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 0 662 45 79 90 0, Belgium 32 0 2 757 00 20, Brazil 55 11 3262 3599, Canada 800 433 3488, China 86 21 6555 7838, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 385 0 9 725 725 11, France 33 0 1 48 14 24 24, Germany 49 0 89 741 31 30, India 91 80 51190000, Israel 972 0 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970, 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 7 095 783 68 51, Singapore 1800 226 5886, Slovenia 386 3 425 4200, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 0 8 587 895 00, Switzerland 41 56 200 51 51, Taiwan 886 02 2377 2222, Thailand 662 992 7519, 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 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. EXCEPT AS SPECIFIED HERIN, 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 INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner’s failure to follow the National Instruments installation, operation, or maintenance instructions; owner’s modification of the product; owner’s abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

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

Trademarks
National Instruments, NI, ni.com, and LabVIEW are trademarks of National Instruments Corporation. Refer to the Terms of Use section on ni.com/legal for more information about National Instruments trademarks.

Members of the National Instruments Alliance Partner Program are business entities independent from National Instruments and have no agency, partnership, or joint-venture relationship with National Instruments.

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

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

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

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR 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 INJURY, DEATH, OR OTHER DAMAGE, 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 THAT NATIONAL INSTRUMENTS SOFTWARE AND HARDWARE DIFFER FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.
Contents

Chapter 1
Introduction
LabVIEW Embedded Development Module Example Targets.................................1-1
Understanding the Target Directory Hierarchy and Naming Conventions .............1-2
  Target Directory Hierarchy .............................................................................1-3
  Target Naming Conventions ..........................................................................1-3
  Embedded Project Manager Plug-In VIs .............................................................1-4
  Elemental I/O Implementations .......................................................................1-4
  Example ...........................................................................................................1-4

Chapter 2
Adding a New Target to the Execution Target List
  How It Works .....................................................................................................2-1
  Tips for Creating New Targets .........................................................................2-2

Chapter 3
Implementing the Plug-In VIs for the Embedded Project Manager
  Defining the Target-Specific Menus ..................................................................3-1
  Implementing the Target-Specific Menu Functionality ......................................3-2
  Implementing Toolbar Buttons ..........................................................................3-2
  Enumerating the Targets ....................................................................................3-3
  How It Works .....................................................................................................3-4

Chapter 4
Defining the Code Generation Options
  Code Generation Attributes ...............................................................................4-2
  Signal Naming Convention ...............................................................................4-7

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

Chapter 7
Memory Mapping

How Users Use Memory Mapping ................................................................. 7-1
Implementing Memory Mapping ................................................................. 7-1

Chapter 8
Porting the LabVIEW Embedded Run-Time and Analysis Libraries

Source Code
Portable Pieces of the Run-Time Library ..................................................... 8-2
LVEncodedMain.c .......................................................................................... 8-3
Include Files ................................................................................................. 8-3
Feature Flags ................................................................................................. 8-4
OS-Specific Components ............................................................................. 8-5
Critical Sections ......................................................................................... 8-5
Events ......................................................................................................... 8-5
Threads ....................................................................................................... 8-6
Non-Blocking Operations .......................................................................... 8-7
LabVIEW G-Based Interrupt Service Routines ........................................... 8-8
Printf ......................................................................................................... 8-8
Time .......................................................................................................... 8-8
Serial ......................................................................................................... 8-9
CAN .......................................................................................................... 8-10
TCP/UDP .................................................................................................. 8-11
Static Memory Allocation Support ............................................................ 8-12
Analysis Library ......................................................................................... 8-12

Chapter 9
Implementing Interrupt Service Routine (ISR) Support

Tips for Supporting ISRs on a New Platform ............................................. 9-2

Chapter 10
Embedded Automated Tests

Configuration Files ...................................................................................... 10-1
Test Sequences ............................................................................................ 10-1
Example ..................................................................................................... 10-2
Chapter 11
Elemental I/O

Overview....................................................................................................................... 11-1
I/O Devices.................................................................................................................. 11-1
Elemental I/O Classes............................................................................................... 11-2
Pins ............................................................................................................................ 11-2
Resources.................................................................................................................. 11-2
Plug-Ins.................................................................................................................... 11-3

How Users Use Elemental I/O Nodes............................................................................ 11-3
The System Description Utility................................................................................... 11-4

Implementing Elemental I/O Devices and Classes..................................................... 11-4
Adding a New Elemental I/O Device......................................................................... 11-5
Adding an Additional Elemental I/O Class............................................................... 11-7
Adding a New Resource............................................................................................. 11-7
Adding a New Pin ....................................................................................................... 11-8
Assigning a Pin to a Resource .................................................................................. 11-8
Assigning an I/O Device to a Target......................................................................... 11-8
Exporting a Device Configuration............................................................................. 11-9
Importing a Device Configuration............................................................................. 11-9
Editing a Pin ............................................................................................................... 11-10
Editing a Resource.................................................................................................... 11-10
Editing the Elemental I/O Plug-in VIs ................................................................. 11-10
Determining Which Targets Use an Elemental I/O Device .................................... 11-11
Determining Which Resources Use a Pin ............................................................... 11-11

Implementing the Elemental I/O Plug-In VIs......................................................... 11-11
GetMethod Plug-In VI.............................................................................................. 11-11
System Description Utility Plug-In VI................................................................. 11-13
GCode Plug-In VI...................................................................................................... 11-13
Configuration Pane Plug-In VI................................................................................. 11-14

Showing the Implementation.................................................................................... 11-15

Chapter 12
Static Memory Allocation

Limitations of the Static Memory Model ................................................................. 12-2
Limitations for End-Users......................................................................................... 12-2
Limitations for OEMs............................................................................................... 12-3
Chapter 13
Debugging

Implementing Instrumented Debugging ............................................................... 13-1

Implementing the Instrumented Debugging Communication Layer
for LabVIEW ......................................................................................................... 13-1
  LEP_x_Debug.vi ............................................................................................... 13-2
  Modifying the LEPPlugins.ini File................................................................. 13-2
  Implementing the Host Plug-In VIs................................................................. 13-2
  Open VI ............................................................................................................. 13-3
  Close VI ........................................................................................................... 13-3
  Read VI ............................................................................................................. 13-4
  Write VI ......................................................................................................... 13-4

Implementing the Instrumented Debugging Communication Layer
on the Embedded Target .................................................................................... 13-5
  Connect ........................................................................................................... 13-5
  Disconnect ..................................................................................................... 13-6
  Write ............................................................................................................... 13-6
  Read ............................................................................................................... 13-6
  Bytes Available ............................................................................................ 13-6

Using the Instrumented Debugging Method ...................................................... 13-6
  Assigning to Function Pointers ..................................................................... 13-6

Troubleshooting ................................................................................................. 13-7

