if none of the example targets are appropriate.

1. 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.

2. Create an Embedded subfolder. LabVIEW does not recognize any targets not in an Embedded folder.
3. Create a subfolder with your target name.
4. Launch LabVIEW.
5. Select Tools»Microprocessor SDK»Target Editor from the Getting Started window, the Project Explorer window, or a blank VI to open the Target Editor.
6. Select File→Open in the Target Editor and select the TgtSupp.xml file for the example target you are copying. TgtSupp.xml files are located in the target directory. For example, the TgtSupp.xml file for the Unix Console example target is located in the labview\Targets\NI\Embedded\unix directory.

7. In the Target Editor, right-click the target you selected in the previous step, select Rename from the shortcut menu, and enter your target name.

8. Select File→Save As and navigate to the directory you created in step 3 to open the Save Target As dialog box.

9. (Optional) LabVIEW prepends prefixes to the plug-in directory and VIs themselves. Edit the prefix for your target in the Replace with text boxes.

10. Click the OK button. LabVIEW copies, renames, and relinks the files for your new target. The copying, renaming, and linking can take a few minutes. Click the OK button to close the dialog box.

**Creating a New Target from Scratch**

Complete the following steps to create a new embedded target from scratch. National Instruments recommends copying and renaming an existing example target if possible.

1. 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.

2. Create an Embedded subfolder. LabVIEW does not recognize any targets not in an Embedded folder.

3. Create an OS subfolder. For example, all of the example targets that run eCos are in the labview\Targets\NI\Embedded\ecos directory.

4. Open your OS subfolder and create a subfolder with your target name.

5. Launch LabVIEW.

6. Select Tools→Microprocessor SDK→Target Editor from the Getting Started window, the Project Explorer window, or a blank VI to open the Target Editor.

7. Select File→New in the Target Editor.
8. Navigate to and select your Embedded folder.
9. Right-click the Embedded folder in the Target Editor and select New»Folder from the shortcut menu. A New Folder folder appears under the Embedded folder.
10. Right-click New Folder and select Rename from the shortcut menu to rename the folder. Rename the folder to match the OS name you created in Windows Explorer.
11. Right-click your OS folder in the Target Editor and select New»Target from the shortcut menu. A New Target item appears under the New Folder folder.
12. Right-click New Target and select Rename from the shortcut menu to rename the target.
13. Right-click your renamed target and select New»Build Specification Type.
14. Right-click New Build Specification Type and select Rename from the shortcut menu to rename the build specification type.

Tip Look at the example targets for examples of various build specification type names.

15. Select File»Save As in the Target Editor and select your target folder to open the Save Target As dialog box.
16. (Optional) LabVIEW prepends prefixes to the plug-in directory and VIs. Edit the prefix for your target in the Replace with text boxes.
17. Click the OK button. LabVIEW creates your TgtSupp.xml file, which is the file LabVIEW uses to integrate your target into the LabVIEW development environment. This can take a few minutes. Click the OK button to close the dialog box.

Modifying an Existing Target

Complete the following steps to modify an existing embedded target.

1. Select Tools»Microprocessor SDK»Target Editor from the Getting Started window, the Project Explorer window, or a blank VI to open the Target Editor.
2. Select File»Open in the Target Editor and select the TgtSupp.xml file for the example target you are copying. TgtSupp.xml files are located in the target directory. For example, the TgtSupp.xml file for the Unix Console example target is located in the labview\Targets\NI\Embedded\unix directory.
3. 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.

4. 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 Specification Type Properties** dialog box to configure the build specification type properties.

5. Select **File»Save** in the Target Editor to save your changes.

### Configuring Target Properties

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 **Enable Set Dimension Size for arrays (Static Memory Allocation)** checkbox if you want to enable fixed-size arrays and preallocate the 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. (Optional) Place a checkmark in the **Enable shared variable support** to enable the Shared Variable Settings page.

3. (Optional) Remove the checkmark from the **Prompt user to create build specification** if you want LabVIEW to automatically create a build specification using the build settings you specify in your default build specification type.

4. Select the tick resolution in Timed Loops as microseconds or milliseconds from the **Timed Loop internal clock resolution** pull-down menu.
5. Navigate to and select or enter the relative path to the target icon file, which is the icon that appears in the Project Explorer window for your target, in the **Project Explorer window icon** text box. Icons must be in .png format. Icons are located in the following directory:

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

6. Navigate to and select or enter the relative path to the directory that contains the TargetInfo.ini file, which is the file that defines the target-specific palette and controls, 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 dynamic memory allocation default configuration, which is located in the following directory:

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

   The static memory allocation default configuration is located in the following directory:

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

7. (Optional) Navigate to and select 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.

8. (Optional) Navigate to and select or enter the relative path to the Elemental I/O configuration file (eio.xml). Leave the **Path to Elemental I/O directory** text box blank if you are not implementing Elemental I/O.

9. (Optional) Navigate to and select or enter the relative path to the directory that contains your Funclist files. Leave the **Path to the Funclist directory** blank if you are not using Funclist files to override subVI calls with C function calls.
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 Item or 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 shortcut menu item name in the Item Name text box.
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.
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.

Refer to the Creating New Plug-In or Subpanel VIs section for more information about creating plug-in VIs.

Defining Target Properties 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.

1. Click the blue plus button, shown at left, to enable the Page Properties section. ??? appears in the Category list.
2. Enter the name of the category in the Item Name text box. The category name appears in the Category list in the Target Properties dialog box for your target.
3. Navigate to and select or enter the path to the subpanel VI in the Path to Subpanel VI 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.

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 Sources section. ?? appears in the Timing Sources 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 Path to OnRelease Plug-In VI 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 plug-in VIs.

Defining Signed Files

Use the Signed Files tab in the Target Properties dialog box to define which plug-in VIs and files 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 licensing information National Instruments sends you if you are reselling your target.

1. Enter your OEM-specific or VAR-specific feature name in the Component Name text box. National Instruments assigns this name. The name matches the name in the OEM-specific or VAR-specific license file.

2. Enter the OEM-specific or VAR-specific feature version in the Component Version text box. National Instruments assigns this version and the version matches the version in the OEM-specific or VAR-specific license file.

3. Enter the OEM-specific or VAR-specific name used to activate the target in the Package Name text box.

4. Enter the OEM-specific or VAR-specific version used to activate the target in the Package Version text box.

5. (Optional) Select the Enable the Evaluate Button button if you want users to be able to evaluate your target in the license activation dialog box.

6. Enter the text you want to display in the title bar of the license activation dialog box in the Dialog Box Title text box.

7. Enter the URL for the Web page on ni.com where users can buy your target in the URL for Purchasing Web Page text box.

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 Specification Type** checkbox to indicate that the build specification you are creating 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 your target’s **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. 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 identify the build specification type. Each target build specification type you create 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 right-clicks **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 Item or 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 shortcut menu item name in the Item Name text box.
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 of the category in the Item Name text box. The category name appears in the Category list in the Build Specification Properties dialog box for your target.
3. Navigate to and select or enter the path to the subpanel VI in the Path to Subpanel VI 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.
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 callback VIs for various button actions.

1. In the Path to OnBuild Plug-In VI text box, navigate to or enter the relative path to the callback VI to call 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 to call 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 to call 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 to call 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 create 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.
   - Select VI with test dialog to create a new plug-in VI that includes the One Button Dialog function for testing purposes.
   - 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.
   - Select Subpanel VI with test controls to create a new subpanel VI that includes some front panel controls for testing purposes.
   - 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.
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. Target syntax also prevents you or other users from importing unsupported VIs from other targets.

Select Tools→Microprocessor SDK→Configure Target Syntax to open the Configure Target Syntax dialog box. Use the Configure Target Syntax dialog box to create the TargetSyntax.xml file, which is the file LabVIEW uses to check the target syntax.

⚠️ **Caution** This process overwrites the default TargetSyntax.xml file, 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 in labview\Targets. Features that the LabVIEW C Generator does not support are already included in the VIs that determine the target syntax.
   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 a 32-bit signed integer numeric, 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 subVIs 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 also can create a VI that contains only the features that the target can support. Name supported functionality VIs LVESupportedxx.vi where xx follows the naming convention of the unsupported VIs.

3. Select Tools»Microprocessor SDK»Configure Target Syntax 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 the `labview\vi.lib` directory.

   a. Navigate to and select `LVEUnsupportedVIs.vi`, which is the VI that contains the unsupported data types.

   b. (Optional) Navigate to and select the VI that contains the supported subVIs.

   c. Place a checkmark in the **Unspecified nodes are unsupported** checkbox if you are using both unsupported and supported VIs and you want subVIs to be unsupported if the subVIs are not in either VI.

8. On the **Default Type** tab, 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.

   a. Enter a maximum recursion depth for supported data types in the **Max Type Depth** box. 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.

      Setting 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.

   b. Enter the maximum number of dimensions for arrays in the **Max Array Dimensions** box. A value of -1 ignores the maximum array dimension.

      Setting the **Max Array Dimensions** to 1 to restrict users to only optimized 1D array operations, which can increase application performance.

   c. Place a checkmark in the **Support global variables** checkbox if the target supports global variables.
d. Place a checkmark in the **Support local variables** checkbox if the target supports local variables.

e. Select the **Errors break the VI** radio button if you want target syntax errors to break the VI.

f. Select the **Errors are considered warnings and do not break the VI** radio button if you want LabVIEW to consider target syntax errors as warnings rather than errors.

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.

12. Click the **Save** button to save the TargetSyntax.xml file. You must save the file as TargetSyntax.xml for LabVIEW to recognize the target syntax for your target.

**Note**  Clicking the **Generate** button does not automatically save the TargetSyntax.xml file. You must click the **Save** button to save the file.

### Modifying the Target Syntax for an Embedded Target

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»Microprocessor SDK»Configure Target Syntax** to open the **Configure Target Syntax** dialog box.

3. Click the **Load** button to load an existing TargetSyntax.xml file.

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.

7. Click the **Save** button to save the modified TargetSyntax.xml file.

**Note**  You must regenerate and save the TargetSyntax.xml file if you modify any of the unsupported or supported VIs.
Implementing Serial or TCP Instrumented Debugging

You can use instrumented debugging with LabVIEW to communicate with an embedded application running on an embedded target to 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. If you are using serial or TCP communication, you do not have to implement this communication layer. Instead, you can use the existing communication layer the Microprocessor SDK example targets use. If you are using a communication protocol other than serial or TCP, you must implement your own communication layer.

Refer to Chapter 17, Implementing Instrumented Debugging for a Custom Communication Layer, for more information about implementing instrumented debugging for a custom communication layer.

Serial Communication

If you are using serial communication, your LEP_x_Connect.vi and LEP_x_Debug.vi plug-in VIs must call the nitargetStartSerialDebug VI. Your plug-in VIs are located in your target directory. The nitargetStartSerialDebug VI is located on the Debugging Utilities palette and in the following directory:

labview\vi.lib\LabVIEW Targets\TargetShared
TCP Communication

If you are using TCP communication, your LEP_x_Connect.vi and LEP_x_Debug.vi plug-in VIs must call the nitargetStartTCPDebug VI. Your plug-in VIs are located in your target directory. The nitargetStartTCPDebug VI is located on the Debugging Utilities palette and in the following directory:

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

You must provide the TCP libraries for your target. If a Linux target and the host computer are different devices, 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 the host computer are the same machine, define TCP_USE_LOCALHOST as 1.

**Note** If you are implementing OCDI debugging or using another communication layer for instrumented debugging, such as CAN or RTDX, you must implement the communication layer for LabVIEW and the communication layer for your target.

Log Files for Troubleshooting

LabVIEW logs all of its actions, such as loading plug-in VIs, if you create a log file, Embedded.log, and place the file in the labview directory before launching LabVIEW. This log file is useful for debugging porting errors because you can quickly locate the plug-in VIs causing the errors. You cannot specify which actions LabVIEW logs.

You can configure LabVIEW to create an embedded debugging diagnostic file you can use to troubleshoot instrumented debugging. To create this diagnostic file, add the line LogCGenMessages=True to the LabVIEW.ini file, which is located in the labview directory. LabVIEW creates a file in the labview directory named cgenlog.txt. This text file 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 your instrumented debugging implementation.
Testing Your Target Implementation

The LabVIEW Microprocessor SDK includes the LabVIEW C Generator Test Framework, which is an automated test framework, and C Generator-specific VI Analyzer tests.

The LabVIEW C Generator Test Framework tests the LabVIEW C Code Run-Time Library. Use the automated tests to verify successful target implementations.

The C Generator-specific VI Analyzer tests can assist you with developing embedded VIs. Use the C Generator-specific VI Analyzer tests to help optimize embedded applications for size and performance.

LabVIEW C Generator Test Framework

Select **Tools** > **Microprocessor SDK** > **Run C Generator Tests** to run the automated tests in the LabVIEW C Generator Test Framework. You can select to run all tests for a target, all tests for all targets, only some of the tests, and so on. Multiple tests for an embedded target execute in the order you select the tests.

The tests are located in the following directory:

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

The LabVIEW C Generator Test Framework executes all the tests in a folder you specify in the LEP_AutoTest.ini file, which is located in the labview\autotest directory. When you run the tests, LabVIEW creates a folder with a date and time label and saves the test results as text files in that directory.

**Tip**  You can specify where LabVIEW creates the results directory in the LEP_AutoTest.ini file. By default, LabVIEW creates the results directory in the labview\autotest\results\win32con directory.
After LabVIEW runs all of the tests and saves the results, LabVIEW creates an `index.html` file that summarizes the test results. The summary file lists the test name and whether the test passed or failed. If the test failed, the `index.html` file provides a hyperlink to the test results text file that contains the errors.

**Tip** You can configure the LabVIEW C Generator Test Framework to email failed test results to the email addresses you specify in the `[SendMail]` section of the `LEP_AutoTest.ini` file.

Target-specific sequence files support text execution on actual hardware. You might need to implement target-specific sequence files to support the execution of tests on actual hardware. Refer to the `win32con_Target.ini` and `ecos_cmd565_Target.ini` sequence files for implementation examples. The example sequence files are located in the following directory:

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

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

### Default Tests

The LabVIEW C Generator Test Framework includes a default set of tests to help you verify your target implementation. 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>
</tbody>
</table>
Adding Your Target to the LabVIEW C Generator Test Framework

Add your target to the LEP_AutoTest.ini file, which lists the targets to test and notification email addresses for failed tests, to include the target in the LabVIEW C Generator Test Framework dialog box. The order of the targets in the file determine the order of the targets in the dialog box.

LEP_AutoTest.ini is located in the following directory:

```
labview\autotest
```

The format of the LEP_AutoTest.ini is as follows:

```
[StartTests]
StartTestsEnable=false
ReportFolder=xxx
ReportFileName=xxx
[Embedded:folder:Target Name]
  WorkFolder=xxx
  TestFolder=xxx
```
The following table lists the INI tokens.

<table>
<thead>
<tr>
<th>Token</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StartTestsEnable</td>
<td>You can configure the LabVIEW C Generator Test Framework to automatically begin executing after 10 seconds unless you click anywhere in the LabVIEW C Generator Test Framework dialog box. Automatic test execution is useful for unattended or automated testing. To enable this feature, change this token to true. The default is false.</td>
</tr>
<tr>
<td>ReportFolder</td>
<td>Specifies the path to a summary report that lists which targets have been tested, the number of failed tests per target, and the number of memory leaks per target.</td>
</tr>
<tr>
<td>ReportFileName</td>
<td>Specifies the filename for the summary report.</td>
</tr>
<tr>
<td>[Embedded:folder: Target Name]</td>
<td>Specifies the folder target name, which is called the target URL. The target URL is located in the TgtSupp.xml file for your target. You must have at least one folder.</td>
</tr>
<tr>
<td>WorkFolder</td>
<td>Specifies the path to the test results for your target. You can use relative paths, which are relative to the labview\autotest directory. National Instruments recommends placing your tests in the autotest directory, if possible. The LabVIEW C Generator Test Framework creates a subfolder that uses the start date and time when the tests start executing. After the tests run, the framework generates an index.html file, which contains the test results, in the same subfolder.</td>
</tr>
<tr>
<td>TestFolder</td>
<td>Specifies the path to the tests for your target. You can use relative paths, which are relative to the labview\autotest directory. National Instruments recommends placing your tests in the autotest directory, if possible.</td>
</tr>
</tbody>
</table>

**Emailing Failed Test Results**

(Optional) Enable email notification for failed tests in the [SendMail] section of the LEP_AutoTest.ini file.