Implementing On Chip Debugging .................................................................... 13-7
  .lvm File ...................................................................................................... 13-9
  .c Parser ...................................................................................................... 13-10
  C Compiler .................................................................................................... 13-10
  .lst Parser .................................................................................................... 13-11
  Linker ............................................................................................................ 13-13
  .map Parser .................................................................................................. 13-13
  Debug Database ........................................................................................... 13-14
  Debug Daemon ............................................................................................. 13-14
  Emulator Plug-in .......................................................................................... 13-15
  Adding Support for an Existing Emulator to a New Target ......................... 13-15
  Adding Support for a New Emulator Interface .............................................. 13-16
  Adding the List Parser for Non-GCC Compiler ........................................... 13-18
    EMB_Debug_Map_x_UpdateDD.vi ............................................................... 13-18
    EMB_Debug_Map_Gcc_ResolveSymbols.vi ................................................. 13-18

Examples .......................................................................................................... 13-18
Chapter 14
Customizing the Editing Environment

Editing the Palette Menus ........................................................................................................ 14-1
Configuring the Target Syntax for a New Target ................................................................. 14-1
Modifying the Target Syntax for a Target ........................................................................... 14-4

Chapter 15
Tutorials

Creating a New Target........................................................................................................... 15-1
  Part 1—Using the Target Creation Utility ......................................................................... 15-1
  Part 2—Adding a New CGen Step .................................................................................... 15-2
  Part 3—Modifying the Build Settings Dialog Box ......................................................... 15-4
  Part 4—Modifying the Compile Step .............................................................................. 15-6
Implementing Elemental I/O .............................................................................................. 15-7
  Part 1—Creating a New Elemental I/O Device ......................................................... 15-7
  Part 2—Implementing I/O for MyDevice ........................................................................ 15-14

Appendix A
Technical Support and Professional Services
Introduction

LabVIEW Embedded Development Module

Example Targets

The LabVIEW Embedded Development Module includes several example targets. Use the example targets as a starting point when you create new embedded targets. The target directory hierarchy is designed to allow for code reuse. Example targets are located in the following directory:

```
labview embedded\resource\LabVIEW Targets\Embedded
```

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

The Embedded Development Module includes a blank template target. This target is not intended to be an implementation example, but serves 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.

The different example targets have different implementations. When you are implementing a feature for a new embedded target, look for an existing implementation in an existing target.

The following table lists some of the implementation features for the example targets. Use this table to find an example of a feature you are implementing for a new target.
### Understanding the Target Directory Hierarchy and Naming Conventions

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

<table>
<thead>
<tr>
<th>Target Name</th>
<th>Instrumental Debugging</th>
<th>On Chip Debugging</th>
<th>Pre-Built Runtime Library</th>
<th>Static Memory Model</th>
<th>Memory Mapping</th>
<th>Elemental I/O</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>
</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>Yes</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>Yes</td>
</tr>
<tr>
<td>Unix Console</td>
<td>TCP</td>
<td>No</td>
<td>Yes</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>Yes</td>
</tr>
<tr>
<td>Axiom CMD565, VxWorks ROM Image</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Axiom CMD565, VxWorks Module</td>
<td>Serial</td>
<td>WindRiver WTX</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Intel IXDP425, VxWorks RAM Image</td>
<td>No</td>
<td>WindRiver VisionICE II</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Intel IXDP425, VxWorks ROM Image</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Intel IXDP425, VxWorks Module</td>
<td>TCP</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>VxWorks Simulation</td>
<td>TCP</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Windows Console Application</td>
<td>TCP</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Target Directory Hierarchy

The target directory organizes targets into a hierarchy, shown in Figure 1-1, to provide greater reuse of target implementation code. This hierarchy organizes targets by operating systems. The LabVIEW Embedded Development Module implementations, which contain the plug-in VIs, libraries, helper scripts, and other files required to implement a target are located in the following directory:

```
labview embedded\resource\LabVIEW Targets\Embedded
```

LabVIEW does not recognize targets outside of this directory.

The target directory hierarchy is OS-centric and supports sub-targeting. The top-level targets are self-contained and do not rely on any sub-target implementations below the sub-target. In contrast, the sub-targets rely on the top-level target implementation.

![Diagram of Target Directory Hierarchy]

**Figure 1-1. Target Directory Hierarchy**

Target Naming Conventions

Although you can name new embedded targets in any manner you want, National Instruments recommends you follow the same naming convention as the LabVIEW Embedded Development Module example targets. The example targets have the following naming convention:

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

For example, Axiom CMD565, eCos ROM Image. Do not use underscores in the target name.
Embedded Project Manager Plug-In VIs

The plug-in VIs for the Embedded Project Manager are located in the following directory:

labview embedded\resource\LabVIEW Targets\Embedded\OS\OS_LEP_TargetPlugin\n
You also can implement Embedded Project Manager plug-in VIs for individual targets, which are located in the following directory:

labview embedded\resource\LabVIEW Targets\Embedded\OS\x\os_x_LEP_TargetPlugin

Elemental I/O Implementations

The implementations to support Elemental I/O are located in the following directory:

labview embedded\resource\LabVIEW Targets\Embedded\eio

Example

labview embedded\resource\LabVIEW Targets\Embedded vxworks\cmd565\ram

The Axiom CMD565, VxWorks RAM Image target reuses the implementation of the following targets:

- Axiom CMD565, VxWorks Module
- VxWorks Simulation

The VxWorks Simulation target provides the build framework to compile generated, runtime, external source files, and link intermediate object files with OS and external libraries to build the final embedded application image. The VxWorks Simulation target is a target that can build a downloadable module for the VxWorks simulator. The VxWorks Simulation target also is the base target for other VxWorks targets. You can override the sub-target specific components (subVIs).

LEP_vxworks_cmd565_Build.vi builds a downloadable module for the Axiom CMD565 running the VxWorks operating system.

LEP_vxworks_cmd565_Build.vi overrides the implementation of the Linker Script, Compiler Script, and Default Config methods and calls back the base build VI of the base target, which is VxWorks Simulator, instead of implementing a complete build framework from scratch.
Adding a New Target to the Execution Target List

Complete the following steps to add a new target to the Execution Target list.

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

2. Click the Browse button next to the Template target path control, navigate to the target directory you are using as a template, and click the Select Cur Dir button.

Note You must have read/write access to the target you are using as a template.

3. Enter the name of the target as you want it to appear in LabVIEW in the Target name text box.

4. Click the Browse button next to the Target folder path path control, navigate to the directory you want to create the new target, and click the Select Cur Dir button.

5. Click the Create Target button. The LabVIEW Embedded Development Module copies the template target to the location you specify and relinks the VIs. Click the Continue button if any warnings appear.

6. Click the Close button.

The new target is located in the following directory:

labview embedded\resource\LabVIEW Targets\embedded\new target folder name

How It Works

The Embedded Development Module performs the following actions when you click the Create Target button:

1. Copies the target folder you are using as a template to a new folder.
2. Edits TgtSupp.ini for the new target name.
3. Edits LEPPlugins.ini to use the new paths and names to the Embedded Project Manager plug-in VIs.
4. Copies and renames all VIs in the plug-ins folder to the new name.
5. Relinks the new VIs.
6. Deletes the old VIs.

Note The path to the new target must be on the same folder directory level as the template target.

Tips for Creating New Targets

- Use the blank target if you are porting LabVIEW to a new target.
- This utility can only copy and relink VIs from one target folder. It copies subfolders, but does not work recursively to relink VIs in subtargets. Therefore, you might have unusable subtargets underneath the leaf target. You can delete these unusable targets. In the Creating a New Target tutorial, the rom and ram subfolders contain subtargets that you can delete.
- If the target is a subtarget of other targets and some of its VIs link to VIs in parent targets, copy it to the same level in the target folder hierarchy. Otherwise, sub to parent target links break.
- Do not use underscores in the new target folder name. All the plug-in VIs for all targets must be unique. Otherwise, the LabVIEW linker might pick up the wrong VI, which causes the target to work incorrectly. By convention, underscores are used to separate levels in a target hierarchy in the names of plug-in VIs to make them globally unique. An underscore in a target name can confuse this utility and make the renaming and relinking of target VIs fail.
- The target creation utility cannot relink typedefs so the names of the typedefs in the new target are the same as the original ones from which the target was created.
Implementing the Plug-In VIs for the Embedded Project Manager

Users use the Embedded Project Manager to manage groups of VIs, select build options, build embedded VIs into embedded applications, and run LabVIEW embedded applications on an embedded target. The functionality of the Embedded Project Manager window varies by target. Embedded Project files have a .lep file extension. LEP files contain target-specific build options and other information necessary to generate C code from VIs.

Implementing the Embedded Project Manager functionality is required when porting LabVIEW to a new target.

Defining the Target-Specific Menus

Implement LEP_x_Target_Menu.vi to define the target-specific Target menu items of the Embedded Project Manager window.

Tip Use LEP_x_Target_OnSelect.vi to implement the functionality of the menu items you define in LEP_x_Target_Menu.vi.

Each menu item is defined as one element of the Menu array. Menu items can perform an action, such as Run or Debug, or be a standard menu separator.

Note The Embedded Project Manager infrastructure automatically adds the Switch Execution Target menu item to the Target menu, which users use to switch execution targets.
The Menu array includes the following elements:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu Hierarchy</td>
<td>Specifies the name of the menu to which the item belongs.</td>
</tr>
<tr>
<td>Item Name</td>
<td>Specifies the name of the item in the menu. Leave blank to add a menu separator.</td>
</tr>
<tr>
<td>Menu Tag</td>
<td>A unique string that is passed to LEP_x_Target_OnSelect.vi when a user selects a menu item or clicks a button on the toolbar. Use only alphanumeric characters and underscores. Leave blank to add a menu separator.</td>
</tr>
<tr>
<td>short cut</td>
<td>Defines the keyboard shortcut for the menu item.</td>
</tr>
<tr>
<td>enabled</td>
<td>Defines if the menu item is enabled or disabled.</td>
</tr>
<tr>
<td>checked</td>
<td>Defines if the menu item is selected by default</td>
</tr>
<tr>
<td>separator</td>
<td>Defines if the menu item is a menu separator</td>
</tr>
</tbody>
</table>

**Implementing the Target-Specific Menu Functionality**

Implement LEP_x_Target_OnSelect.vi to define what happens when a user selects a menu item in the **Embedded Project Manager** window.

**Tip** You define the menu items in LEP_x_Target_Menu.vi.

LEP_x_Target_OnSelect.vi uses the tag string as an input parameter and dispatches it to the menu item action. This VI usually contains a case structure that invokes the target-specific action based on the tag string that was passed. The **Embedded Project Manager** window calls LEP_x_Target_OnSelect.vi every time a user selects a menu item.

**Implementing Toolbar Buttons**

Implement LEP_x_BindShortcuts.vi to define which toolbar buttons are available in the **Embedded Project Manager** window. The available toolbar buttons are **Build**, **Download**, **Run**, and **Debug**.

LEP_x_BindShortcuts.vi maps the toolbar buttons into the tag strings. If a button does not have an associated string, the button is not visible in the **Embedded Project Manager** window. The menu tag must match the menu tag in LEP_x_Target_Menu.vi.
Enumerating the Targets

TgtSupp.dll enumerates all targets to build the Execution Target list. Copy TgtSupp.dll from an example target to the new target directory.

You can modify the DLL if you need to implement a more complex enumeration, for example, to enumerate actual devices that are connected to the same bus. The source code for TgtSupp.dll is located in the following directory:

labview embedded\examples\TgtSupp\n
Note Microsoft Visual C++ is required to build this DLL.

TgtSupp.dll exports the following C functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned long CountDevices(void)</td>
<td>Returns the number of discovered devices or subtargets.</td>
</tr>
<tr>
<td>unsigned long DescriptorSize(void)</td>
<td>Returns the length of the longest target descriptor string.</td>
</tr>
<tr>
<td>unsigned long IdSize(void)</td>
<td>Returns the length of the longest ID string.</td>
</tr>
</tbody>
</table>
| unsigned long EnumerateDevices(char* idBuf, char* descrBuf, unsigned long* kindList, unsigned long nDevices, int* ContainsSearchStrings) | Enumerates all subtargets and returns a list of all targets.  
  - idBuf returns a list of ID strings.  
  - descrBuf returns a list of description strings.  
  - kindDevices returns a list of the kinds of devices. Returns 10 for all embedded targets.  
  - ContainsSearchStrings is reserved for future use. Set to FALSE. |
How It Works

The code that enumerates targets executes during the LabVIEW startup. It recursively searches the \labview_embedded\resource\LabVIEW Targets\Embedded directory for TgtSupp.dll, and calls the enumeration method on all of the libraries found. The target names are used to build the Execution Target list. The default implementation of the libraries reads the target name from TgtSupp.ini, which is located in the same directory as the enumeration library.

LEPPlugins.ini, which is modified through the Create Embedded Target dialog box, lists the paths to the target-specific configuration and action VIs. The Embedded Project Manager reads this text file every time a user switches the execution target to update the target-specific menu items and actions for the destination target. LEPPlugins.ini defines the following configuration tokens:

<table>
<thead>
<tr>
<th>Token</th>
<th>Description</th>
<th>Required/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu</td>
<td>Relative path to LEP_x_Target_Menu.vi, which defines the target-specific menu items and associated tag strings.</td>
<td>Required</td>
</tr>
<tr>
<td>OnSelect</td>
<td>Relative path to LEP_x_Target_OnSelect.vi, which executes the desired action based on the passed tag string (maps tag strings into actions).</td>
<td>Required</td>
</tr>
<tr>
<td>BindShortcuts</td>
<td>Relative path to LEP_x_BindShortcuts.vi, which maps the toolbar buttons into the tag strings.</td>
<td>Required</td>
</tr>
<tr>
<td>OnOpen</td>
<td>Relative path to the VI that executes when a user opens an existing embedded project.</td>
<td>Optional</td>
</tr>
<tr>
<td>OnClose</td>
<td>Relative path to the VI that executes when a user closes an existing embedded project.</td>
<td>Optional</td>
</tr>
<tr>
<td>OnSwitchTargetTo</td>
<td>Relative path to the VI that executes when a user changes to a different execution target.</td>
<td>Optional</td>
</tr>
<tr>
<td>OnSwitchTargetFrom</td>
<td>Relative path to the VI that executes when a user changes from an execution target. The VI takes a path input that indicates the directory path of the target.</td>
<td>Optional</td>
</tr>
<tr>
<td>OnStop</td>
<td>Relative path to the VI that executes when a user clicks the Stop button in an embedded VI.</td>
<td>Optional</td>
</tr>
</tbody>
</table>
Defining the Code Generation Options

A VI Server call translates the VI hierarchy into C code. Each VI in the VI hierarchy is translated into separate C files. The C files are the same name as the VI, with a .c file extension instead of a .vi file extension.