The format of the [SendMail] section of the LEP_AutoTest.ini file is as follows:

```
[SendMail]
SendMailEnable=
MailServer=
ReturnAddress=
Recipient0=
```

Microprocessor SDK Porting Guide 10-4 ni.com
The following table lists the INI tokens.

<table>
<thead>
<tr>
<th>Token</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SendMailEnable</td>
<td>Set this token to true to enable email notification. The default is false.</td>
</tr>
<tr>
<td>MailServer</td>
<td>Specifies the name or IP address of the SMTP server.</td>
</tr>
<tr>
<td>ReturnAddress</td>
<td>Email address from which the email notification is sent.</td>
</tr>
<tr>
<td>Recipient0</td>
<td>The email address to which the email notification is sent. Enumerate additional recipient email addresses on separate lines. For example: Recipient0=<a href="mailto:Bob@company.com">Bob@company.com</a> Recipient1=<a href="mailto:Laura@company.com">Laura@company.com</a></td>
</tr>
</tbody>
</table>

After you enable the email notification in LEP_AutoTest.ini, place a checkmark in the Send Mail checkbox in the LabVIEW C Generator Test Framework dialog box to email the name of failed tests and the path to the relevant index.html file.

### Creating Configuration Files

In addition to the LEP_AutoTest.ini file, additional .ini files describe the tests for different targets. You must create configuration .ini files for your target after you add the target to LEP_AutoTest.ini. You create separate configuration .ini files for each test subfolder your target uses. Each section of the configuration .ini file is a test for the target. Each section must have a unique name to appear in the LabVIEW C Generator Test Framework dialog box. Duplicate sections are not shown.

National Instruments recommends that you organize your tests in subfolders based on the functionality of the tests. For example, consider putting all tests for strings in the same subfolder. Use LEP_AutoTest_GetNameIniFile.vi, located in the labview\autotest directory, to generate the name of your target configuration files. The target name in the Target URL text box and in the LEP_AutoTest.ini file must match exactly. When you run the VI, the LabVIEW C Generator Test Framework generates the name for the configuration files in the ini filename text box.
Default Configuration Files

You can specify a default configuration file for each target. The default configuration file is located in the TestFolder path in the LEP_Autotest.ini file. You must append DefaultConfiguration to the configuration filename. For example, the default configuration file for the Windows Console example target is win32con_Target_DefaultConfiguration.ini. Tokens in the default configuration file override the tokens in an individual configuration file.

Autotest Implementation Example

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

labview\autotest\tests\Large semi-auto tests\arr-dr

The format of the ini file is as follows:

```
[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
_Tes_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>Token</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[arr_drv_Test_Debug]</td>
<td>The name of the test. You must enclose test names in brackets. The test name 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 LEP_AutoTest_TestSequence_1.vi.</td>
</tr>
</tbody>
</table>
C Generator-Specific VI Analyzer Tests

The Microprocessor SDK includes VI Analyzer tests for C code generation options and performance.

Select Tools » VI Analyzer » Analyze VIs to open the VI Analyzer. Refer to the test descriptions in the VI Analyzer for more information about the tests.

Note You have access only to the C Generator-specific tests unless you install and activate the LabVIEW VI Analyzer Toolkit.

Log Files for Troubleshooting

LabVIEW logs all of its actions, such as loading plug-in VIs, if you create a log file, Embedded.log, and place the file in the labview directory before launching LabVIEW. This log file is useful for debugging porting errors because you can quickly locate the plug-in VIs causing the errors. You cannot specify which actions LabVIEW logs.

You can configure LabVIEW to create an embedded debugging diagnostic file you can use to troubleshoot instrumented debugging. To create this diagnostic file, add the line LogCGenMessages= True to the LabVIEW.ini file, which is located in the labview directory. LabVIEW

<table>
<thead>
<tr>
<th>Token</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_Test_BaudRate_</td>
<td>The baud rate of the capturing serial port. Only test sequences that capture output on the serial port use this token.</td>
</tr>
<tr>
<td>s_Test_ResourceName_</td>
<td>The name of the capturing serial port. Only test sequences that capture output on the serial port use this token.</td>
</tr>
<tr>
<td><em>Test_Timeout</em></td>
<td>The time interval in milliseconds to obtain the result before the test fails.</td>
</tr>
<tr>
<td>iBuildConfiguration</td>
<td>Target-specific configuration tokens. Each target-specific configuration token is prepended with a letter that indicates the data type of the token.</td>
</tr>
<tr>
<td>bGenerateDebugInfo</td>
<td>b = Boolean value</td>
</tr>
<tr>
<td>iMemoryModel</td>
<td>i = integer value</td>
</tr>
<tr>
<td>bGenerateSerialOnly</td>
<td>p = path value</td>
</tr>
<tr>
<td>bGenerateGuardCode</td>
<td>s = string value</td>
</tr>
<tr>
<td>bUseStackVariables</td>
<td></td>
</tr>
</tbody>
</table>
creates a file in the labview directory named \texttt{cgenlog.txt}. This text file 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 your instrumented debugging implementation.
Advanced Porting Implementations

Depending on your target, you might need to implement and port additional functionality. The additional pieces include the following:

- Implementing support for user interfaces
- Adding I/O through Elemental I/O
- Implementing ISRs
- Implementing timing
- Implementing memory mapping
- Porting the LabVIEW C Code Run-Time Library to a new platform
- Implementing instrumented debugging for a custom communication layer
- OCDI debugging
- Understanding the LVM file
- Integrating LabVIEW with another IDE
- Overriding subVI calls to improve performance
- Implementing static memory allocation
- Implementing a target server
Implementing User Interface Support

If your target has an LCD, you can implement user interface support. The Unix UI and Unicon example targets implement user interface support. The Unix UI example target is similar to the Unix Console example target.

The example user interface support uses FLTK, which is an open source C++ graphical user interface toolkit. You must develop your own video driver if you want to port user interface support and use something other than FLTK.

To implement user interface support, you must do the following:

1. Port the following three open source libraries to your target:
   - **Nano-X (version 0.91)**—low-level screen drawing functions
     You might have to develop a screen driver to work with your target. Also, Nano-X assumes you have a mouse, so if your target uses a touchscreen for input, you also have to develop a touchscreen driver that looks like a mouse to Nano-X. Refer to the Nano-X Web site at www.microwindows.org for more information about and documentation for Nano-X.
   - **nxLib (version 0.46)**—interface between FLTK and Nano-X
   - **FLTK (version 2.0.x-r5940)**—C++ graphics library
     FLTK assumes you have certain header files with certain functions. If these functions do not exist for your target, you must find a workaround. Refer to the FLTK Web site at www.fltk.org for more information about FLTK.

   The source code for these libraries are located in the following directory:

   `labview\CCodeGen\external`

2. After building the libraries, run and debug the libraries using your toolchain.
3. Build the LabVIEW C Code Run-Time Library and the fltkapi library, which is the interface between FLTK and the LabVIEW C Code Run-Time Library. The source code for the fltkapi library also is located in the labview\CCodeGen\external directory.

You must define the following flags when you build the LabVIEW C Code Run-Time Library:

- **GUISupport**—Enables user interface support.
- **FltkGUI**—Enables support for the FLTK widget library.
- **Supports3DControls**—Enables support for 3D controls.
- **SupportsColorBG**—Enables support for background color.
- **FP_SIZE_HORIZ**—Specifies the horizontal size in pixels of the target user interface.
- **FP_SIZE_Vert**—Specifies the vertical size in pixels of the target user interface.

4. Implement user interface settings in your Build Specification Properties dialog box.

**Limitations in Designing User Interfaces**

Designing user interfaces for an embedded target is different than designing user interfaces for Windows targets because of the following:

- Embedded targets with LCDs have a smaller screen and a significantly slower CPU than desktop computers.
- The input device generally is a pen, a stylus, or a finger, which makes inputting text slower on an embedded device. Use default values, if possible, to reduce the amount of text you must enter in an application.
- Embedded VIs support some front panel controls and indicators differently than LabVIEW for Windows.
- Embedded VIs do not support all types of controls and indicators.
- Some controls and indicators use native controls, so the controls might not look the same on the target as on the host computer.
- You cannot use the <Tab> key to navigate the controls or indicators or use keyboard shortcuts.
- You can hide controls and indicators in edit mode, but you cannot disable them. Use the Visible property to hide controls and indicators programmatically.
- Embedded VIs do not include toolbar icons or standard desktop menus, but you can create custom run-time menus.
Front Panel Controls and Indicators

Use the front panel controls and indicators located on the Controls palette to build the user interface. Designing user interfaces for an embedded target is different than designing user interfaces for Windows targets. In addition to some general design considerations, Microprocessor SDK does not support all types of controls and indicators. Microprocessor SDK supports the following front panel controls and indicators differently than VIs you create for Windows:

- Array controls and indicators
- Boolean controls
- Charts
- Cluster controls and indicators
- Combo boxes
- Decorations
- Enumerated type controls
- Graphs
- Listboxes
- Numeric controls and indicators
- Path controls
- Picture controls
- Refnums
- String controls and indicators
- Tab controls
- Time stamp controls and indicators
- Variant controls

Native Controls and Indicators

Most controls and indicators you can use in the VIs for the example targets are based on the FLTK native controls to save memory, so the controls might not look the same on the target as they do on the host computer. The following controls and indicators are native to FLTK:

- Button
- Checkbox
- Enumerated
- Listbox
Array Controls and Indicators

If you have an array of clusters containing arrays on the front panel window and you change the index of one of the arrays within the cluster, the index of the other arrays in the cluster does not change.

Boolean Controls

Boolean controls support the following:

- Labels
- Captions
- Boolean text
- Six types of mechanical action for Boolean controls, including Switch when pressed, Switch when released, Switch until released, Latch when pressed, Latch when released, and Latch until released.

Color in Boolean Controls

The following Boolean controls support color differently than LabVIEW for Windows:

- Toggle switches—Different colors for a toggle switch and the base of a toggle is unsupported. The color you select for the toggle switch is also the color of the toggle base regardless of the color you select for the base.
- Push buttons—Changing the background color of a push button is unsupported.
- Vertical and horizontal slide buttons—Changing the foreground color of a slide button is unsupported.
- Decorations—If you place a Boolean control on a decoration, the shadow of the Boolean control uses the background color of whatever the decoration is on instead of the decoration if the color is different.

Checkboxes

- You cannot resize checkbox controls.
Boolean Text

- The Square Button on the the Classic Boolean palette does not support Boolean text.

Charts

Charts support the following:
- Captions and labels
- Maximum of eight plots
- One horizontal scale on the bottom of the chart
- One vertical scale on the left side of the chart
- Numeric values, arrays of numeric values, and numeric clusters
- Autoscaling and visibility of the x- and y-axis and scale formatting
- Scale and cursor legends

Charts do not support the following:
- Duplicate scales
- Grids
- Plot legends

You cannot resize the x- or y-axis of a chart. The x-axis of a chart supports only integer numbers. Any floating-point numbers round to the nearest whole number.

If you use a Property Node to change the update mode of a chart, the x-axis is reset to start at zero.

Color

FLTK supports 32 bits per pixel color. LabVIEW maps the colors you use in LabVIEW to the colors the video driver supports.

Some Boolean controls support color differently than Windows.

Refer to the Boolean Controls section in this chapter for more information about using color with Boolean controls.
Cluster Controls and Indicators

If a control overlaps the cluster shell in a Windows VI, the part of the control outside of the cluster shell is clipped.

If a control overlaps the cluster shell in a Microprocessor SDK VI, the entire control is still completely visible when you build the VI into an application. If the control is completely outside the cluster shell, the control is not visible.

Combo Boxes

Combo boxes support the following:
- Captions
- Labels
- 16-bit signed integer numeric representation only

Combo boxes do not support the following:
- Digital displays
- Disabling individual items

Decorations

The following decorations are supported:
- Boxes
- Circles
- Horizontal Smooth Box
- Lines
- Triangles
- Vertical Smooth Box

Note  Raised and recessed decorations appear flat.

The following decorations are unsupported:
- Arrows
- Shadows
Enumerated Type Controls

Enumerated type controls support the following:

- Captions
- Labels
- Resizing and changing the color of the increment and decrement buttons

Enumerated type controls do not support the following:

- Digital displays
- Disabling individual items
- Text justification and underlining text

The text area of the enumerated type control is a string control on the target.

Selecting items in enumerated type controls is different in embedded applications:

- Click the control to display the shortcut menu so all options are visible.
- Tap the increment and decrement buttons to cycle through the options.

Fonts

When you build a VI into an application, LabVIEW scales the fonts by the scale factor you enter in the build specification.

The following fonts are available on the Unicon example target:

- Arial
- Arial Black
- Comic Sans MS
- Courier New
- Impact Condensed
- Tahoma
- Times New Roman

**Note**  Tahoma font is not scalable.
Graphs

Graphs can display plots that contain any number of points of the following data types:

- A single array of values. Graphs interpret array data as points on the graph and increment the x index by one starting at $x = 0$.
- A cluster with an array of x values and an array of y values.
- A 2D array of values, where each row of the array is a single plot.
- An array of plots, where each plot is a cluster that contains an array of x values and an array of y values.
- An array of clusters of plots, where a plot is an array of points and a cluster contains an x and a y value.
- A cluster of x, delta-x, and an array of y values.

Graphs do not support the following:

- Duplicate scales
- Grids
- Plot legends

Waveform graphs do not support the following:

- Horizontal scroll bars

Plots on Waveform Graphs

You can display multiple plots on waveform graphs. Plotting more than 32,000 elements is unsupported. If you attempt to plot more than 32,000 elements, the first element of the array overwrites all points between 32,000 and 64,000. The waveform repeats every 64,000 elements. After the first 32,000 elements are plotted, the first element repeats 32,000 times.

The number of plots in the Plots shown numeric control in the Waveform Graph Properties dialog box must equal the actual number of plots or the application behaves incorrectly. The maximum number of plots is eight.

Plots on XY Graphs

Autoscaling the Y-axis for multi-plot XY graphs is not supported. Only one plot shows at a time if you enable autoscaling. You must disable autoscaling if you want multiple plots on an XY graph.
Chapter 11  Implementing User Interface Support

Labels

Word wrapping in free labels is unsupported. You must create new lines for text that spans multiples lines.

Listboxes

The following listbox elements are unsupported:
• Auto-size row height
• Editable headers
• Horizontal scroll bar
• Keyboard modes
• Index display
• Item dragging
• Moveable column separators
• Multiline input
• Multiple fonts in the same table
• Row and column headers
• Selection scrolling
• Smooth scrolling

Multicolumn Listboxes

The Top Left Visible Cell property returns only the top row element of the cluster.

Menus

Empty menus do not generate events.

Customizing shortcut menus for front panel controls is unsupported.

Numeric Controls and Indicators

Numeric controls and indicators support the following:
• Captions and labels.
• Floating-point, scientific, and engineering notations.
• Multiple needles on dials, meters, and gauges.
• Resizing and changing the color of the increment and decrement buttons.
Numeric controls and indicators do not support the following:

- Radix or unit labels.
- Significant digits.
- Options in the advanced editing mode of the **Display Format** page of the **Numeric Properties** dialog box.
- SI (System International) or relative time notations.
- Time and date formats. Use a time stamp control instead.
- Dials, meters, and slides do not support digital displays or ramps. You cannot invert the scales for dials, meters, or slides. Scales must be linear.
- Dials, meters, and slides do not support transparency.
- Data ranges that are different than the display range.
- 64-bit signed or unsigned integers. If you use 64-bit signed or unsigned integers, LabVIEW generates 32-bit signed or unsigned integers when you build the VI into an application.
- Extended-precision, floating-point numerics. If you use extended-precision, floating-point numerics, LabVIEW generates double-precision, floating-point numerics.
- Two sliders on a slide control. If you use two sliders, an error occurs when you build the VI into an application, and the application contains only the first slider and its default value.

**Path Controls**

Path controls do not have the same functionality on embedded targets as on Windows. Applications translate path controls and indicators into string controls and indicators and treat the data as strings, which means the **Browse** button in the path control is not visible on the target.

**Picture Controls**

If you use a picture control, LabVIEW ignores the **Erase First** shortcut menu item and draws the new image over the previous image.

**Events**

You can use the following Control events with picture controls:

- Mouse Up
- Mouse Down
- Mouse Move
Showing Ampersands (&) in Picture Controls

You must include two & symbols in a string to display a single & in a picture control if you are using the Draw Text at Point or Draw Text in Rect VI.

Refnums

You can add refnums to the front panel window, but refnums are not visible when you run the application on the target.

VIs support the following refnums:

- Application
- Control
- File
- Menu
- Occurrence
- TCP Network Connection
- UDP Network Connection
- VI

Scaling Front Panel Objects

You can change the scale factor of front panel objects to accommodate different screen sizes.

Scaling has the following implications in LabVIEW:

- The default scale factor for FLTK is one. The controls and indicators on the Controls palette are not scaled. You can select a different scale factor in the Build Specifications dialog box.
- When you right-click a block diagram terminal and select Create►Control or Create►Indicator from the shortcut menu, LabVIEW adds a control or indicator to the front panel window in the size and style for the current target.
- LabVIEW does not scale bitmap images when you build the VI into an application. The bitmaps retain their original size to avoid the aliasing effects that can result if the bitmaps are reduced in size.
String Controls and Indicators

If you limit a string to a single line, text justification is disabled for the string control.

String controls support the following display modes:
- Normal
- Password

String indicators support all types of display modes, which includes the following:
- Normal
- "\" codes
- Hex
- Password

Using Password Display

The following are unsupported if you use the password display with string controls:
- Scroll bars
- Multiline input
- Using a Property Node with the Limit to Single Line? property to change the multiline option

Note: The Limit To Single Line? property removes the scroll bars but does not limit the text to a single line when you use this property.

Tab Controls

Tab controls support the following:
- Decorations
- Free labels

The following configuration options are unsupported:
- Tabs on the left or right side of the tab control
- Disabled pages
- Variable size tabs
- Different color tabs
- Hidden tabs
• Tabs with multiple rows
• Images on tabs

Note Although images are not supported on the actual tabs, you can place images on tab control pages.

Time Stamp Controls and Indicators
Embedded VIs support time stamp controls and indicators differently than VIs you run on Windows.

Minimum and Maximum Time Values
FLTK uses Unix time, so the minimum time value is 00:00:01 a.m., Friday, January 1, 1970 Unix Time. The maximum time value is 06:28:15 a.m., Monday, February 6, 2038 Unix Time.

If a date is outside of the minimum and maximum range, you receive an error. With VIs you run on Windows, LabVIEW rolls over the value.

Editing Modes
The editing mode sets whether to edit the format and precision of the data using the default view or using format codes. You select the editing mode on the Display Format page in the Time Stamp Properties dialog box.

Default Editing Mode
Time stamp controls support the Absolute time default format only.

Advanced Editing Mode
Use %<%.3X> to display the time with three digits of precision after seconds, %<%.4X> to display the time with four digits of precision, and so on.

Variant Controls
Variant controls are supported but are not visible in the application running on the target.

Variant data does not conform to a specific data type and can contain attributes. LabVIEW represents variant data with the variant data type.
You cannot initialize variant data with default data in a VI. Placing a control, indicator, or constant that contains default data on the block diagram of the VI results in an empty LabVIEW variant.

**Supported Properties**

You can use the following properties with the Property Node. Each type of control has its own class. You receive a broken Run button if you attempt to use an unsupported property.

**Note** If you pass a control reference to an embedded subVI, the control reference is not valid within that subVI.

**Generic Properties**

You can use the following generic properties on all supported controls:

- Bounds
- Bounds:Area Height
- Bounds:Area Width
- Disabled
- Key Focus
- Position
- Position:Left
- Position:Top
- Text (control and indicator labels only)
- Value
- Value (Signaling)
- Visible

**Array Controls and Indicators**

You can use the following properties with array controls and indicators:

- Index Values
- Index Visible (read only)
- Number of Columns (read only)
- Number of Rows (read only)
Boolean Controls and Indicators
You can use the following properties with Boolean controls and indicators:
- Blinking
- Boolean Text
- Cluster Size (radio buttons control only)
- Cluster Size:Height (radio buttons control only)
- Cluster Size:Width (radio buttons control only)
- Color (radio buttons control only)
- Colors [4]
- Strings [4]
- Text Colors:BG Color
- Text Colors:Text Color

Cluster Controls
You can use the following properties with cluster controls:
- Cluster Size
- Cluster Size:Height
- Cluster Size:Width
- Controls[]

Enumerated Type Controls
You can use the following properties with enumerated type controls:
- Number of Items (read only)
- Ring Text
- Strings []
- Text Colors:BG Color
- Text Colors:Text Color

Graphs and Charts
You can use the following properties with graphs and charts:
- Allow Drag (waveform graph only)
- Cursor Legend Visible (waveform graph only)
- Display Format
• Display Format: Format
• Display Format: Precision
• Mapping Mode
• Name Label
• Plot Area: Colors: BG Color
• Plot Area: Colors: FG Color
• Range: Increment
• Range: Maximum
• Range: Minimum
• Range: Start
• Visible

Listbox Controls
You can use the following properties with listbox controls:
• Item Names
• Number of Rows
• Top Row

Numeric Controls and Indicators
You can use the following properties with numeric controls and indicators:
• Housing Colors: BG Color (dials, meters, gauges, and slide controls only)
• Housing Colors: FG Color (dials, meters, gauges, and slide controls only)
• Data Entry Limits: Increment
• Data Entry Limits: Minimum
• Data Entry Limits: Maximum
• Display Format
• Display Format: Format
• Display Format: Precision
• Range: Increment
• Range: Maximum
• Range: Minimum
• Response to Value Outside Limits: Increment
• Response to Value Outside Limits: Maximum
• Response to Value Outside Limits: Minimum
• Text Colors: BG Color
• Text Colors: Text Color
• Visible

**Ring Controls**

You can use the following properties with ring controls:

• Number of Items (read only)
• Ring Text
• Strings []
• Text Colors: BG Color
• Text Colors: Text Color

**String Controls and Indicators**

You can use the following properties with string controls and indicators:

• Display Style
• Limit To Single Line?

**Note** The Limit To Single Line? property removes the scroll bars but does not limit the text to a single line when you use this property except for string controls that use the password display type, which is always a single line with no scroll bars.

• Text
• Text Colors: BG Color
• Text Colors: Text Color

**Time Stamp Controls and Indicators**

You can use the following properties with time stamp controls and indicators:

• Colors: BG Color
• Colors: FG Color
• Data Entry Limits: Increment
• Data Entry Limits: Maximum
• Data Entry Limits: Minimum
Chapter 11 Implementing User Interface Support

- Format String
- Response to Value Outside Limits: Increment
- Response to Value Outside Limits: Maximum
- Response to Value Outside Limits: Minimum

Waveform Charts
You can use the following property with waveform charts:
- Update Mode

Waveform Graphs
You can use the following properties with waveform graphs:
- Active Cursor
- Cursor Color
- Cursor Position
- Cursor Position: Cursor X
- Cursor Position: Cursor Y
- Cursor Style
- Visible

Supported Methods
You can use the following methods with the Invoke Node. Each type of control has its own class. You do not receive a broken Run button if you attempt to use an unsupported method, but you do receive an error when you build the VI into an application.

Generic Methods
You can use the following generic method on all supported controls and VIs except for graphs and charts:
- Reinitialize To Default

Graphs and Charts
You can use the following methods with graphs and charts:
- Map Coordinates To XY
- Map XY To Coordinates
String Controls and Indicators
You can use the following methods with string controls and indicators:
- Get Nth Line
- Size to Text

Waveform Graphs
You can use the following method with waveform graphs:
- Get Plot At Position

VIs
You can use the following methods in VIs:
- Abort VI
- Default Values: Reinitialize All To Default

Unsupported Controls and Indicators
The following controls and indicators are unsupported:
- 3D graphs
- ActiveX controls
- Digital waveform graphs
- I/O controls
- Intensity graphs and charts
- Minmax plots
- Mixed checkbox controls
- Picture ring controls
- Polar plots
- Radar plots
- Tables
- Text and picture ring controls
- Tree controls
- Transparent controls (with the exception of Boolean controls and picture controls)
- VISA resource name controls
- XControls
Elemental I/O

Elemental I/O is a flexible, user-defined way to create I/O for LabVIEW embedded 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 among 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
- I/O classes
- I/O pins
- I/O resources
- I/O plug-in VIs

National Instruments strongly recommends looking at the Simulated I/O implementation example as you implement your Elemental I/O. The Simulated I/O implementation example is located in the `labview\Targets\NI\Embedded\eio` directory.

**I/O Devices**

An Elemental I/O device is the software representation of hardware that performs I/O on an embedded target by using a collection of I/O resources, pins, and plug-in VIs to define the I/O for a target. You can reuse an I/O device on multiple targets so 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.
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.

I/O Pins

A Elemental I/O 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.

Pins are optional. Use pins if you have conflicting I/O devices.

I/O 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 analog input, analog output, digital input, and digital output. You can bind a pin to a resource.

Elemental I/O Plug-In VIs

The Elemental I/O plug-in VIs define the properties of a resource and the implementation of the I/O.
How Users Use Elemental I/O Nodes

Users create Elemental I/O items in the **Project Explorer** window by right-clicking a target and selecting **New»Elemental I/O** from the shortcut menu to open the **Elemental I/O Node Properties** dialog box. 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 Node Implementation plug-in VIs.

When the Elemental I/O Node executes on the embedded target, the target executes the 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 whatever is valid for the target.

Creating Elemental I/O Plug-In VIs

Use the Elemental I/O 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 you do not create the plug-in VIs before running the wizard, you receive an error when you try to generate the **eio.xml** file, which is the file that defines the Elemental I/O implementation for your target. If you modify the plug-in VIs in any way, you must rerun the wizard and restart LabVIEW to ensure that the information in the target matches the VIs.

**Note** All Elemental I/O 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.

The template VIs include all possible controls and indicators for the Elemental I/O 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 controls that you do not need. The template VIs are located in the following directory:

```
labview\examples\lvemb\EDM\EIO\EIO User Plugin Templates
```
Refer to the block diagram comments in the template VIs for more information about the Elemental I/O plug-in VIs, such as required and optional parameters.

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 a user right-clicks an I/O item in the Project Explorer window and selects Properties from the shortcut menu. The values you set in the I/O Node Project Panel page apply to all nodes that refer to this I/O item. The Elemental I/O framework passes the project attributes to the I/O Implementation plug-in VI and the I/O Property Node Implementation plug-in 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 a 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. The Elemental I/O framework passes the node attributes to the I/O Implementation plug-in VIs.

Example plug-in VIs are located in the following directory:

`labview\Targets\NI\Embedded\eio\simulatedIO`

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

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

The I/O Node Implementation plug-in 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 Elemental I/O framework passes the project attributes and the node attributes to the I/O Implementation plug-in VI for the I/O item.
I/O Property Node Implementation Plug-In VI

If you want to support programmatically setting I/O properties, you must include a I/O Property Node Implementation plug-in VI for each property you support. The I/O Property Node Implementation plug-in VI is substituted for the Elemental I/O Property Node wherever you place an Elemental I/O Property Node on the block diagram. This plug-in VI can communicate with hardware, call device drivers, set global variables used by the I/O Implementation plug-in VI, and so on. These VIs run on the embedded target, so anything that is valid for the target is allowed in these VIs. The Elemental I/O framework passes the I/O Property Node Implementation plug-in VI to the project attributes for the selected I/O item.

I/O Specify Node Plug-In VI

Use the I/O Specify Node plug-in VI if you want Elemental I/O Nodes type determined at edit time, as opposed to when you define the I/O through the Elemental I/O Wizard.

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

The I/O Specify Node plug-in VI returns the type of the I/O Node, which is either the return type for read Elemental I/O Nodes or the input type for write Elemental I/O Nodes. If you use an I/O Specify Node plug-in VI, you must include an I/O Node Implementation plug-in VI for each type returned.

I/O Project and Node Attribute Validation Plug-In VI

Use an I/O Project and Node Attribute Validation plug-in VI to validate the attributes of the I/O item in the Project Explorer window and the Elemental I/O Nodes on the block diagram. The I/O Project and Node Attribute Validation plug-in VI runs at edit time on the host computer.
Chapter 12  Elemental I/O

Using the Elemental I/O Device Wizard to Create Elemental I/O Devices, Classes, Pins, and Resources

After you create the Elemental I/O plug-in VIs, use the Elemental I/O Device Wizard to define the Elemental I/O devices, classes, pins, and resources 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 use the Elemental I/O Device Wizard.

1. Select Tools»Product Name»Elemental I/O Device Wizard, where Product Name is the product menu item, to display the Elemental I/O Device Wizard.

2. On the Name the Device page of the wizard, enter the name of the new Elemental I/O device.

3. Enter a description for the Elemental I/O device in the I/O Device Description text box. LabVIEW displays what you enter here in the Select Elemental I/O dialog box.

4. Click the Next button to display the Select the Elemental I/O Classes page.

5. 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.

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

7. Add I/O 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.

8. Click the Next button to display the Add Pins page.

Note You also can click the Load existing I/O target button to select an existing I/O device.
9. (Optional) Define the physical pins on the device. Use I/O 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. When 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 define any pins. If you have shared resources, National Instruments recommends defining the pins.

10. Click the Next button to display the Assign the Pins to Resources page.

11. (Optional) Define the connection between the I/O pins and the I/O 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. If you do have shared resources, National Instruments recommends assigning pins to I/O resources.

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

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

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

15. Select the target to which you want to add Elemental I/O and click the Select Target button. When you select a target, LabVIEW displays the path in the Output Directory.

16. Click the Generate button. The wizard creates a directory in the location you specify in the Output Directory and adds several VIs the Elemental I/O framework needs and an eio.xml file in the new directory. The eio.xml file contains the info from the Elemental I/O Wizard. The eio.xml file must be in the directory or subdirectory specified in the Output Directory box.

17. Click the Finish button to exit the wizard.

Note If you have more than one Elemental I/O device for a target, you select the Elemental I/O device by right-clicking the target in the Project Explorer window and selecting Select Elemental I/O from the shortcut menu.
Adding Elemental I/O Devices to an Embedded Target

After you create Elemental I/O, complete the following steps to add the Elemental I/O device implementation to an embedded target.

1. Select `Tools»Microprocessor SDK»Target Editor` to display the Target Editor.
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 through the Elemental I/O Wizard in the `Path to Elemental I/O directory` text box. This makes any `eio.xml` file in that directory or subdirectory available for that target.
5. Click the `OK` button.
6. Select `File»Save` in the Target Editor to save the target.
7. Restart LabVIEW to start using Elemental I/O with the target.

Elemental I/O Implementation Example

The Simulated I/O example contains an Elemental I/O implementation example.

(Microprocessor SDK) The PHYTEC example target and the Simulated I/O example contain Elemental I/O implementation examples.

The Simulated I/O example implementation is located in the following directory:

```
labview\Targets\NI\Embedded\eio
```
Porting the LabVIEW C Code Run-Time Library to a New Platform

With the exception of the os directory, you usually do not have to modify the code in the labview\CCodeGen\libsrc subdirectories because 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.

If you are porting LabVIEW to a new platform, you must do the following:

- Map the generic function calls to OS-specific function calls using #define statements
- Set feature flags
- Port OS-specific components
- (Optional) Override the default RT FIFO size limits
- (Optional) Define constants for static memory allocation support

#defines in LVDefs_plat.h

A series of #define statements map the generic function calls, such as StrCaselessCompare( x, y ), to OS-specific function calls, such as strcasecmp( 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:

- #define BigEndian—Defines the data storage format as big endian or little endian. Define as 0 for little endian and 1 for big endian.
- #define ALIGNMENT—Defines the alignment of data.
- typedef signed char int8;—In the LabVIEW C Code Run-Time Library code, int8 is used everywhere and must be defined to a signed 8-bit integer.
Chapter 13  Porting the LabVIEW C Code Run-Time Library to a New Platform

- `#define StrCopy(x, y) strcpy(x, y)`—In the LabVIEW C Code Run-Time Library code, `StrCopy` is used everywhere and must be defined to a function that copies a string.
- `#define SocketSupport`—Defines the sockets. Define as 1 to support BSD sockets.

**Feature Flags**

The following are just some of the flags in `LVDefs_plat.h`. Feature flags turn on and off large sets of features that correspond to pieces of hardware that might 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 Microprocessor SDK example targets and operating systems, but the code is present.</td>
</tr>
<tr>
<td>IrDASupport</td>
<td>Controls infrared (IrDA) support. IrDA support has not been ported to any of the Microprocessor SDK example targets and operating systems, but the code is present.</td>
</tr>
<tr>
<td>SerialSupport</td>
<td>Controls serial communication support.</td>
</tr>
<tr>
<td>TCPUDPSupport</td>
<td>Controls TCP and UDP support along with the <code>SocketSupport</code> flag in <code>CCGNetConnSupport.c</code>, which is located in the <code>CCodeGen/libsrc/comms</code> directory.</td>
</tr>
<tr>
<td>FileSupport</td>
<td>Controls file I/O support.</td>
</tr>
<tr>
<td>PosixFiles</td>
<td>Controls support for the POSIX file system.</td>
</tr>
</tbody>
</table>
OS-Specific Components

Some functionality is OS-specific. The generated C code calls these OS-specific functions when the corresponding 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:

`sources\labview\CCodeGen\libsrc\os`

**Critical Sections: LVCritSect.c**

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

- `InitLVCriticalSections()` and `UninitLVCriticalSections()`
  
  **Purpose:** Called from LVEmbeddedMain.c to initialize and tear down critical sections.

- `LVCriticalSectionCreate()`
  
  **Purpose:** Allocates the critical section record and returns a pointer to it.

- `LVCriticalSectionDelete(LVCriticalSection critsect)`
  
  **Purpose:** Frees a critical section record.

- `LVCriticalSectionEnter(LVCriticalSection critsect)`
  
  **Purpose:** Called to indicate when a section of code is entered that must not be interrupted.

- `LVCriticalSectionLeave(LVCriticalSection critsect)`
  
  **Purpose:** Called to indicate when the critical section is finished.

**Events: LVEvent.c**

*Note* The event functions are real-time events. The functions are not UI events.
Occurrences and Timed Loops depend on `LVEvent.c`, which contains the following functions:

- `InitLVEvents()`, `UninitLVEvents()`  
  **Purpose:** Called from `LVEmbeddedMain.c` to initialize and tear down events.

- `LVEventCreate()`  
  **Purpose:** Allocates an event record and returns a pointer to it.

- `LVEventDelete(LVEvent event)`  
  **Purpose:** Frees an event record.

- `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 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 Real-Time 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)

**Purpose:** Implements atomic compare-exchange operations for unsigned 32-bit integers.
LVNBOpsAtomicCompareExchange16(uInt16 *inspectedLocation, uInt16 oldValue, uInt16 newValue)

**Purpose:** Implements atomic compare-exchange operations for unsigned 16-bit integers.

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

**Purpose:** Implements atomic compare-exchange operations for unsigned 8-bit integers.

### 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.

InitOEMISRs(), UninitOEMISRs()

**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

This OS-specific C file implements printf, which uses the One Button Dialog function for non-UI targets. If you implement user interface support, use the CCG Console Out VI for printf functionality instead of PDAStrSupport_os.c. If you do not implement user interface support, you can use PDAStrSupport_os.c or the CCG Console Output VI.
Time: CCGTimeSupport_os.c

This OS-specific C file 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 LabVIEW C Code Run-Time Library 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 In Seconds function.

Serial: PlatformSerial_os.c

Use this OS-specific C file, which contains the following functions, for serial functions and for debugging over a serial connection.

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.
Chapter 13 Porting the LabVIEW C Code Run-Time Library to a New Platform

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 this C file, which contains the following functions, for the Embedded CAN VIs and for debugging over a controller area network (CAN) connection.

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 Microprocessor SDK example targets support Berkeley sockets. The implementation file is CCGNetConnSupport.c, which is located in the following directory:

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

CCGNetConnSupport.c can 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 table 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 a Real-Time FIFO in elements.</td>
</tr>
</tbody>
</table>
RT FIFO Size Limits

The following two constants in nbitemtable.c limit the number of RT FIFOs in an embedded application:

- kNBITEMTABLE_DEFAULT_MAJOR_SIZE
- kNBITEMTABLE_DEFAULT_MINOR_SIZE

The limit is the multiplier of these two numbers. The default values are both 64; thus the limit is 4,096. If you want to increase this limit, override the definitions in LVDefs_plat.h.
Implementing Interrupt Service Routine 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 in OEM_LVISR.c if you want to support ISRs.

**InitOEMISRs**

```c
Boolean InitOEMISRs();
```

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

Returns TRUE on success. Returns FALSE on failure.

**UninitOEMISRs**

```c
Boolean UninitOEMISRs();
```

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

Returns TRUE on success. Returns FALSE on failure.
Chapter 14 Implementing Interrupt Service Routine Support

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. isr_vector is the intended vector, isr_param is the parameter. Both isr_vector and isr_param are passed to ISRFunc isr_runFunc. register_param is an output parameter that is unique to this ISR registration instance. The register_param value is passed to OEMISRUnregisterHandler when unregistering the ISR.

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. register_param is the register parameter that OEMISRRegisterHandler returns.

Returns TRUE on success. Returns FALSE on failure.

Tips for Supporting ISRs on a New Target

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. You must 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.

ISR VIs have the following characteristics:

- Are always serial
- Use stack variables
- Do not contain code that allows other VIs to directly call the ISR VI
Allowable operations for an ISR VI are a subset of the allowable operations for a subroutine reentrant VI. Unallowable operations include asynchronous operations, such as file I/O, TCP/IP, Bluetooth, IrDA, and so on, plus any operations that can cause memory allocation, such as manipulating strings, arrays, and so on.

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.

**ISR Implementation Example**

Refer to the VxWorks ISR example implementation, which is located in the following directory:

```
labview\CCodeGen\libsrc\os\vxworks\OEM_LVISR.c
```
15

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

You must define and implement your target-specific timing sources. You use the Target Editor to define the timing sources.

Refer to the *Defining Timing Sources* section of Chapter 7, *Creating a New Embedded Target*, for information about the *Timing Sources* tab in the Target Editor.

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 `LVDefs_plat.h` as the name of the function to call before calling the top-level VI and after initializing the LabVIEW C Code Run-Time Library.

Call the `FireInternalTSource()` function from driver code to generate the external timing source. You must call `FireInternalTSource()` with the same index as `CreateInternalTSource()` creates.

**Note** `FireInternalTSource()` calls the `LVEventSet()` function, which not all targets can call from ISRs. 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 releasing the LabVIEW C Code Run-Time Library.
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 `iIntTSource` 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.
Using the Timed Loop VIs to Define Timing Sources

Use the polymorphic Timed Loop VIs to create and manipulate timing sources for a Timed Loop. The polymorphic 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.

Implement the LVGetTicksExt () function to 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. Adding additional timing sources make 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.
Memory Mapping

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

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. You can define this macro as section sdram_section to notify the compiler that the array goes to the sdram_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. You cannot 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 _TEXT_SECTION to place the generated C code of a VI in a non-default code section.

The memory mapping configuration persists in the LabVIEW project file (.lvproj).
Memory Mapping Plug-In VIs

To implement memory mapping, you must implement the Memory Mapping plug-in VIs.

Refer to the Axiom CMD565, VxWorks example target for an example implementation.

**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:

```plaintext
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:

```plaintext
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.

**How Users Use Memory Mapping**

Users can change the implicit mapping, if necessary, by right-clicking the build specification in the Project Explorer window and selecting Map Memory from the shortcut menu to open the Map Memory dialog box. 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.
You can implement instrumented debugging with LabVIEW to communicate with an embedded application running on an embedded target to 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 use C functions to implement the instrumented debugging communication layer on the embedded target. If you are using serial or TCP communication, you do not have to implement these communication layers. Instead, you can use the existing communication layer the Microprocessor SDK example targets use.

**Implementing the Instrumented Debugging Communication Layer for LabVIEW**

If you are using a communication layer other than serial or TCP, you must implement host debugging plug-in VIs to initialize the debug connection. The host debugging plug-in VIs only implement the 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.

The following host debugging plug-in VIs implement the instrumented debugging communication layer for LabVIEW:

- Open VI
- Close VI
Chapter 17  Implementing Instrumented Debugging for a Custom Communication Layer

- Read VI
- Write VI

Place these VIs in your target directory.

Refer to the Spectrum Digital DSK6713 example target for a host debugging implementation example. The host debugging plug-in VIs for the Spectrum Digital DSK6713 example target are located in the following directory:

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

**Note**  If you start with this example implementation, the labels of the controls and indicators must match your implementation exactly.

**Open VI**

Establishes a connection with the embedded target.

Your Open VI must have the following parameters on the connector pane:
- **connection data**—A variant indicator. Place data for the debugging connection in this variant. You pass in this variant 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.
- **error out**—An error cluster indicator.

**Close VI**

Closes the connection with the embedded target.

Your Close VI must have the following parameters on the connector pane:
- **connection data**—A variant control. You can modify this variant so you can communicate between debugging plug-in VIs you call at different times.
- **error out**—An error cluster indicator.

**Read VI**

Reads data from the embedded target.

Your Read VI must have the following parameters on the connector pane:
- **connection data**—A variant control. The contents of this variant come from the last invocation of an instrumented debugging plug-in VI.
Chapter 17  Implementing Instrumented Debugging for a Custom Communication Layer

- **bytes to read**—A 32-bit signed integer numeric control that is the number of bytes read from the embedded target.
- **connection ID**—A 32-bit signed integer numeric control that identifies the instrumented debugging session.
- **error out**—An error cluster indicator.

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

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

nixtargetReadData.vi sends the data that you read to LabVIEW. Wire the data read to the **Data Read** input in nixtargetReadData.vi. Wire the **connection ID** from your Read VI to the **Connection ID** input in nixtargetReadData.vi.

**Write VI**

Writes data from LabVIEW to the embedded target. Your Write VI must be blocking so when the VI finishes, LabVIEW expects this VI to send the data to the embedded application.

Your Write VI must have the following items on the connector pane:
- **connection data**—A variant control. The contents of this variant come from the last invocation of an instrumented debugging plug-in VI.
- **data to send**—A string control. The contents of this string is the data to send to the embedded target.
- **error out**—An error cluster indicator.

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

To implement instrumented debugging for your embedded target, you must create an instance of the following C functions in the LabVIEW C Code Run-Time Library:
- DebugBytes
- DebugConnect
- DebugDisconnect
- DebugRead
• DebugWrite
• PrepareConnection
• ReleaseConnection

Also, you must assign global function pointers to the functions you create so that the LabVIEW C Code Run-Time Library can use the functions for debugging connections.

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

The Microprocessor SDK provides examples of serial and TCP implementations for the target-side communication layer.

The serial implementation is located in the following file:

labview\CCodeGen\libsrc\comms\SerialMessaging.c

The TPC implementation is located in the following file:

labview\CCodeGen\libsrc\comms\TCPMessaging.c

If you are using TCP communication, you should not have to modify TCPMessaging.c unless you want to use a port other than 0.

**DebugBytes**

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.

**DebugConnect**

eRunStatus DebugConnect(void)

Opens a connection to LabVIEW and establishes a connection to your Open host debugging plug-in VI. Return eFail if there is an error. Return eNotFinished if you cannot connect to the host. Return eFinished for a successful connection.
Chapter 17  Implementing Instrumented Debugging for a Custom Communication Layer

DebugDisconnect

eRunStatus DebugDisconnect(void)

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

PrepareConnection

eRunStatus PrepareConnection(void)

Prepares a debugging connection. If you are using TCP/IP, this function opens a non-blocking TCP/IP port and listens for a connection. The DebugConnect function then accepts the connection. If you are using a GSM modem to connect to the host computer, this function configures the radio modem, starts the radio modem, and enters the PIN code. The DebugConnect function checks for and accepts pending calls. Return eFail if there is an error. Otherwise, return eFinished.

DebugRead

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 and reqBytes are equal when the read is complete. Allocate a buffer in pData to return the data. Return eFail if there is an error. Otherwise, return eFinished.

ReleaseConnection

eRunStatus ReleaseConnection(void)

Releases the resources the PrepareConnection function allocates. If you are using TCP/IP, this function also closes the listening port. Return eFail if there is an error. Otherwise, return eFinished.

DebugWrite

eRunStatus DebugWrite(const char* pData, uInt32 size)

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

To build an embedded VI into an embedded application that uses instrumented debugging, you must provide the appropriate arguments to the LabVIEW C Code Generator and assign the target-side instrumented debugging functions to the run-time function pointers in `DebugComm.c`, which is located in the following directory:

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

Assign the target-side instrumented debugging functions to the global LabVIEW C Code Run-Time Library function pointers in the `DebugComm.c` run-time source file in the `CCGDebugInitializeComm()` function.

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 an application you can debug. Make a new `#elif` case in `CCGDebugInitializeComm()`, and assign the follow function pointers:

- `CCGDebugConnect`: Assign this pointer to the `DebugConnect` function.
- `CCGDebugDisconnect`: Assign this pointer to the `DebugDisconnect` function.
- `CCGDebugRead`: Assign this pointer to the `DebugRead` function.
- `CCGDebugWrite`: Assign this pointer to the `DebugWrite` function.
- `CCGDebugBytes`: Assign this pointer to the `DebugBytes` function.
- `CCGDebugPrepareConnection`: Assign this pointer to the `PrepareConnection` function.
- `CCGDebugReleaseConnection`: Assign this pointer to the `ReleaseConnection` function.
- `CCGDebugWaitOnMessage`: Assign this pointer to the `CCGDebugGenericWaitOnMessage` function.
- `CCGDebugGetMsg`: Assign this pointer to the `CCGDebugGenericGetMsg` function.
- `CCGDebugSendMsg`: Assign this pointer to the `CCGDebugGenericSendMsg` function.
On Chip Debugging

You can use custom plug-in VIs to extend the on chip debugging (OCDI) debugging framework, which is compatible with the stabs format, to support various in-circuit emulators (ICE) and toolchains for your target. The following figure illustrates the OCDI debugging framework.

The core of the OCDI debugging framework is the debug database. The debug database contains a map of all possible signals, breakpoint, controls, and indicators in the physical memory of the embedded target. The debug database also contains line numbers and information for each node on the block diagram of the embedded VI.

When you build an embedded VI into an embedded application using OCDI debugging options, the LabVIEW C Code Generator generates C files from the block diagram.
The C parser parses every C file the LabVIEW C Code Generator generates for special comments to initialize the debug database with breakpoint, signal, and control records.

Your toolchain compiler outputs an intermediate object module and list files so the list parser can translate line numbers in the C code into relative symbol offsets. The list parser updates the line number to the function (name, offset) pair.

The linker generates a map file that contains the absolute addresses for all symbols. The map parser reads the map file the linker generates and resolves all symbols in the debug database by iterating through and updating all of the records in the debug database.

The debug database generates a LabVIEW map (LVM) file, which is an ASCII text file. The LVM file is the contents of the debug database and contains all of the information required to debug an embedded application without the source code and intermediate files.

The debug daemon loads the LVM file when the debugging session is initialized. The debug daemon communicates with the emulator using the OCDI Debugging plug-in VIs, which implement a low-level interface to a third-party emulator library.

Refer to Chapter 19, *Understanding the LabVIEW Embedded LVM File*, for more information about the LVM file. Refer to the following sections in this chapter for more information about the other pieces of the OCDI debugging framework.

`simple_math.vi`, shown in the following figure, is used as an example throughout the information about OCDI debugging.

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

The debug database is a collection of records for all controls, signals, and breakpoints in the embedded application VI hierarchy. The debug database also contains C line number information for nodes on the block diagram. The build VI initializes the database before the first C file is compiled, usually in the setup stage of the build VI. The C parser inserts records in the debug database for all signals, controls, and breakpoints. The list parser maps the breakpoint line numbers into relative symbol offsets and updates all breakpoint records in the debug database. Finally, the map parser resolves all symbols and updates all database records so they contain absolute addresses of the embedded application. The database snapshot is then saved in the LVM file, which is 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.

C Parser

The C parser searches the generated C code for special comments and maps the comments to line numbers. 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 indicate potential probes, breakpoints, controls, and indicators. For example, the following code sample shows the comments generated for simple_math.vi:

```
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_OCDIFlag[0] & 0x1,
&simple_math_OCDIFlag, 0)/* OCDI::BreakPoint::Node(1,
2, 3) */
```
This information, which is stored in the debug database, translates control and signal IDs into relative symbol addresses. The database records for all breakpoints only map breakpoint IDs into line numbers. For controls and indicators, the C parser does not store actual offsets because the parser does not know the layout of the ControlData structure. Instead, the parser stores indexes to the ControlData table. To convert the index value to the actual offset, you must multiply the index value with the size of the ControlData structure and add the offset of the hValue variable.

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:

```text
[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
```
For nodes, the syntax for this portion of the LVM file is as follows:

\[
\text{Node}(a, b, c)@x=\ldots
\]

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:

\[
\text{Location.Node}(1,2,3)@[123\ldots456]
\]
\[
\text{Location.SNode}(1,2,3)@[123\ldots456]
\]
\[
\text{Location.SRN}(1,2,3)@[123\ldots456]
\]

where \text{Location.Node} represents nodes, \text{Location.SNode} represents structures, and \text{Location.SRN} represents self-referencing nodes. \([123\ldots456]\) 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:

\[
\text{CID}_i=\ldots
\]
\[
\text{SID}_i=\ldots
\]

where \text{CID} represents Control ID, \text{SID} represents Signal ID, and \(i\) represents the unique LabVIEW-generated object ID.

\[\text{C Compiler}\]

The build plug-in VI calls a third-party target-specific compiler for each generated LabVIEW C Code 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 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. Executing the compiler with the correct debugging and list options is important for generating 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 example targets 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 still can build embedded applications in the On Chip (Release) debugging mode. The On Chip (Release) mode implements the OCDI_Checkpoint macro, so the running embedded application must evaluate simple checkpoint expressions every time the application approaches a possible LabVIEW breakpoint. The OCDI_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 embedded application.

Note The OCDI_Checkpoint macro is defined in LVDefs_plat.h.

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, the list parser populates data type alignment information that is specific to how 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 resolving the signals and nodes to a series of hexadecimal offsets, which are shown in bold. The memory locations are still only known relative to yet unresolved symbols. Absolute memory address information is not available at this phase.

```
[simple_math.vi]
Node(1, 2, 3)@64=simple_math_Run+0x58
Node(2, 3, 3)@68=simple_math_Run+0x6C
```
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 which 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, generate this information 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:

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

\[ L \] represents the length of the entire data type in bytes
\[ j \] represents the offset from the beginning of the data structure in bits
\[ k \] represents the length of the data member in bits

Refer to Chapter 19, *Understanding the LabVIEW Embedded LVM File*, for more information about the LVM file and the [Offset_Table].

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

**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. You can execute the linker 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, such as the VxWorks downloadable module. The target must resolve all symbols at the module load time.

**Map Parser**

The map parser runs through the map file (.map) the linker generates 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. The map parser then resolves all symbols in the debug database. The map parser does not execute for targets that build a downloadable module. The target must resolve all symbols at module load time.
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 bold in the following LVM file for `simple_math.vi`:

```
[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 Daemon**

The debug database provides 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 the OCDI debugging plug-in VIs, which are emulator-specific VIs that instrument actual emulator hardware. All OCDI debugging plug-in VIs 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 OCDI debugging plug-in VIs do not have to deal with internal LabVIEW object IDs.

**Comparing On Chip and On Chip (Release) Modes**

Depending on how you design and implement your build specifications, 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 the phrase 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 in the following situations:

- If your C compiler cannot generate list files to map lines of generated C code to the relative symbol offsets.
- If you need to debug an optimized application.
- If your emulator has limited breakpoints.
- If you want to use hardware breakpoints.

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

### Adding Support for an Emulator that Uses the GCC Toolchain

Complete the following steps to add support for the GCC toolchain.

**Note** If you have a DWARF parser, generate the LVM file instead of completing the following steps.

1. Call `EMB_Debug_DD_Init.vi`, which is located in the `labview\vi.lib\LabVIEW Targets\Embedded\Debug\DD` directory, from your build VI before the build script executes to initialize the debug database. Refer to `LEP_vxworks_Build.vi` for an example of how to use `EMB_Debug_DD_Init.vi`. `LEP_vxworks_Build.vi` is located in the following directory:

   `labview\Targets\NI\Embedded\vxworks\vxworks_LEP_TargetPlugin`

2. Call `EMB_Debug_Map_Gcc_UpdateDD.vi`, which is located in the `labview\vi.lib\LabVIEW Targets\Embedded\Debug\Map\GCC` directory, 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. `LEP_x_ScriptCompiler.vi` should call this VI 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 so that the list parser that EMB_Debug_Map_Gcc_UpdateDD.vi calls can understand it. Refer to the following example batch file, which executes a GCC compiler to compile the C file and generates the list file in stabs format.

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

3. Call EMB_Debug_Map_GccResolveSymbols.vi and EMB_Debug_Map_GccResolveExtraSymbols.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 an example of 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. EMB_Debug_Map_GccResolveSymbols.vi and EMB_Debug_Map_GccResolveExtraSymbols.vi are located in the following directory:

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

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 LabVIEW project.

Refer to LEP_vxworks_cmd565_ScriptCompiler.vi for an example of retrieving configuration data, defining the release debug macro, and calling EMB_Debug_Map_Gcc_UpdateDD.vi to parse source and list files. LEP_vxworks_cmd565_ScriptCompiler.vi is located in the following directory:

```
labview\Targets\NI\Embedded\vxworks\cmd565\vxworks_cmd565_LEP_TargetPlugin
```

**Note** You must implement a new list parser for a non-GCC toolchain.
Adding Support for an Emulator that Uses a Non-GCC Toolchain

You must implement the list parser if your target uses a non-GCC toolchain. The purpose of the list parser is to map lines of special debugging comments in the generated code to relative offsets.

You must implement the following VIs:
- EMB_Debug_Map_x_UpdateDD.vi
- EMB_Debug_Map_x_ResolveSymbols.vi

**EMB_Debug_Map_x_UpdateDD.vi**

Calls the C parser to initialize the debug database with all breakpoints, signal, and control IDs. The object absolute addresses are left blank at this phase. LEP_x_ScriptCompiler.vi calls the list parser to parse the list file the compiler generates and assigns relative symbol offsets to all objects 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`. You must find the absolute addresses of the symbol, add the offset, and store the resulting address back to the debug database. 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. The debug daemon uses the type information 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 \((\text{type} \_\text{code}1, \text{type} \_\text{code}2, \text{and so on})\), where \text{type} \_\text{code}1, \text{type} \_\text{code}2 are numeric values 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_RelativeAddress.vi
- Emb_Debug_Map_SignalOffset.vi

These VIs are located in the following directory:

```
labview\vi.lib\LabVIEW Targets\Embedded\Debug\Map\GCC
```
Refer to the following VI for an implementation example:

```plaintext
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 with the name of your emulator interface:

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

2. Copy the contents of the iOPEN directory, which contains the plug-in VI implementations for all WindRiver emulators.

3. Rename all of the VIs in the directory you created in step 1 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 VIs you renamed in step 3.

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

## OCDI Debugging Example

The following example targets support OCDI debugging:

- Axiom CMD565, VxWorks RAM Image
- Axiom CMD566, VxWorks
- Unix Console

An example list parser for GCC-compatible compilers that can generate list files in stabs format is located in the following directory:

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

Refer to the following directory for examples of support for various emulators:

```plaintext
labview\vi.lib\LabVIEW Targets\Embedded\Debug\OCDI
```
List Parser Plug-In VIs

You must implement the following plug-in VIs to compute the member offset from signal type information and target-specific alignment:

- EMB_Debug_Map_RelativeAddress.vi
- EMB_Debug_Map_SignalOffset.vi

Refer to the GCC implementation for an example of how to implement these VIs. The GCC implementation is located in the following directory:

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

**EMB_Debug_Map_RelativeAddress.vi**

Calculates the relative address of a symbol in the format `variable + offset` or `variable.member (type_code)`.

<table>
<thead>
<tr>
<th>relative address</th>
<th>base variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>last base variable</td>
<td>offset</td>
</tr>
<tr>
<td>last offset</td>
<td>offset+size</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.

**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>offset+length</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbol</td>
<td>offset</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.

## OCDI Debugging Plug-In and Utility VIs

The OCDI debugging plug-in VIs, which are emulator-specific VIs that instrument actual emulator hardware by implementing a low-level emulator interface the debug daemon uses. In most cases, the OCDI debugging plug-in VIs translates calls of the debug daemon interface into the API of the emulator vendor.

### IInit.vi

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 the low-level driver signals 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 (no error)** describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the **error in** value to **error out**. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI runs, it runs normally and sets its own error status in **error out**. 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

**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 that produced the error or warning. The default is an empty string.

**occurrence** indicates a breakpoint occurrence, which the low-level emulator driver signals 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 ran, **error out** contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the **error out** indicator and select **Explain Error** from the shortcut menu for more information about the error.

**status** is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

**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 that produced the error or warning. The default is an empty string.
IClearBreakpoints.vi

Clears breakpoints on given absolute addresses.

- **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 (no error)** describes error conditions that occur before this VI runs. The default is **no error**. If an error occurred before this VI runs, the VI passes the **error in** value to **error out**. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

- **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 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 ran, **error out** contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the **error out** indicator and select **Explain Error** from the shortcut menu for more information about the error.

- **status** is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.
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 that produced the error or warning. The default is an empty string.

IConfig.vi

Invokes an emulator-specific configuration dialog box. This VI is called when a user selects Configure from the build specification shortcut menu to configure the emulator. A use case for a configuration dialog box is if you want one configuration dialog box for multiple emulators.

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 (no error) describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to error out. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI runs, it runs normally and sets its own error status in error out. 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

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 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 ran, error out contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the error out indicator and select Explain Error from the shortcut menu for more information about the error.

status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

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 that produced the error or warning. The default is an empty string.

IContExecution.vi

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 (no error) describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to error out. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

- **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 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 ran, **error out** contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the **error out** indicator and select **Explain Error** from the shortcut menu for more information about the error.

- **status** is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

- **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 that produced the error or warning. The default is an empty string.
Downloads the embedded application to target memory. This plug-in VI is called when a user selects Download from the build specification shortcut menu, which instruments the emulator to download the application using dedicated or just memory write functions.

For example, some APIs have a download function and some do not. If an API does not have a download function, you must implement download using memory write functions.

**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 to the application image.

**error in (no error)** describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to error out. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

**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 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 ran, error out contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the error out indicator and select Explain Error from the shortcut menu for more information about the error.

status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

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 that produced the error or warning. The default is an empty string.

IExit.vi

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 (no error) describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to error out. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this...
information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

**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 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 ran, **error out** contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the **error out** indicator and select **Explain Error** from the shortcut menu for more information about the error.

**status** is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

**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 that produced the error or warning. The default is an empty string.

**IGetBreakpoint.vi**

Retrieves the current breakpoint when the low-level emulator driver signals a breakpoint occurrence.

**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 (no error) describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to error out. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI runs, it runs normally and sets its own error status in error out. 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

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 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 ran, error out contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the error out indicator and select Explain Error from the shortcut menu for more information about the error.

status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

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 that produced the error or warning. The default is an empty string.
IGo.vi

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 (no error)** describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the **error in** value to **error out**. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

**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 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 ran, **error out** contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the **error out** indicator and select **Explain Error** from the shortcut menu for more information about the error.
status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

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 that produced the error or warning. The default is an empty string.

**IMemoryRead.vi**

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 (no error)** describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the **error in** value to **error out**. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.
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 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 ran, error out contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the error out indicator and select Explain Error from the shortcut menu for more information about the error.

status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

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 that produced the error or warning. The default is an empty string.

IMemoryWrite.vi

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 (no error) describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to error out. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI runs, it runs normally and sets its own error status in error out. 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

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 that produced the error or warning. The default is an empty string.

event out contains error information. If error in indicates that an error occurred before this VI ran, error out contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the event out indicator and select Explain Error from the shortcut menu for more information about the error.

status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

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 that produced the error or warning. The default is an empty string.
IPauseExecution.vi

Pauses the execution of a debugged application when a user clicks the Pause button in LabVIEW and sends an indSuspend indication from another IDE back to LabVIEW. Implementing this VI is optional.

<table>
<thead>
<tr>
<th>target ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>error in (no error)</td>
</tr>
<tr>
<td>error out</td>
</tr>
</tbody>
</table>

**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 (no error)** describes error conditions that occur before this VI runs. The default is **no error**. If an error occurred before this VI runs, the VI passes the **error in** value to **error out**. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

**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 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 ran, **error out** contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the **error out** indicator and select Explain Error from the shortcut menu for more information about the error.

**status** is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.
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 that produced the error or warning. The default is an empty string.

IRelease.vi

Closes all background VIs and releases 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, so this VI is called multiple times. 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. If you try to call IRelease before the download, debug, and/or run commands, debugging and running the application does not work.

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 (no error) describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to error out. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

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 that produced the error or warning. The default is an empty string.

does not contain information. If error in indicates that an error occurred before this VI ran, error out contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the error out indicator and select Explain Error from the shortcut menu for more information about the error.

status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

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 that produced the error or warning. The default is an empty string.

IResolveAddress.vi

Converts an address to a C filename/line number pair for dual debugging with another IDE. Implementing this VI is optional.

<table>
<thead>
<tr>
<th>target ID</th>
<th>address</th>
<th>status</th>
<th>code</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>TRUE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FALSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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 the absolute breakpoint address.

**error in (no error)** describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the **error in** value to **error out**. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

**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 that produced the error or warning. The default is an empty string.

**file name** is the corresponding C file for the address.

**line number** is the line number that corresponds to the address.

**error out** contains error information. If **error in** indicates that an error occurred before this VI ran, **error out** contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the **error out** indicator and select **Explain Error** from the shortcut menu for more information about the error.

**status** is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

**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 that produced the error or warning. The default is an empty string.

**IResolveSymbols.vi**

Resolves relative breakpoint, signal, and control addresses on targets that use loaders or have operating systems that support dynamic linking. You must implement this VI if the final linking happens on the embedded target. If not, implementing this VI is optional. The loader or OS must expose the ability to return the absolute address of the symbols in the symbol list. LabVIEW uses the information this plug-in VI returns to resolve all symbols in the debug database.

<table>
<thead>
<tr>
<th>target ID</th>
<th>symbol list</th>
<th>address list</th>
<th>error in (no error)</th>
<th>error out</th>
</tr>
</thead>
</table>

**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.

**error in (no error)** describes error conditions that occur before this VI runs. The default is **no error**. If an error occurred before this VI runs, the VI passes the **error in** value to **error out**. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

**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 that produced the error or warning. The default is an empty string.

address list is an array of absolute addresses of symbols in the symbol list.

error out contains error information. If error in indicates that an error occurred before this VI ran, error out contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the error out indicator and select Explain Error from the shortcut menu for more information about the error.

status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

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 that produced the error or warning. The default is an empty string.

ISetBreakpoints.vi

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

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 (no error) describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to error out. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

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 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 ran, error out contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the error out indicator and select Explain Error from the shortcut menu for more information about the error.

status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

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 that produced the error or warning. The default is an empty string.

IStop.vi

Aborts the execution of a debugged application when a user clicks the Abort Execution button. Implementing this VI is optional.

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 (no error)** describes error conditions that occur before this VI runs. The default is **no error**. If an error occurred before this VI runs, the VI passes the **error in** value to **error out**. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI runs, it runs normally and sets its own error status in **error out**. 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

- **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 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 ran, **error out** contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the **error out** indicator and select **Explain Error** from the shortcut menu for more information about the error.

- **status** is **TRUE** (X) if an error occurred or **FALSE** (checkmark) to indicate a warning or that no error occurred.
- **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 that produced the error or warning. The default is an empty string.
Understanding the LabVIEW Embedded LVM File

The LabVIEW map (LVM) file is an ASCII text file that the debug database generates. The debug database contains a map of all possible signals, breakpoints, controls, and indicators in the physical memory of the embedded target. The debug database also contains line numbers and information for each node on the block diagram. LabVIEW uses this information 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 resolves each object ID 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 an embedded application.

Note  The LVM file described here is not the same text-based measurement file (.lvm) that is used in VIs for Windows, but the file extensions are the same.

Format of the LVM File

The LVM file maps functions, signals, and front panel controls to absolute addresses that the LabVIEW debugger uses to debug built applications. The LVM file contains a VI section for each source VI and a single offset table section.

LVM files use the following format:

[vi name n]
Node entries for vi name n
Signal entries for vi name n
Control entries for vi name n
Location entries for vi name n
[offset_table]
Type and size of internal data structures
Example LVM File

[foo.vi]
SNode(10,11,11)\$527=0x16340
Node(13,14,17)\$563=0x16580
Node(14,15,17)\$570=0x165BC
SRN(17,13)\$627=0x168A4
Location.SNode(10,11,11)@[523..540]
Location.Node(13,14,17)@[559..565]
Location.Node(14,15,17)@[566..577]
Location.SRN(17,13)@[626..628]
SID_13908=0xB4520
SID_13740=0xB4530
CID_10132=0xA9290
CID_9512=0xA92A4
[bar.vi]
SNode(12,13,19)\$299=0x18EA4
Node(20,21,28)\$407=0x19AE8
SRN(28,20)\$472=0x19EF8
Location.SNode(12,13,19)@[298..333]
Location.Node(20,21,28)@[403..409]
Location.SRN(28,20)@[471..473]
SID_11792=0xB5418
CID_2492=0xA9444
CID_2048=0xA9458
[Offset_Table]
ArrayControlData=24
ArrayControlData.bStatic=64,8
ArrayControlData.dataType=160,32
ArrayControlData.elt_datatype=128,32
ArrayControlData.hDefaultValue=32,32
ArrayControlData.hValue=0,32
ArrayControlData.nDims=96,32
BigEndian=1
BooleanData=4
BooleanData.bDefaultVal=16,8
BooleanData.bInput=24,8
BooleanData.bStatic=0,8
BooleanData.bVal=8,8
ClusterControlData=152
ClusterControlData.bInput=1192,8
ClusterControlData.bStatic=1184,8
ClusterControlData.datatype=1152,32
ClusterControlData.pBigCls=96,32
ClusterControlData.pDefaultVal=32,32
ClusterControlData.pOldVal=0,32
ClusterControlData.pSmallCls=128,1024
ClusterControlData.pVal=64,32
EnumCtlData=20
EnumCtlData.bInput=136,8
EnumCtlData.bStatic=128,8
EnumCtlData.datatype=0,32
EnumCtlData.defaultVal=64,32
EnumCtlData.nValue=32,32
EnumCtlData.numItems=96,32
NumericData=24
NumericData.bInput=48,8
NumericData.bStatic=40,8
NumericData.datasize=32,8
NumericData.datatype=0,32
NumericData.pData=64,128
NumericData.padding1=56,8
PictInfo=52
PictInfo.bHidden=384,8
PictInfo.bounds=128,64
PictInfo.color=352,32
PictInfo.currentBackColor=224,32
PictInfo.currentForeColor=192,32
PictInfo.currentTextColor=256,32
PictInfo.currentX=64,32
PictInfo.currentY=96,32
PictInfo.dc=32,32
PictInfo.dirty=16,16
PictInfo.hDefaultValue=320,32
PictInfo.hPictCode=288,32
PictInfo.refcnt=0,16
ReleaseDebugMode=0
StringData=16
StringData.bStatic=64,8
StringData.bufferLen=32,32
StringData.len=0,32
StringData.pValue=96,32
StringData.padding=72,24
_ControlData=20
_ControlData.controlIdx=0,32
_ControlData.dataType=96,32
_ControlData.fieldType=128,32
_ControlData.hValue=64,32
Chapter 19 Understanding the LabVIEW Embedded LVM File

VI Section

Each source VI is a separate section in the VI section part of the LVM file. The following types of VIs are not included in the LVM file:

- Reentrant VIs
- VIs that you cannot debug, which occurs when you do not set the Execution:Allow Debugging VI property

SNode Entries

SNode(id1,id2,id3)@lineNumber=absolute address

Description:
Maps the LabVIEW diagram structure node to the line number in the C file and the absolute address. This entry determines the emulator breakpoint address when a user sets a breakpoint on the block diagram. This entry also locates and highlights the LabVIEW diagram structure when the executable hits the breakpoint.

Parameters:

id1, id2, id3 are the internal LabVIEW IDs that locate the LabVIEW diagram structure node on the block diagram.
lineNumber is the line number in the C file that implements the structure node.

absoluteAddress is the absolute address of the code that implements the structure node on the target.

**Example:**
SNode(10,11,11)@527=0x16340

**Node Entries**

Node(id1, id2, id3)@lineNumber=absoluteAddress

**Description:**
Maps the LabVIEW VI node to the line number in the C file and the absolute address. This entry determines the emulator breakpoint address when a user sets a breakpoint on the block diagram. This entry also locates and highlights the block diagram node when the executable hits the breakpoint.

**Parameters:**

id1, id2, id3 are the internal LabVIEW IDs that locate the LabVIEW node on the block diagram.

lineNumber is the line number in the C file that implements the node.

absoluteAddress is the absolute address of the code that implements the LabVIEW node on the target.

**Example:**
Node(13,14,17)@563=0x16580

**SRN Entries**

SRN(id1, id2)@lineNumber=absoluteAddress

**Description:**
Maps the LabVIEW self-referencing node to the line number in the C file and the absolute address. A self-referencing node is a node that maintains pointers to the controls, indicators, and constants on the block diagram. This entry determines the emulator breakpoint address when a user sets a breakpoint on the block diagram. This entry also locates and highlights the LabVIEW self-referencing node when the executable hits the breakpoint.
Parameters:
$id1, id2, id3$ are the internal LabVIEW IDs that locate the LabVIEW self-referencing node on the block diagram.

$lineNumber$ is the line number in the C file that implements the self-referencing node on the target.

$absoluteAddress$ is the absolute address of the code that implements the self-referencing node.

Example:
$SRN(17, 13)@627=0x168A4$

**Location.SNode Entries**

$Location.SNode(id1, id2, id3)@[startLine..endLine]$

Description:
Maps the LabVIEW diagram structure node to the first and last line in the generated C code that implements the structure. This entry locates the structure node when a user sets a breakpoint or single-steps in the generated C code. This entry is used only when you have integrated the LabVIEW debugger with an IDE.

Parameters:
$id1, id2, id3$ are the internal LabVIEW IDs that locate the LabVIEW diagram structure node on the block diagram.

$startLine$ is the first line number in the C file that implements the structure node.

$endLine$ is the last line number that implements the structure node.

Example:
$Location.SNode(10, 11, 11)@[523..540]$

**Location.Node Entries**

$Location.Node(id1, id2, id3)@[startLine..endLine]$

Description:
Maps the LabVIEW node to the first and last line in the generated C code that implements the node. This entry locates the LabVIEW node when a user sets a breakpoint or single-steps in the generated C code. This entry is used only when you have integrated the LabVIEW debugger with an IDE.
Parameters:
id1, id2, id3 are the internal LabVIEW IDs that locate the LabVIEW node on the block diagram.

startLine is the first line number in the C file that implements the LabVIEW node.

gendLine is the last line number that implements the LabVIEW node.

Example:
Location.Node(13,14,17)@[559..565]

Location.SRN Entries

Location.SRN(id1, id2)@[startLine..endLine]

Description:
Maps the self-referencing node to the first and last line in the C file. This entry locates the self-referencing node when a user sets a breakpoint or single-steps in the generated C code. This entry is used only when you have integrated the LabVIEW debugger with an IDE.

Parameters:
id1, id2, id3 are the internal LabVIEW IDs that locate the LabVIEW diagram self-referencing node on the block diagram.

startLine is the first line number in the C file that implements the self-referencing node.

gendLine is the last line number that implements the self-referencing node.

Example:
Location.SRN(17,13)@[626..628]

SID Entries

SID_signalId=absoluteAddress

Description:
Maps the block diagram signal ID to the absolute target memory address. This entry determines the absolute target address of the signal variable when a user sets a probe on the block diagram.

Parameters:
signalID is the internal LabVIEW signal ID that locates the signal on the block diagram.
absoluteAddress is the absolute address of the signal variable in the target memory.

Example:
SID_13908=0xB4520

**CID Entries**

CID_controlID=absoluteAddress

**Description:**
Maps the front panel control ID to the absolute target memory address. This entry determines the absolute target address of the control variable when a user opens a front panel window.

**Parameters:**
- controlID is the internal LabVIEW control ID that locates the control on the front panel window.
- absoluteAddress is the absolute address of the control variable in the target memory.

Example:
CID_10132=0xA9290

**Offset_Table Section**

The [Offset_Table] section defines sizes and offsets of internal LabVIEW Embedded Run-Time Library structures and other attributes. The LVM file contains a single [Offset_Table] section.

**ArrayControlData Entries**

ArrayControlData=structureSize
ArrayControlData.bStatic=memberOffset, memberSize
ArrayControlData.dataType=memberOffset, memberSize
ArrayControlData.elt_datatype=memberOffset, memberSize
ArrayControlData.hValue=memberOffset, memberSize
ArrayControlData.hDefaultValue=memberOffset, memberSize
ArrayControlData.nDims=memberOffset, memberSize

**Description:**
Describes the layout of the ArrayControlData structure in the target memory. The LabVIEW debugger uses the ArrayControlData structure information to access the front panel array control data structures.
**Parameters:**

*structureSize* is the size of the ArrayControlData structure in bytes.

*memberOffset* is the offset of the structure member variable in bits.

*memberSize* is the size of the structure member variable in bits.

**Location:**
LVArrayControl.h

**Example:**

```
ArrayControlData=24
ArrayControlData.bStatic=64,8
ArrayControlData.dataType=160,32
ArrayControlData.elt_datatype=128,32
ArrayControlData.hDefaultValue=32,32
ArrayControlData.hValue=0,32
ArrayControlData.nDims=96,32
```

**Endianness Entry**

```
BigEndian=bigEndian
```

**Description:**
Indicates the endianness of the target.

**Parameters:**

*bigEndian* determines the endianness of the target.

0 = little endian

1 = big endian

**Example:**