This VI Server call also generates the following header files.

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

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

There are many attributes that determine how to generate the C code. Most of the attributes are optimizations that make the generated C code smaller and run faster. Refer to the Code Generation Attributes section for information about when you can use the different attributes. The generated C code might differ in behavior from LabVIEW desktop VIs.

**Note**  This VI Server call is licensed. The user must have a valid, activated license to generate C code for the target you are creating.
**Code Generation Attributes**

The following attributes determine how the LabVIEW C Code Generator generates the C code.

**GenerateDebugInfo**

*Type:* Boolean  
*Default:* FALSE

Set to TRUE to enable instrumented debugging and add extra code needed to debug over serial, Ethernet, or CAN. The extra code takes the form of function calls that update the program state and communicate the state to the host machine to 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

Guard code prevents a user from making simple 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 does make an application slightly larger and slower so it’s a trade-off between performance and reliability. Set 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 mistakes.

The following LabVIEW functions use guard code:
- Array index
- Array replace
- Compound arithmetic
- All floating-point arithmetic operations
- Quotient and remainder

**GenerateSerialOnly**

*Type:* Boolean  
*Default:* FALSE
The default generated C code has the same execution behavior as LabVIEW on the desktop. 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 code that a human would write.

Set 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 for Windows. If a user sets the priority of an individual VI to subroutine on the Execution page on the VI Properties dialog box, the C code is generated for serial only execution regardless of the value of GenerateSerialOnly. Setting GenerateSerialOnly to TRUE causes the C code to be generated for all VIs in the VI hierarchy as if all VIs are subroutine priority VIs. The main difference between the VI Properties subroutine priority VIs and the GenerateSerialOnly C code generation attribute is that subroutine Windows desktop 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. An example of an asynchronous function is TCP Read.

**UseStackVariables**

*Type:* Boolean  
*Default:* FALSE

You can use stack variables in the generated C code to represent wires only if you set GenerateSerialOnly to TRUE. Set UseStackVariables to TRUE to use stack variables.

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

Type: Enumerated type (static, small, large)

Default: small

The MemoryModel attribute defines how to manage memory in an embedded application.

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

**DestinationFolder**

Type: Path

Default: <empty path>

DestinationFolder indicates where you want the LabVIEW C Code Generator to save the generated C files. Unless the IncrementalBuild attribute is TRUE, any C files in the destination folder that have the same name as VIs in the build are overwritten.

**GenerateIntegerOnly**

Type: Boolean

Default: FALSE

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

interruptServiceRoutineVIs

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

This attribute is the list of VIs that are intended to be interrupt service routine (ISR) VIs, which are very specialized and restrictive VIs for use as interrupts. The Embedded Development Module does not support the ability to set a flag in a VI indicating that it is an ISR VI.

GenerateInlinePrintf

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

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

BigEndian

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

The LabVIEW C Code Generator must know if a platform is big endian or little endian in some cases when compound data is generated in byte form. This attribute specifies if a platform is big endian or little endian.
**GenerateCFunctionCalls**

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

Determines the calling interface to subVIs only if you set GenerateSerialOnly to TRUE. The calling interface to a subVI usually is generated in such a way that if an input to or output from the subVI is unwired, the default data is used. This increases the overall amount of code and generated data for a VI relative to what you might use in a normal C program. If all the inputs and outputs are wired, default data is not needed and a more efficient interface could be generated.

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

**GenerateLibraryInterface**

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

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

The C Generator generates a C style function interface for each VI, which sets up the inputs and calls the normal LabVIEW VI interface. The C function name is the same as the VI unless the user uses disallowed characters, such as spaces, in the filename. Underscores replace disallowed characters. The VI behaves the same as a VI running on a Windows desktop by using default data if any input or output is null. Finally, a header file is created that contains all 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 must be 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 a multi-dimensional array in, it is copied into an internal array with the LabVIEW data type determining the number of dimensions. All 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 size limit supplied. The easiest way to pass arrays is as 1D arrays that get reshaped as necessary on the block diagram.

**IncrementalBuild**

*Type:* Boolean  
*Default:* FALSE

Set to TRUE to only overwrite C files that are older than the VIIs from which they are generated during code generation. For example, if `MyProg.vi` has a modification date that is more recent than `MyProg.c`, the C file is overwitten.

Incremental builds allows users to write and combine large applications with a makefile build process that does not rebuild everything every time a change is made.

**OCDIComments**

*Type:* Boolean  
*Default:* FALSE

Extra comments are placed in the generated C code to help figure out where certain items, such as wires and nodes, are in the generated code during non-intrusive debugging. These comments make the generated code harder to read. Use this option to turn off these comments when they are not needed.

**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.

Use the VarNames VI, located in `labview embedded\CCodeGen\` 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.
## Chapter 4: Defining the Code Generation Options

### LabVIEW Signal Source

<table>
<thead>
<tr>
<th>Attribute Used</th>
<th>Block Diagram Examples</th>
<th>Generated C Variable Name</th>
<th>Generated C Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front panel control</td>
<td><img src="image" alt="Motor_On" /></td>
<td>Motor_On</td>
<td>/* Motor On */</td>
</tr>
<tr>
<td>Constant</td>
<td><img src="image" alt="Trigger_Level" /></td>
<td>Trigger_Level</td>
<td>/* Trigger Level */</td>
</tr>
<tr>
<td>Function</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><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><img src="image" alt="Relay_CS" /></td>
<td>Relay_CS</td>
<td>/* Relay: Case Selector */</td>
</tr>
<tr>
<td>Case tunnel</td>
<td><img src="image" alt="My_Structure_CT" /></td>
<td>My_Structure_CT</td>
<td>/* My Structure: CT */</td>
</tr>
<tr>
<td>Shift register</td>
<td><img src="image" alt="Init_SR, Init_SR_1" /></td>
<td>Init_SR, Init_SR_1</td>
<td>/* Init: Shift Register, /* Init: Shift Register 1 */</td>
</tr>
<tr>
<td>LabVIEW Signal Source</td>
<td>Attribute Used</td>
<td>Block Diagram Examples</td>
<td>Generated C Variable Name</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------</td>
<td>------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>Shift register (uninitialized)</td>
<td>Source name, “SR”</td>
<td>Foo_Loop_SR, Foo_Loop_SR_1, Foo_Loop_SR_2</td>
<td>/* Foo Loop: SR */</td>
</tr>
<tr>
<td>Sequence Local</td>
<td>Source name, “SL”</td>
<td>Init_SL</td>
<td></td>
</tr>
<tr>
<td>Sequence tunnel</td>
<td>Source name, “ST”</td>
<td>Level_Y_ST</td>
<td></td>
</tr>
<tr>
<td>Structure terminal</td>
<td>Label, Terminal name</td>
<td>Wave_Loop_i</td>
<td></td>
</tr>
<tr>
<td>Local variable</td>
<td>Label, local variable name</td>
<td>Motor_Control_v, Motor_Control</td>
<td></td>
</tr>
</tbody>
</table>
### Note

Generated variables must be unique. If multiple variables have identical names, the LabVIEW C Code Generator appends a sequential number to the end of the variable name to make the name unique.

<table>
<thead>
<tr>
<th>LabVIEW Signal Source</th>
<th>Attribute Used</th>
<th>Block Diagram Examples</th>
<th>Generated C Variable Name</th>
<th>Generated C Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global variable</td>
<td>Label, global variable name</td>
<td><img src="image" alt="Sensor VI" /></td>
<td>sensors.Temp_1</td>
<td>/* sensors: Temp 1 */</td>
</tr>
<tr>
<td>User VI</td>
<td>Label, terminal name</td>
<td><img src="image" alt="Filter VI" /></td>
<td>Filter_Analog</td>
<td>/* Filter: Analog */</td>
</tr>
</tbody>
</table>
Defining Compiler Directives

Compiler directives enable and disable features, and are passed to the compiler from the command line by LEP_x_ScriptCompiler.vi. The Build Settings dialog box exposes the implementation of the compiler directives.

The macros in this chapter define how to compile generated and external C code. The LabVIEW Embedded Development Module example targets compile the individual generated C files by scripting the C compiler from the LEP_x_ScriptCompiler.vi plug-in located in the following directory:

```
labview embedded\resource\LabVIEWTargets\Embedded\os\os_LEP_TargetPlugin
```

where os is the target operating system.

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

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

The VxWorks CMD565 example target uses a subtarget-specific implementation. Navigate to and open the following file:

```
labview embedded\LabVIEW Targets\Embedded\vxworks\cmd565\vxworks_cmd565_LEP_TargetPlugin\LEP_vxworks_cmd565_Build.vi
```

This implementation overrides the default implementation of the Compiler Script, Linker Script, and Default Config VIs. Notice the case structure with target-specific VIs. The VIs in the case structure never execute, but are always loaded into memory. Because the VIs are loaded into memory, the Open VI Reference function can refer to VIs by name instead of absolute path.

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

## Operating System Macros

The following table describes the operating system macros.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win32</td>
<td>Define for all Windows targets.</td>
</tr>
<tr>
<td>_vxworks</td>
<td>Define for all VxWorks targets except VxWorks Simulation targets.</td>
</tr>
<tr>
<td>_vxworksim</td>
<td>Define for all VxWorks Simulation targets.</td>
</tr>
<tr>
<td>_ECOS</td>
<td>Define for all eCos targets.</td>
</tr>
<tr>
<td>linux</td>
<td>Define for all Linux targets.</td>
</tr>
</tbody>
</table>
**VxWorks-Only Macros**

The following table describes the VxWorks-only macros.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Defines the CPU family.</td>
</tr>
<tr>
<td></td>
<td>• SIMNT for VxWorks Simulation running on Windows.</td>
</tr>
<tr>
<td></td>
<td>• PPC5XX for Power PC 5XX class CPU.</td>
</tr>
<tr>
<td></td>
<td>• PPC405 for Power PC 405 CPU.</td>
</tr>
<tr>
<td></td>
<td>• XSCALE for Intel XScale class CPU</td>
</tr>
</tbody>
</table>

**Debugging Macros**

The following table describes the debugging macros.

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

**General Macros**

The following table describes the general macros.

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

The following table describes the feature macros.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANSupport</td>
<td>Set to 1 to enable CAN support. To use Embedded CAN with your target, you must obtain a driver for the board from Wind River and compile it into your VxWorks image.</td>
</tr>
<tr>
<td>SocketSupport</td>
<td>Set to 1 to enable BSD socket support.</td>
</tr>
<tr>
<td>_Win32_Scheduler</td>
<td>Set to 1 to enable multithreading support. This macro is not Windows only.</td>
</tr>
<tr>
<td>HeapListSupport</td>
<td>Set to 1 to enable head support.</td>
</tr>
<tr>
<td>BluetoothSupport</td>
<td>Set to 1 to enable Bluetooth support.</td>
</tr>
<tr>
<td>FileSupport</td>
<td>Set to 1 to enable file I/O support.</td>
</tr>
<tr>
<td>IrDA Support</td>
<td>Set to 1 to enable IrDA support.</td>
</tr>
<tr>
<td>SerialSupport</td>
<td>Set to 1 to enable serial support.</td>
</tr>
<tr>
<td>DatalogSupport</td>
<td>Set to 1 to enable datalogging support.</td>
</tr>
<tr>
<td>TCPUDPSupport</td>
<td>Set to 1 to enable TCP/UDP support.</td>
</tr>
</tbody>
</table>

# Memory Mapping Macros

The following table describes the memory mapping macros.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DATA_Section</td>
<td>Defines the input memory section for array structures.</td>
</tr>
<tr>
<td>TEXT_Section</td>
<td>Defines the input memory section for generated code.</td>
</tr>
</tbody>
</table>
Defining the Linking of Object Files and Libraries

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

```
labview embedded\resource\LabVIEWTargets\Embedded\os\os_LEP_TargetPlugin
```

where `os` is the target operating system.

You must implement the LEP_x_ScriptLinker VI if you require behavior that differs from the default implementation.

The Build VI executes the LEP_x_ScriptLinker VI only once to produce the final application image. The VI must enumerate all intermediate object files, external files, run-time libraries, and operating system libraries.

Some targets use `prjObjs.lst` to get a list of all modules to avoid long command lines. The Build VI generates `prjObjs.lst` and lists all binary modules and libraries.

VxWorks module targets do not link with the operating system libraries because the VxWorks module loader resolves all symbols. The VxWorks and eCos example targets do not have pre-compiled run-time libraries, so the LEP_x_ScriptLinker VI also must link with the run-time library intermediate files. The Build VI executes the default implementation of the LEP_x_ScriptLinker VI unless you override the default implementation and create a custom LEP_x_ScriptLinker VI. You can override the default implementation by wiring a VI reference to the custom LEP_x_ScriptLinker VI to the Linker Script input of the Build VI. The custom implementation of the LEP_x_ScriptLinker VI must have the same connector pane as the default implementation.

The LEP_x_ScriptLinker VI plug-in can directly execute the C linker 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 examples use a batch file because you must set the Tornado system variables before the C compiler executes.

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

The VxWorks CMD565 example target uses a subtarget-specific implementation. Navigate to and open the following file:

![labview embedded\LabVIEW Targets\Embedded\vxworks\cmd565\vxworks_cmd565_LEP_TargetPlugin\LEP_vxworks_cmd565_Build.vi](labview embedded\LabVIEW Targets\Embedded\vxworks\cmd565\vxworks_cmd565_LEP_TargetPlugin\LEP_vxworks_cmd565_Build.vi)

This implementation overrides the default implementation of the Compiler Script, Linker Script, and Default Config VIs. Notice the case structure with target-specific VIs. 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.
Memory Mapping

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

How Users Use Memory Mapping

Users open the Memory Mapper dialog box from the Tools menu in the Embedded Project Manager window to change implicit mapping, if necessary. For example, if the user's application uses a large, read-only global variable that he wants to download to ROM memory, the user can change the memory mapping for the variable.