```
BigEndian=1
```

**BooleanData Entries**

```
BooleanData=structureSize
BooleanData.bDefaultVal=memberOffset,memberSize
BooleanData.bInput=memberOffset,memberSize
BooleanData.bStatic=memberOffset,memberSize
BooleanData.bVal=memberOffset,memberSize
```
Description:
Describes the layout of the BooleanData structure in the target memory. The LabVIEW debugger uses the BooleanData structure information to access the front panel Boolean control data structures.

Parameters:
structureSize is the size of the BooleanData structure in bytes.
memberOffset is the offset of the structure member variable in bits.
memberSize is the size of the structure member variable in bits.

Location:
LVBoolean.h

Example:
BooleanData=4
BooleanData.bDefaultVal=16,8
BooleanData.bInput=24,8
BooleanData.bStatic=0,8
BooleanData.bVal=8,8

ClusterControlData Entries
ClusterControlData=structureSize
ClusterControlData.bInput=memberOffset,memberSize
ClusterControlData.bStatic=memberOffset,memberSize
ClusterControlData.datatype=memberOffset,memberSize
ClusterControlData.pBigCls=memberOffset,memberSize
ClusterControlData.pDefaultVal=memberOffset,memberSize
ClusterControlData.pOldVal=memberOffset,memberSize
ClusterControlData.pSmallCls=memberOffset,memberSize
ClusterControlData.pVal=memberOffset,memberSize

Description:
Describes the layout of the ClusterControlData structure in the target memory. The LabVIEW debugger uses the ClusterControlData structure information to access the front panel cluster control data structures.

Parameters:
structureSize is the size of the ClusterControlData structure in bytes.
memberOffset is the offset of the structure member variable in bits.
memberSize is the size of the structure member variable in bits.
Location:
LVClusterControl.h

Example:
ClusterControlData=152
ClusterControlData.bInput=1192,8
ClusterControlData.bStatic=1184,8
ClusterControlData.datatype=1152,32
ClusterControlData.pBigCls=96,32
ClusterControlData.pDefaultVal=32,32
ClusterControlData.pOldVal=0,32
ClusterControlData.pSmallCls=128,1024
ClusterControlData.pVal=64,32

EnumCtlData Entries
EnumCtlData=structureSize
EnumCtlData.bInput=memberOffset,memberSize
EnumCtlData.bStatic=memberOffset,memberSize
EnumCtlData.datatype=memberOffset,memberSize
EnumCtlData.defaultVal=memberOffset,memberSize
EnumCtlData.nValue=memberOffset,memberSize
EnumCtlData.numItems=memberOffset,memberSize

Description:
Describes the layout of the EnumCtlData structure in the target memory. The LabVIEW debugger uses the EnumCtlData structure information to access the front panel enumerated type control data structures.

Parameters:
structureSize is the size of the EnumCtlData structure in bytes.
memberOffset is the offset of the structure member variable in bits.
memberSize is the size of the structure member variable in bits.

Location:
LVEnumCtl.h

Example:
EnumCtlData=20
EnumCtlData.bInput=136,8
EnumCtlData.bStatic=128,8
EnumCtlData.datatype=0,32
EnumCtlData.defaultVal=64,32
EnumCtlData.nValue=32,32
EnumCtlData.numItems=96,32

**NumericData Entries**

NumericData=structureSize
NumericData.bInput=memberOffset,memberSize
NumericData.bStatic=memberOffset,memberSize
NumericData.datasize=memberOffset,memberSize
NumericData.datatype=memberOffset,memberSize
NumericData.pData=memberOffset,memberSize
NumericData.padding1=memberOffset,memberSize

**Description:**

Describes the layout of the NumericData structure in the target memory. The LabVIEW debugger uses the NumericData structure information to access the front panel numeric control data structures.

**Parameters:**

*structureSize* is the size of the NumericData structure in bytes.

*memberOffset* is the offset of the structure member variable in bits.

*memberSize* is the size of the structure member variable in bits.

**Location:**

LVNumeric.h

**Example:**

NumericData=24
NumericData.bInput=48,8
NumericData.bStatic=40,8
NumericData.datasize=32,8
NumericData.datatype=0,32
NumericData.pData=64,128
NumericData.padding1=56,8

**PictInfo Entries**

PictInfo=structureSize
PictInfo.bHidden=structureSize
PictInfo.bounds=structureSize
PictInfo.color=structureSize
PictInfo.currentBackColor=structureSize
PictInfo.currentForeColor=structureSize
PictInfo.currentTextColor=structureSize
PictInfo.currentX=structureSize
PictInfo.currentY=structureSize
PictInfo.dc=structureSize
PictInfo.dirty=structureSize
PictInfo.hDefaultValue=structureSize
PictInfo.hPictCode=memberOffset,memberSize
PictInfo.refcnt=memberOffset,memberSize

Description:
Describes the layout of the PictInfo structure in the target memory. The LabVIEW debugger uses the PictInfo structure information to access the front panel array picture control data structures.

Parameters:
structureSize is the size of the PictInfo structure in bytes.

memberOffset is the offset of the structure member variable in bits.

memberSize is the size of the structure member variable in bits.

Location:
PictInfo.h

Example:
PictInfo=52
PictInfo.bHidden=384,8
PictInfo.bounds=128,64
PictInfo.color=352,32
PictInfo.currentBackColor=224,32
PictInfo.currentForeColor=192,32
PictInfo.currentTextColor=256,32
PictInfo.currentX=64,32
PictInfo.currentY=96,32
PictInfo.dc=32,32
PictInfo.dirty=16,16
PictInfo.hDefaultValue=320,32
PictInfo.hPictCode=288,32
PictInfo.refcnt=0,16
Debugging Mode Entry
ReleaseDebugMode=debugMode

Description:
Indicates the debugging mode for the built application. An application built in the release debugging mode uses a single breakpoint and does not require the .text section to be located in RAM. A typical use case for this entry is debugging an application running in ROM and/or using hardware breakpoints.

Parameters:
debugMode determines the debugging mode.

0 = standard debugging mode
1 = release debugging mode

Example:
ReleaseDebugMode=0

StringData Entries
StringData=structureSize
StringData.bStatic=memberOffset,memberSize
StringData.bufferLen=memberOffset,memberSize
StringData.len=memberOffset,memberSize
StringData.pValue=memberOffset,memberSize
StringData.padding=memberOffset,memberSize

Description:
Describes the layout of the StringData structure in the target memory. The LabVIEW debugger uses the StringData structure information to access the front panel string control data structures.

Parameters:
structureSize is the size of the StringData structure in bytes.

memberOffset is the offset of the structure member variable in bits.

memberSize is the size of the structure member variable in bits.

Location:
LVString.h
Chapter 19  Understanding the LabVIEW Embedded LVM File

Example:
StringData=16
StringData.bStatic=64,8
StringData.bufferLen=32,32
StringData.len=0,32
StringData.pValue=96,32
StringData.padding=72,24

_ControlData Entries

_ControlData=structureSize
_ControlData.controlIdx=memberOffset,memberSize
_ControlData.dataType=memberOffset,memberSize
_ControlData.fieldType=memberOffset,memberSize
_ControlData.hValue=memberOffset,memberSize
_ControlData.parentIdx=memberOffset,memberSize

Description:
Describes the layout of the _ControlData structure in the target memory. The LabVIEW debugger uses the _ControlData structure information to access the list of front panel control data structures.

Parameters:
structureSize is the size of the _ControlData structure in bytes.

memberOffset is the offset of the structure member variable in bits.

memberSize is the size of the structure member variable in bits.

Location:
LVCCG.h

Example:
_ControlData=20
_ControlData.controlIdx=0,32
_ControlData.dataType=96,32
_ControlData.fieldType=128,32
_ControlData.hValue=64,32
_ControlData.parentIdx=32,32
_LVGraphData Entries

_description:
Describes the layout of the _LVGraphData structure in the target memory. The LabVIEW debugger uses the _LVGraphData structure information to access the front panel graph control data structures.

_parameters:
structureSize is the size of the _LVGraphData structure in bytes.
memberOffset is the offset of the structure member variable in bits.
memberSize is the size of the structure member variable in bits.

_location:
LVGraph.h

_example:
_LVGraphData=16
_LVGraphData.bStatic=64,8
_LVGraphData.datatype=0,32
_LVGraphData.dtDataIn=32,32
_LVGraphData.vhDataIn=96,32

_OCDI_Alignment Entries

_description:
Describes the layout of the _OCDI_Alignment structure in the target memory. The LabVIEW debugger uses the _OCDI_Alignment structure information to access the cluster signal and control cluster member data structures.
**Parameters:**

*structureSize* is the size of the _OCDI_Alignment structure in bytes.

*memberOffset* is the offset of the structure member variable in bits.

*memberSize* is the size of the structure member variable in bits.

**Location:**

LVCCG.h

**Example:**

```markdown
_OCDI_Alignment=16
_OCDI_Alignment.el_1=0,8
_OCDI_Alignment.el_2=64,64
```

**_PDAArr Entries**

```markdown
_PDAArr=structureSize
_PDAArr.arr=memberOffset,memberSize
_PDAArr.datatype=memberOffset,memberSize
_PDAArr.fill=memberOffset,memberSize
_PDAArr.refcnt=memberOffset,memberSize
_PDAArr.staticArray=memberOffset,memberSize
```

**Description:**

Describes the layout of the _PDAArr structure in the target memory. The LabVIEW debugger uses the _PDAArr structure information to access the front panel array signal and control data structures.

**Parameters:**

*structureSize* is the size of the _PDAArr structure in bytes.

*memberOffset* is the offset of the structure member variable in bits.

*memberSize* is the size of the structure member variable in bits.

**Location:**

CCGArrSupport.h

**Example:**

```markdown
_PDAArr=12
_PDAArr.arr=64,32
_PDAArr.datatype=0,32
_PDAArr.fill=56,8
_PDAArr.refcnt=32,16
_PDAArr.staticArray=48,8
```
Chapter 19 Understanding the LabVIEW Embedded LVM File

_PDAStr Entries
_PDAStr=structureSize
_PDAStr.len=memberOffset,memberSize
_PDAStr.refcnt=memberOffset,memberSize
_PDAStr.staticStr=memberOffset,memberSize
_PDAStr.str=memberOffset,memberSize

Description:
Describes the layout of the _PDAStr structure in the target memory. The LabVIEW debugger uses the _PDAStr structure information to access the front panel string signal and control data structures.

Parameters:
structureSize is the size of the _PDAStr structure in bytes.
memberOffset is the offset of the structure member variable in bits.
memberSize is the size of the structure member variable in bits.

Location:
CCGSTrSupport.h file

Example:
_PDAStr=12
_PDAStr.len=32,32
_PDAStr.refcnt=0,16
_PDAStr.staticStr=16,16
_PDAStr.str=64,16

Generating the LVM File

Complete the following steps to generate the LVM file, which is generated during the build process into the directory you specify in EMB_Debug_DD_Save.vi.

1. Call EMB_Debug_DD_Init.vi to initialize the debug database and erase records from any previous builds.
2. Call EMB_Debug_DD_ParseCFile.vi for every generated C file. This VI performs the following:
   • Opens the generated C file.
   • Parses the C file for OCDI comments.
   • Inserts breakpoint, signal, and control records into the debug database.
Chapter 19  Understanding the LabVIEW Embedded LVM File

3. Resolve breakpoint addresses and symbols. For all breakpoints, call EMB_Debug_DD_GetBP.vi in a loop to map the breakpoint line numbers into absolute target addresses. Call EMB_Debug_DD_SetAbsoluteBP_Address.vi to update the absolute address of each breakpoint record.

4. Resolve memory records. For all memory records, call EMB_Debug_DD_GetMem.vi in a loop to map the memory line numbers into absolute target addresses. Call EMB_Debug_DD_SetAbsoluteMemAddress.vi to update the absolute address of each memory record.

Note: You must resolve all breakpoints, symbols, and memory records when an embedded VI is built into an embedded application. Resolving is the most complicated part of generating the LVM file because it involves integration with your third-party resolver that can parse the built application and resolve all memory symbols and map breakpoint line numbers to absolute target addresses.

5. Use EMB_Debug_DD_GetSetAttrib_Numeric.vi and EMB_Debug_DD_GetSetAttrib_Numeric_Pair.vi to set all of the attributes in the [Offset_Table] section.

Note: You must set all attributes in the [Offset_Table] section to ensure the LabVIEW debugger correctly dereferences the pointers.

6. Call EMB_Debug_DD_Save.vi to save the debug database to the LVM file.

Plug-In VIs for the Debug Database

The following VIs are the plug-in VIs for the debug database:
- EMB_Debug_DD_GetBP.vi
- EMB_Debug_DD_GetMem.vi
- EMB_Debug_DD_GetSetAttrib.vi
- EMB_Debug_DD_Init.vi
- EMB_Debug_DD_ParseCFile.vi
- EMB_Debug_DD_Save.vi
- EMB_Debug_DD_SetAbsoluteBPAddress.vi
- EMB_Debug_DD_SetAbsoluteMemAddress.vi
Chapter 19  Understanding the LabVIEW Embedded LVM File

**EMB_Debug_DD_GetBP.vi**

Queries the debug database for the breakpoint record with a matching breakpoint ID. You must call this VI in a loop when all breakpoint records are in place to resolve the line numbers of all breakpoints.

Place this VI in a While Loop and wire **Breakpoint ID** to the iteration (i) terminal of the While Loop. Terminate the loop when **EMB_Debug_DD_GetBP.vi** returns error code 5001 (ErrNotFound). You must resolve **Breakpoint Line** to the absolute target address and call **EMB_Debug_DD_SetAbsoluteBPAddress.vi** to update the absolute address of the resolved breakpoint record.

You can ignore the following outputs:

- **Breakpoint Index**
- **Breakpoint Actual Index**
- **Breakpoint Step Out Index**
- **Breakpoint Type**
- **Sub VI Name**
- **Breakpoint Relative Address**
- **Breakpoint Address**

**Breakpoint ID** is the incrementing zero-based breakpoint record.

**target ID** is the ID of the target used to support concurrent debugging sessions. Wire 0 to this input when you call this VI from a build plug-in VI.

**error in (no error)** describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the **error in** value to **error out**. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this
information to decide if functionality if you want to bypass functionality in
the event of errors from other VIs.

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 that produced the error or warning. The default is an empty string.

Breakpoint Index is an internal LabVIEW breakpoint indicator. You can ignore this output.

Breakpoint Actual Index is an internal LabVIEW breakpoint indicator. You can ignore this output.

Breakpoint Step Out Index is an internal LabVIEW breakpoint indicator. You can ignore this output.

VI Name is the name of the VI for which this breakpoint belongs.

Breakpoint Type is an internal LabVIEW breakpoint type indicator. Possible values for this output are Node, SNode, IUse, AfterSNode, and SRN. You can ignore this output.

Breakpoint Line is the line in the generated C file that contains the breakpoint OCDI comment.

Sub VI Name is the name of the subVI. This output is valid only when Breakpoint Type is IUse. You can ignore this output.

error out contains error information. If error in indicates that an error occurred before this VI ran, error out contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the error out indicator and select Explain Error from the shortcut menu for more information about the error.

status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.
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 that produced the error or warning. The default is an empty string.

Breakpoint Relative Address is the relative address of the breakpoint address. This output is used only for relocatable modules when the absolute address cannot be resolved at build time. The format of the relative address is <symbolName>+0x<Offset>.

Breakpoint Address is the absolute address for the breakpoint. For relocatable modules, set this output to 0. You can ignore this output.

EMB_Debug_DD_GetMem.vi

Queries the debug database for the memory record with a matching memory ID. You must call this VI in a loop when all memory records are in place to resolve the relative addresses of all signals and controls.

Place this VI in a While Loop and wire Memory ID to the iteration (i) terminal of the While Loop. Terminate the loop when EMB_Debug_DD_GetMem.vi returns error code 5001 (ErrNotFoud). You must resolve Relative Address to the absolute target address and call EMB_Debug_DD_SetAbsoluteMemAddress.vi to update the absolute address of the resolved memory record.

target ID is the ID of the target used to support concurrent debugging sessions. Wire 0 to this input when you call this VI from a build plug-in VI.

Memory ID is the zero-based memory record ID.
	error in (no error) describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to error out. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

**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 that produced the error or warning. The default is an empty string.

**VI Name** is the name of the VI for which this breakpoint belongs.

**Signal/Control ID** is an internal LabVIEW signal ID or control ID.

**Memory Type** is the type of memory record. Possible values are **Signal** or **Control**.

**error out** contains error information. If **error in** indicates that an error occurred before this VI ran, **error out** contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the **error out** indicator and select **Explain Error** from the shortcut menu for more information about the error.

**status** is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

**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 that produced the error or warning. The default is an empty string.

**Relative Address** is the relative address of the signal or control. The format of the relative address is `<symbolName>+0x<Offset in hex>`.

**Address** is the absolute address for the control or signal.
EMB_Debug_DD_Init.vi

Initializes the debug database. This VI deletes all of the records from the debug database. Call this VI before you call the EMB_Debug_DD_ParseCFile.vi.

<table>
<thead>
<tr>
<th>target ID</th>
<th>error in (no error)</th>
<th>error out</th>
</tr>
</thead>
</table>

**target ID** is the ID of the target used to support concurrent debugging sessions. Wire 0 to this input when you call this VI from a build plug-in VI.

**error in (no error)** describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to error out. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

**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 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 ran, error out contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the error out indicator and select Explain Error from the shortcut menu for more information about the error.

**status** is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.
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 that produced the error or warning. The default is an empty string.

**EMB_Debug_DD_ParseCFile.vi**

Parses the C source file for OCDI comments and appends new records to the debug database. Call this VI for each C file the LabVIEW C Code Generator generates. You resolve all symbols and breakpoint line numbers before the debug database is saved in the LVM file.

**target ID** is the ID of the target used to support concurrent debugging sessions. Wire 0 to this input when you call this VI from a build plug-in VI.

**Path** is the path to the generated C file.

**error in (no error)** describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to error out. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

**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 that produced the error or warning. The default is an empty string.

**VI Name** is the name of the VI.

error out contains error information. If error in indicates that an error occurred before this VI ran, error out contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the error out indicator and select Explain Error from the shortcut menu for more information about the error.

status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

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 that produced the error or warning. The default is an empty string.

**EMB_Debug_DD_Save.vi**

Serializes the debug database to the LVM file. You must call this VI when all symbols are resolved and the offset table attributes are set.

**target ID** is the ID of the target used to support concurrent debugging sessions. Wire 0 to this input when you call this VI from a build plug-in VI.

Path is the path to the generated C file.

error in (no error) describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to error out. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this
information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

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 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 ran, error out contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the error out indicator and select Explain Error from the shortcut menu for more information about the error.

status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

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 that produced the error or warning. The default is an empty string.

**EMB_Debug_DD_SetAbsoluteBPAddress.vi**

Sets the absolute address of a given breakpoint record.

**target ID** is the ID of the target used to support concurrent debugging sessions. Wire 0 to this input when you call this VI from a build plug-in VI.

**Breakpoint ID** is the incrementing zero-based breakpoint record.

**Breakpoint Address** is the absolute address for the breakpoint.
**error in (no error)** describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the **error in** value to **error out**. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI runs, it runs normally and sets its own error status in **error out**. 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

**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 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 ran, **error out** contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the **error out** indicator and select **Explain Error** from the shortcut menu for more information about the error.

**status** is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

**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 that produced the error or warning. The default is an empty string.
EMB_Debug_DD_SetAbsoluteMemAddress.vi

Sets the absolute address of a given memory record.

target ID is the ID of the target used to support concurrent debugging sessions. Wire 0 to this input when you call this VI from a build plug-in VI.

Memory ID is the zero-based memory record ID.

Address is the absolute address for the memory record.

error in (no error) describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to error out. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

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 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 ran, error out contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the error out indicator and select Explain Error from the shortcut menu for more information about the error.
status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

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 that produced the error or warning. The default is an empty string.
You can integrate side-by-side debugging sessions with LabVIEW and another Integrated Development Environment (IDE) if you fully implement OCDI debugging for your target. Traditional embedded developers can use their C knowledge to debug LabVIEW embedded applications 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.

Refer to Chapter 18, *On Chip Debugging*, for information about implementing OCDI debugging.

**Additional Debugging Features with IDE Integration**

IDE integration provides 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. Selecting this shortcut menu option displays the corresponding C code in the IDE.
Example of Code Comments

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. The C parser parses the comments and adds the comments to the debug database. The following code sample shows an example of these comments:

```c
hA8EA7A0->sCA8/* n: MultiplyConstant */ = 5.0000000000000000000E+2;
OCDI_BEGIN_NODE(profileInfo_simple_math, 0, 2) /*
OCDI::Begin::Node(1, 2, 3) */
/**/
/* Multiply */
/**/
OCDI_CHECK_POINT(simple_math_OCDIFlag[0] & 0x1,
&s_simple_math_OCDIFlag, 0)/* OCDI::BreakPoint::Node(1,
2, 3) */
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,
&s_simple_math_OCDIFlag, 1)/* OCDI::BreakPoint::Node(2,
3, 3) */
OCDI::End::Node(2, 3, 3) */
if (NumberOut__616) {
*NumberOut__616 = hA8EA7A0->sA08/* n: Divide: 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;
```
Example IDE Integration Implementation

The Unix Console example target implements IDE integration with the open source Eclipse framework. Refer to the following directory for the Unix Console and Eclipse example implementation:

Note 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 Microprocessor SDK:

- Eclipse SDK 3.2.2
- C/C++ Development Tools for Eclipse (CDT) SDK 3.1.2 or later
- LabVIEW Embedded Eclipse plug-in
- Cygwin and GNU tools

Eclipse


Note Version 3.2.2 is an archived version.

C/C++ Development Tools for Eclipse (CDT) SDK

After you install Eclipse, you must install the C/C++ Development Tools for Eclipse (CDT) SDK plug-in. Complete the following steps to install the CDT SDK.

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. Select Callisto Discovery Site and click the Finish button.
4. Expand C and C++ Development.
5. Select Eclipse C/C++ Development Tool 3.1.2 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 Eclipse SDK workspace.

**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. Click the OK button on the Edit Local Site dialog box.
6. Click the Finish button.
7. Select LabVIEW Embedded Eclipse plug-in 1.0.1 and click the Next button.
8. Review the license agreement and click the Next button.
9. Click the Finish button.
10. Click the Install All button.
11. Click the Yes button to restart the workbench.

**Cygwin 1.5.23-2 or later**

Refer to the Cygwin Web site at [www.cygwin.com](http://www.cygwin.com) for download and installation instructions.

Install the following GNU tools in Cygwin:

- binutils 20060817-1 or later
- gcc 3.4.4 or later
- gdb 20060706-2 or later
- make 3.81 or later
Implementation Example

The Unix Console example implementation uses TCP/IP to communicate between LabVIEW and Eclipse. If the communication fails, you see a TCP Write or a TCP Read error. Restart Eclipse and remove 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 you pause the application being debugged. The application might fail when you debug a running embedded application using IDE integration with Eclipse because Eclipse and gdb slow down the update rate for controls, indicators, and probes. LabVIEW must interrupt the application before reading from or writing to the target memory. Single-stepping or pausing the application is not an issue.

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 the shortcut menu item and 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:

```
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.

**Note** If you use the Unix Console implementation as a starting point, use the Documentation page in the VI Properties dialog box to change the context help for each control.
IDE Integration Plug-In VIs

To implement IDE integration with LabVIEW, you must modify and implement the following OCDI debugging plug-in VIs:

- IInitIndication
- IGetIndication
- IShowCrash
- IShowSource
- (Optional) IResolveAddress

**IInitIndication.vi**

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 (no error)** describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to **error out**. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.
- **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 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 ran, error out contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the error out indicator and select Explain Error from the shortcut menu for more information about the error.

status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

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 that produced the error or warning. The default is an empty string.

IGetIndication.vi

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 (no error) describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to error out. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

**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 that produced the error or warning. The default is an empty string.

**indication type** is the type of event.

**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 ran, **error out** contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the **error out** indicator and select **Explain Error** from the shortcut menu for more information about the error.
status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

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 that produced the error or warning. The default is an empty string.

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.

IShowCrash.vi

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 (no error) describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to error out. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

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 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 ran, error out contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the error out indicator and select Explain Error from the shortcut menu for more information about the error.

status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

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 that produced the error or warning. The default is an empty string.

IResolveAddress.vi

Converts an address to a C filename/line number pair for dual debugging with another IDE. Implementing this VI is optional.
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 the absolute breakpoint address.

error in (no error) describes error conditions that occur before this VI runs. The default is no error. If an error occurred before this VI runs, the VI passes the error in value to error out. This VI runs normally only if no error occurred before this VI runs. If an error occurs while this VI 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. You can use this information to decide if functionality if you want to bypass functionality in the event of errors from other VIs.

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 that produced the error or warning. The default is an empty string.

file name is the corresponding C file for the address.

line number is the line number that corresponds to the address.

error out contains error information. If error in indicates that an error occurred before this VI ran, error out contains the same error information. Otherwise, it describes the error status that this VI produces. Right-click the error out indicator and select Explain Error from the shortcut menu for more information about the error.

status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.

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 that produced the error or warning. The default is an empty string.
Overriding SubVI Calls with C Function Calls to Improve Performance

Funclist files contain entries that enable you to override subVI calls with C function calls. For example, the Semaphore VIs use a CIN, which embedded targets do not support. For embedded targets, the Semaphore VIs have an alternate implementation that makes direct calls to a C function instead. Funclist entries is how you provide an alternative implementation. Using Funclist removes the overhead of subVI calls and can produce substantial performance improvements in LabVIEW embedded applications. SubVI calls have overhead to support multithreading, reentrancy, front panel controls and indicators, and so on.

Funclist 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 for internal use only. Funclist entries 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 Entries section in this chapter for information about the formatting requirements.
Format of Funclist Entries

Funclist files are XML files that contain 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. The Microprocessor SDK includes common Funclist XML files that all example targets use. The common Funclist files are in the following directory:

```
\labview\CCodeGen\Funclist
```

You also can supplement Funclist with your own target-specific Funclist files.

Each Funclist entry must contain the following three tags:

- **<VI Name>**
- **<Function>**
- **<Feature>**

**<VI Name>**

The first line of a Funclist entry is the name of the VI you are overriding enclosed in a `<VI Name>` tag. 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.

**<Function>**

The second line of a Funclist entry contains the line of C code that replaces the subVI call enclosed in a `<Function>` tag. If you call a function, the second line contains the function parameters. You can pass the parameters directly or with `ArgList`. National Instruments strongly recommends that you pass the parameters directly.

**Passing Parameters Directly**

Passing parameters for C function calls, which is the second line in Funclist entries, 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 LabVIEW C Code Generator passes the parameter. Refer to the following bullet for more information about how the LabVIEW C Code Generator passes the parameter. \( n \) is the number of the input or output on the connector pane.

**Note** The numbering on the connector pane is not intuitive. You might have to use trial and error to find the correct number.

- Use the following characters to indicate how the LabVIEW C Code Generator passes the parameter:
  - \( I \) 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 \( @I3(\text{Null}) \).

### Passing Parameters by ArgList

**Note** National Instruments strongly recommends you pass parameters directly rather than use 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:

```plaintext
labview\CCodeGen\include\blockdiagram\LVCCG.h
```

**<Feature>**

The third line of a Funclist entry contains a string that describes the feature you are overriding enclosed in a `<Feature>` tag. You can use any string for the third line as long as the string is a valid C identifier and the string is unique to the other Feature strings in the Funclist file.
Implementing Target-Specific Funclist Files

The Microprocessor SDK includes common Funclist XML files that all example targets use. The common Funclist files are in the following directory:

labview\CCodeGen\Funclist

You also can supplement Funclist with your own target-specific Funclist files. Target-specific Funclist files override the common Funclist files. Target-specific Funclist files override subVIs per target so you can have different overrides if you implement more than one target.

Entries in target-specific Funclist files must follow the formatting requirements, but the XML filenames do not have formatting requirements. Place all target-specific Funclist files in a Funclist folder in your target directory. When LabVIEW launches, LabVIEW loads the common Funclist files in alphabetical order. When you create an application instance for your target, LabVIEW loads any target-specific Funclist files in alphabetical order. To override an entry in the common Funclist files, use the same name in the <VI Name> tag in your target-specific Funclist entry.

Debugging and Troubleshooting Funclist Entries

If you override a subVI call with a Funclist entry, you must debug the subVI in C using your toolchain. You cannot debug Funclist subVIs in LabVIEW.

If you create a Funclist entry and the generated C code is still generating the subVI call instead of the C function call, verify you restarted LabVIEW to reload the Funclist files. After you restart LabVIEW, verify the subVI name in the first line of the Funclist entry.

Refer to the <VI Name> section in this chapter for information about how to specify VI names in Funclist entries.

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 the second line of the Funclist entry.

Refer to the <Function> section in this chapter for information about the required formatting for passing parameters in a C function call.
If you receive compile errors in LVFuncsUsed.h, verify that the third line in the Funclist entry is a valid C identifier.

Refer to the <Feature> section in this chapter for information about the string on the third line.

Note Funclist files contain little error reporting.

Implementation Example

National Instruments has implemented many subVIs for the Microprocessor SDK example targets using Funclist files, which are located in the following directory:

labview\CCodeGen\Funclist
Static Memory Allocation

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

If you build an embedded application using the static memory model, the generated code and the LabVIEW 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 for an application that uses static memory allocation, the Run button on the block diagram breaks and an error message indicates the problem.

Supporting a Static Memory Model

To support a static memory model, you must compile all of the C files with the CStatic compiler directive. You must set the memory model parameter in the LabVIEW C Code Generator to 1.

You must define the CStatic compiler directive for all LabVIEW C Code Run-Time Library files and for all source files in the embedded project. When you define CStatic, the LabVIEW C Code Generator defines
dynamic memory management calls, such as malloc, realloc, and free, as no-ops. Additional function arguments are produced that are pointers to global variables the LabVIEW C Code Generator declares. The functions use macros to use the global data in these extra function arguments instead of calling malloc.

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 compile both a static memory and a dynamic memory version of the pre-built run-time libraries and link 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.

As part of creating separate targets, you must create separate TargetSyntax.xml files for each target. Use the TargetSyntax.xml file located in the following directory as a template for dynamic memory targets:

labview\Targets\NI\Embedded\common

Use the TargetSyntax.xml file 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 ability to set a dimension size for static memory targets. Place a checkmark in the Enable Set Dimension Size for arrays (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 7, Creating a New Embedded Target, for more information about the Target Editor.
Limitations of the Static Memory Model

Static memory allocation presents some limitations, some of which you must define for your users.

OEM-Defined Limitations

You define certain static memory model limitations with the following macros:

- 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.

User Limitations

Embedded VIs for static memory model targets have the following limitations:

- 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. Target syntax checking 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 **Configure Timed Loop** dialog box. Open the **Configure Timed Loop** dialog box by right-clicking the Timed Loop on the block diagram and selecting **Configure Timed Loop** from the shortcut menu. Remove the checkmark from the **Use terminal** checkbox. Enter 1 in the **Period** box.
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 RTFIFOCreate 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.
- TCP functions and UDP functions are not supported.
- The LabVIEW Analysis Library is not supported.
Using the Target Server

Use the Target Server to download, execute, and monitor an embedded application over a TCP connection. The Target Server provides a way to download and run an embedded application without requiring both a serial and Ethernet connection. To use the Target Server with an embedded target, the target must have a TCP connection, a TFTP client, and a telnet server. You use a configuration cluster in the Target Server Utilities VIs to define command strings and corresponding regular expressions to parse the command output.

If you implement a Target Server, users open it by right-clicking a build specification under your target in the Project Explorer window.

Target Server Window

The Target Server window displays real-time status of the following:

- Telnet connection to the target
- Status of the embedded application running on the target
- Current processes running on the target

LabVIEW displays the Target Server window when the embedded provider calls the following utility VIs, which are located on the Target Server Utilities palette:

- TargetServer_DownloadCmd
- TargetServer_RunCmd
- TargetServer_ShowWindowCmd
Target Server Daemon

The Target Server daemon maintains an open telnet connection to the target, monitors the status of the running embedded application, and updates the list of processes currently running on the embedded target. The Target Server daemon automatically launches the first time you call the download or run command.

The Target Server Utilities VIs provides an interface to the Target Server daemon.

**Note**  All embedded applications that you launch using the Target Server close when the Target Server shuts down.

Telnet Client

Use the Telnet Utilities VIs to implement a simple telnet client.

Implementation Example

The Freescale M5329EVB example target uses the Target Server. The following directory contains the plug-in VIs for the Freescale M5329EVB example target:

```
labview\Targets\NI\Embedded\unix\m5329evb\unix_m5329evb_LEP_TargetPlugin
```
Technical Support and Professional Services

As an OEM/VAR, you must consider providing technical support for your product if you decide to bundle and resell your target. Customers will contact you to purchase your target, obtain a valid serial number, and receive technical support for your target. You are the expert for your target, but National Instruments can help you with general support issues.

In addition to the Microprocessor documentation, you also have access to the LabVIEW Embedded Discussion Forum and training.

- The LabVIEW Embedded Discussion Forum provides a discussion forum for the Microprocessor SDK and other LabVIEW Embedded products. You can access the forum by visiting ni.com/info and typing edmforum.
- On site training at National Instruments corporate headquarters or Web-based training available upon request.

Visit the following sections of the award-winning National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Technical support resources at ni.com/support include the following:
  - **Self-Help Technical Resources**—For answers and solutions, visit ni.com/support for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on. Registered users also receive access to the NI Discussion Forums at ni.com/forums. NI Applications Engineers make sure every question submitted online receives an answer.
  - **Standard Service Program Membership**—This program entitles members to direct access to NI Applications Engineers via phone and email for one-to-one technical support as well as exclusive access to on demand training modules via the Services Resource Center. NI offers complementary membership for a full
year after purchase, after which you may renew to continue your benefits.

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

- **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.