Implementing Memory Mapping

Implementing memory mapping is optional. The LabVIEW C Code Generator prepends the _DATA_SECTION macro to all array constants. The C compiler puts arrays into the default input section (.data) if you define _DATA_SECTION as empty for example. You can define this macro as section (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 on how to explicitly place variables.

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

The memory mapping configuration persists in the LabVIEW Embedded Project (LEP) file.
To implement memory mapping, you must implement the following VIs:

**LEP_x_SectionNames.vi**
Defines the name of all memory sections users can map, such as RAM, ROM, and flash memory. Use descriptive names that users understand, because users select the memory mapping from the names you implement in **LEP_x_SectionNames.vi**.

**LEP_x_MemoryMap_Default.vi**
Defines the default mapping of VIs to memory sections on the target. The connector pane of this VI must match **EMB_Utility_MemMap_VI_Default.ctl**, which is located in the following directory:

```
\labview embedded\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. The connector pane of this VI must match **EMB_Utility_MemMap_VI_Compare.ctl**, which is located in the following directory:

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

**LEP_x_MemoryMap.vi**
Reads the memory mapping configuration data from the LEP file, initializes the memory mapping framework, and displays the **Memory Mapper** dialog box to the user.

Use **EMB_Utility_MemMap_Init.vi** to initialize the memory mapping framework. **Call EMB_Utility_MemMap.vi** to display the **Memory Mapper** 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 or 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.

Refer to the following directory for an example implementation.

```plaintext
labview embedded\resource\LabVIEW Targets\Embedded\vxworks\cmd565\rom\vxworks_cmd565_rom_LEP_TargetPlugin\LEP_vxworks_cmd565_rom_MemoryMap_Query.vi
```

Refer to the following directory for an example usage.

```plaintext
labview embedded\resource\LabVIEW Targets\Embedded\vxworks\cmd565\rom\vxworks_cmd565_rom_LEP_TargetPlugin\LEP_vxworks_cmd565_rom_ScriptCompiler.vi
```

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

The LabVIEW C Code Generator run-time source code is located in the following directory:

```
labview embedded\CCodeGen\libsrc
```

Figure 8-1 shows the directory structure of the run-time library source code.

```
libsrc
  blockdiagram
  comms
  frontpanel
  lvanalysis
  main
  os
    common
    ecos
    unix
    vxworks
    win
```

**Figure 8-1.** Run-Time Library Source Code Directory Structure

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

```
labview embedded\CCodeGen\include
```

The directory structure of the `include` directory mirrors the directory structure of the `libsrc` directory shown in Figure 8-1.
Portable Pieces of the Run-Time Library

You should not have to modify the code in the following folders in `labview embedded\CCodeGen\libsrc`. The C files in these folders include a header file, which includes an OS-specific header file that contains macros and definitions.

`labview embedded\CCodeGen\libsrc\blockdiagram` contains code for arrays, strings, clusters, integers, floating-point numerics, complex numbers, file support, and number to text.

`labview embedded\CCodeGen\libsrc\comms` contains code for serial and TCP/IP communications.

Note `labview embedded\CCodeGen\libsrc\comms` also contains code for Bluetooth and IrDA, which has only been ported to the Palm OS and Pocket PC platforms.

`labview embedded\CCodeGen\libsrc\frontpanel` contains code for front panel terminals only; there are no user interface components to display the terminals. The Embedded Development Module supports the following controls:

- Array
- Boolean
- Chart
- Cluster
- Enumerated type
- Graph
- Listbox
- Numeric
- Picture
- Ring
- String
- Tab
LVEmbeddedMain.c

LVEmbeddedMain.c contains the main entry point for all applications LabVIEW generates for all platforms. This file contains functions that perform set up and tear down of common pieces, such as occurrences and FIFOs. It 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. After the top-level VI has completed, shutdown functionality is called.

There are two macros in LVEmbeddedMain.c called LV_PLATFORM_INIT and LV_PLATFORM_FINI. If you define LV_PLATFORM_INIT as a function, LV_PLATFORM_INIT is called before any VI is called and before any other setup is performed. The target defines this function and you can implement it in any way that is appropriate for the 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.

labview embedded\CCodeGen\libsrc\os\ folder hierarchy does not define LV_PLATFORM_INIT and LV_PLATFORM_FINI because there is not a one-to-one mapping between operating systems and targets. For example, an eCos target on a Gameboy Advanced device needs a certain LV_PLATFORM_INIT() to set up the hardware, but eCos on the CMD565 does not.

Include Files

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

- LVSysIncludes.h
- LVDefs_plat.h

There is a version of these two include files, located in labview embedded\CCodeGen\include\os\ for each supported OS.

LVSysIncludes.h

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

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

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

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

**Feature Flags**

The following flags in LVDefs_plat.h turn on and off large sets of features that correspond to pieces of hardware that might or might not be present on the target device.

<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 application exits. This 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 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 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 SocketSupport flag.</td>
</tr>
<tr>
<td>FileSupport</td>
<td>Controls file I/O support.</td>
</tr>
</tbody>
</table>
**OS-Specific Components**

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

**Critical Sections**

**LVCritSect.c**

ISRs, Timed Loops, RT-FIFO VIs, and occurrences are dependent on LVCritSect.c, which contains the following functions:

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

- `LVCriticalSection 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 when a section of code is entered that must not be interrupted.

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

**Events**

**LVEvent.c**

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

InitLVEvents(), UninitLVEvents()

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

LVEvent 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 will 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:** Wait on event with timeout. Suspends the executing thread until the event is set.

### Threads

**LVThreads.c**

Timed Loops uses 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 makes a thread active and running. It
might be a no-op if threads are created and activated by 
LVThreadCreate.

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 of the main thread.

LVTHREAD_PRIORITY LVThreadGetMainPriority()

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

void LVThreadSleep_ms(unsigned long ms)

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

---

**Non-Blocking Operations**

**LVNBOps.c**

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

LVNBOpsAtomicCompareExchange(uInt32 
*inspectedLocation, uInt32 oldValue, uInt32 newValue)
LVNBOpsAtomicCompareExchange16(uInt16 *inspectedLocation, uInt16 oldValue, uInt16 newValue)
LVNBOpsAtomicCompareExchange8(uInt8 *inspectedLocation, uInt8 oldValue, uInt8 newValue)

**LabVIEW G-Based Interrupt Service Routines**

**OEM_LVISR.c**

This functionality is used to implement G-based ISRs. It contains the following functions:

OEMISRBoilerPlate(int param)

**Purpose**: Looks up ISR VI in the lookup table indexed by param.

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**: Looks for empty slot in the lookup table, places ISRFunc in it, 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**: Remove ISR from the lookup table.

**Printf**

This functionality is used by the One Button Dialog function. The function is implemented in PDAStrSupport_os.c, where os is the operating system. For example, PDAStrSupport_vxworks.c.

**Time**

**CCGTimeSupport_OS.c**

uInt32 LVGetTicks()

**Purpose**: Returns clock ticks in milliseconds. You must port this function because this function is used throughout the code.
Boolean DtToSecs( VoidPtr vpIn, DataType dt, uInt32 *pSecs )
  
  **Purpose:** Converts a cluster containing date and time to seconds. It is called directly from generated 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. It is called directly from generated 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 generated code to implement the Get Date Time function.

---

**Serial**

**PlatformSerial_OS.c**

This functionality is used for serial functions and for debugging over a serial connection. It contains the following functions:

SerialInit(SerialDevice ser, SerialConfig *cfg)
  
  **Purpose:** Initializes the serial driver.

SerialOpen(int portNumber, SerialDevice *serptr)
  
  **Purpose:** Opens a serial port.

SerialClose(SerialDevice ser)
  
  **Purpose:** Closes a serial port.

SerialRead(SerialDevice ser, char *buffer, int *length)
  
  **Purpose:** Reads bytes from a serial port.

SerialWrite(SerialDevice ser, const char *buffer, int *length)
  
  **Purpose:** Writes bytes to a serial port.

SerialBytesAvail(SerialDevice ser, int *bytes)
  
  **Purpose:** Returns the number of bytes available from a serial port.
SerialBreak(SerialDevice ser)

Purpose: Sends a break on a serial line.

CAN

PlatformCAN_OS.c

This functionality is used for embedded CAN VIs and for debugging over a controller area network (CAN) connection. It contains the following functions:

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

Purpose: Opens a session to a CAN controller.

MgErr LvCanClose(uInt32 reference);

Purpose: Closes an open session to a CAN controller.

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

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

MgErr LvCanFreeChannel(uInt32 reference, uInt8 channelNum);

Purpose: Frees a channel you open with LvCanGetChannel.

MgErr LvCanStart(uInt32 reference);

Purpose: Starts communicating with the CAN controller.

MgErr LvCanStop(uInt32 reference);

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

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

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

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

Purpose: Writes a CAN message frame to the specified channel.
MgErr LvCanGetGlobalFilter(uInt32 reference, int32 *filter);

**Purpose:** Retrieves the filter you set with LvCanSetGlobalFilter.

MgErr LvCanSetGlobalFilter(uInt32 reference, int32 filter);

**Purpose:** Sets the global filter for CAN messages.

MgErr LvCanGetChannelFilter(uInt32 reference, uInt8 channel, int32 *filter);

**Purpose:** Returns the channel filter you set with LvCanSetChannelFilter.

MgErr LvCanSetChannelFilter(uInt32 reference, uInt8 channel, int32 filter);

**Purpose:** Sets the CAN message filter for the specified open channel.

MgErr LvCanGetBaudRate(uInt32 reference, uInt32 *baudRate);

**Purpose:** Returns the current baud rate of the CAN controller.

MgErr LvCanSetBaudRate(uInt32 reference, uInt32 baudRate);

**Purpose:** Sets the baud rate of the CAN controller.

MgErr LvCanMessageAvail(uInt32 reference, uInt8 channel, uInt8 *bAvail);

**Purpose:** Polls channels for new CAN messages to read.

**TCP/UDP**

This functionality 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. The Embedded Development Module example targets support Berkeley. The implementation file is PDANetConnSupport.c, which is located in the labview_embedded\CCodeGen\comms directory, because there is no OS-specific piece, but it might be OS-specific.

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

If you want to support applications that allocate no memory, you must define some constants to determine the maximum number of items associated with various features that users can use with the new target. You must statically declare these items using the following constants:

- `GENERATE_OCCURRENCE_MAX`—Defines the maximum number of generate occurrence functions.
- `WAIT_ON_OCCURRENCE_MAX`—Defines the maximum number of wait on occurrence functions.
- `ISR_HANDLE_MAX`—Defines the maximum number of ISR handler VIs.
- `CRITICAL_SECTION_MAX`—Defines the maximum number of critical section handles.
- `EVENT_MAX`—Defines the maximum number of events.
- `NODE_MAX`—Defines the maximum number of external time sources.
- `FIFO_MAX`—Defines the maximum number of RT-FIFOs.
- `FIFO_SIZE_MAX`—Defines the maximum size of an RT-FIFO in elements.

Analysis Library

Note  Targets must support floating-point to support the LabVIEW analysis library.

To port the analysis library, look for an OS similar to the one you are working on 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\embedded\CCodeGen\analysis\development\source\include\
```

You can make the analysis library with or without BLAS/LAPACK, which is a set of highly optimized, open source math functions widely used in industry and academia. LabVIEW Embedded Development Module does not include BLAS/LAPACK because BLAS/LAPACK makes analysis functionality very large. BLAS/LAPACK is more appropriate for desktop PCs where large amounts of memory are available. Therefore, you must define `COMPILE_FOR_SMALL_RT` to remove BLAS/LAPACK support.
Removing BLAS/LAPACK support gives you the same analysis library as LabVIEW 7.0.

The following directories contain the analysis source code:

- `labview embedded\CCodeGen\analysis\LV\source` contains code to convert LabVIEW data types to C data types.
- `labview embedded\CCodeGen\analysis\development\source` contains the actual analysis algorithms.

The C++ code in the 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 desktop LabVIEW. The peak detector and pulse duration functions will crash and do not work on embedded targets.

The LabVIEW source includes two versions of the linear algebra functions.

- BLAS/LAPACK version is located in `CCodeGen\analysis\development\source\linalg`.
- Non-BLAS/LAPACK version is located in `CCodeGen\analysis\development\source\linalgVer1`.

Use the non-BLAS/LAPACK version.

There are other #defines to note:

```
NoFFTtablePersist=1
LVANALYSIS=1
ThreadsUsed=0
```

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

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

```
labview embedded\PDA\libsrc\lvanalysis\arrresize.c
```
Implementing Interrupt Service Routine (ISR) Support

If you want to support ISR functionality for a new target, you must implement a set of functions in OEM_LVISR.c, which is located in the following directory:

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

The function prototypes are defined in LVISR.h, which is located in the following directory:

```
labview embedded\CCodeGen\include\blockdiagram
```

You must implement the following functions:

**Boolean InitOEMISRs();**

This function is only called by InitISRs. It must do any required ISR subsystem initialization.

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

**Boolean UninitOEMISRs();**

This function is only called by UninitISRs. It must do any necessary ISR subsystem cleanup.

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

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

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 to be passed to the
ISRFunc isr_runFunc, register_param is an output parameter that is
unique to this ISR registration instance. The register_param value will be
passed to the OEMISRUnregisterHandler function when the ISR is being
unregistered.

Returns TRUE on success.
Returns FALSE on failure.

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

This function is only called by ISRUnregisterHandler. This function
unregisters an ISR with the OS or with the hardware. The register_param
passed in is the register parameter that was returned by the
OEMISRRegisterHandler routine.

Returns TRUE on success.
Returns FALSE on success.

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

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

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

LabVIEW 7.1 Embedded\CCodeGen\libsrc\os\vxworks\OEM_LVISR.c
Embedded Automated Tests

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

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

Each test contains separate configuration files for each target. The automated test framework automatically switches targets. Add configuration files for all applicable tests when you add a new target.

Configuration Files

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

labview embedded\autotest

Test Sequences

LEP_AutoTest_TestSequence_1.vi

This is the default test sequence. It expects test results in the text file. This sequence is for all targets that can redirect standard output to file. Refer to the LEP_win32con_Run VI for more details how to redirect standard output to a test text file. The Windows Console Application and VxWorks Simulation targets use this sequence.

Example test file: labview embedded\autotest\tests\Large semi-auto tests\arr-dr\win32con_Target.ini
LEP_AutoTest_TestSequence_2.vi

Supports target hardware reset and serial port capture capability. This sequence expects that the Run button can download and execute an embedded application, and that standard output is redirected on a serial port.

Example test file: labview embedded\autotest\tests\Large semi-auto tests\arr-dr\vxworks_cmd565_Target.ini

LEP_AutoTest_TestSequence_3.vi

Similar to LEP_AutoTest_TestSequence_2.vi. The only difference is that it terminates the Target Server process prior to executing an embedded application. This sequence works exclusively with WindRiver Tornado.

Example test file: labview embedded\autotest\tests\Large semi-auto tests\arr-dr\vxworks_cmd565_Target.ini

LEP_AutoTest_TestSequence_4.vi

Similar to LEP_AutoTest_TestSequence_2.vi. The only difference is that this sequence doesn't reset the target and explicitly downloads the embedded application.

Example test file: labview embedded\autotest\tests\Large semi-auto tests\arr-dr\vxworks_ixdp425_ram_Target.ini

Example

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

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

[arr_dr_Test_Debug]
_Test_TopLevelVI_arr-dr-con.vi
_Test_SequenceVI_LEP_AutoTest_TestSequence_2.vi
_i_Test_BaudRate_=57600
_s_Test_ResourceName_=COM6
_Test_Timeout_=300000
iBuildConfiguration=1
bGenerateDebugInfo=FALSE
iMemoryModel=0
bGenerateSerialOnly=FALSE
bGenerateGuardCode=TRUE  
bUseStackVariables=FALSE

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[arr_drv_Test_Debug]</td>
<td>The name of the test. Test names must be enclosed in brackets and cannot contain spaces.</td>
</tr>
<tr>
<td><em>Test_TopLevelVI</em></td>
<td>The name of the top-level 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>
<tr>
<td>i_Test_BaudRate_</td>
<td>The baud rate of the capturing serial port. This key is used only by test sequences that capture output on the serial port.</td>
</tr>
<tr>
<td>s_Test_ResourceName_</td>
<td>The name of the capturing serial port. This key is used only by test sequences that capture output on the serial port.</td>
</tr>
<tr>
<td><em>Test_Timeout</em></td>
<td>The time interval in milliseconds to get the result before the test fails.</td>
</tr>
<tr>
<td>iBuildConfiguration</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>i = integer value</td>
</tr>
<tr>
<td>iMemoryModel</td>
<td>b = Boolean 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>
Elemental I/O

Overview

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

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

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

I/O Devices

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

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

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

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

Pins

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

Pins usually have a memory mapped register in the hardware's address space for access, or you can access the pins through a device driver.

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. Resources can share pins, but there is a way to enforce mutual exclusion by implementing a plug-in for the Edit System Description button in the Elemental I/O Node configuration dialog box.

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

The Elemental I/O plug-in VIs enforce how users can use pins and resources when configuring and using Elemental I/O Nodes. The Elemental I/O Device Editor creates most of the plug-in VIs for you to modify and implement. The Elemental I/O plug-in VIs are located in the following directory:

```
labview embedded\resource\LabVIEW Targets\embedded\eio\<device name>\Plugins
```

The I/O device name is prepended to the plug-in names to make the plug-ins unique across devices, which avoids linker problems when loaded. You can open the plug-in VIs through the Elemental I/O Device Editor.

How Users Use Elemental I/O Nodes

Users place Elemental I/O Nodes on the block diagram. The Elemental I/O palette contains an Elemental I/O Node for each supported I/O class: analog input, analog output, digital input, digital output, digital bank input, digital bank output, and pulse width modulation output. When you create an Elemental I/O Device, you define which of the I/O classes users can use. For example, an embedded target might only support digital input and digital output. If the target you are creating does not support certain I/O classes, you can remove the Elemental I/O Nodes from the palette.

The Elemental I/O Node does not have any inputs or outputs when a user first places an Elemental I/O Node on the block diagram. Users configure an Elemental I/O Node by double-clicking the node on the block diagram to open a configuration dialog box. Users also can right-click the node and select Configure from the shortcut menu. Users select the Elemental I/O resource to use for the I/O operation along with any other necessary configuration parameters you implement in the ConfigurationPane plug-in VI. You also can implement configuration panes for I/O classes. For example, an analog input implementation might have an option specifying the sample rate of the signal. This option appears on the Configuration tab of the Elemental I/O Node configuration dialog box.

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

Figure 11-1 shows the System Description Utility, which users open from the Elemental I/O Node configuration dialog box, to add resources usable in Elemental I/O Nodes from the pool of all available resources. You use the SDUtilityPlugin VI to implement the functionality behind the System Description Utility. The System Description Utility can prevent users from adding unusable resources due to mutual exclusion of pins and I/O circuitry.

Implementing Elemental I/O Devices and Classes

You implement Elemental I/O by providing a set of configuration files, plug-in VIs, and XML files. Use the Elemental I/O Device Editor to define the I/O classes, pins, and resources for a device. The Elemental I/O Device Editor also creates the necessary files for implementing Elemental I/O in a new folder in the following directory:

```
labview embedded\resources\LabVIEW Targets\eio
```

This directory contains a subfolder for each I/O class.
Select **Tools»Edit Elemental I/O Devices** to open the Elemental I/O Device Editor.

Implementing elemental I/O consists of the following:

1. Adding a new elemental I/O device.
2. Adding resources.
3. Adding pins.
4. Assigning pins to resources.
5. Assigning the I/O device to targets.
6. Implementing the Elemental I/O plug-in VIs.

### Adding a New Elemental I/O Device

Complete the following steps to add a new elemental I/O device.

1. Select **Tools»Edit Elemental I/O Devices** to open the **Edit Elemental I/O Devices** window.
2. Select the **Devices** folder and click the **Add** button to launch the **Add Elemental I/O Device** wizard.
3. On the **Name the device** page of the wizard, enter the name of the new device in the **I/O device name** text box. The name you enter is the folder name for the device information and plug-ins. The I/O device name must be unique and follow standard Windows rules for naming files.
4. Click the **Next** button to open the **Select the Elemental I/O Classes** page.
5. Define the Elemental I/O classes the I/O device supports. This is equivalent to defining what kinds of I/O, for example analog input or analog output, the device supports.
   a. Select the elemental I/O classes the I/O device supports in the **All Elemental I/O classes** list.
   b. Click the **Add** button to add the elemental I/O classes to the **Elemental I/O classes for this device** list. Click the **Remove** button to remove an Elemental I/O class from the list.
6. Click the **Next** button to open the **Add Resources** page.
7. Define the resources the I/O device supports. Each resource has a name and type.
   a. Click the **Add** button to open the **Add Resources** dialog box.
   b. Enter a prefix, `AI` for analog input for example, for the resource in the **Resource prefix** text box.
c. Enter a start index in the **Start index** text box. Each resource contains a sequential number. The start index defines the starting number.

d. Enter the number of resources for the I/O device in the **Number of resources** text box.

e. Select the I/O class from the **I/O class** drop-down list, which contains the I/O classes you select on the **Select the Elemental I/O Classes** page of the wizard.

f. Click the **OK** button to return to the wizard.

8. Click the **Next** button to open the **Add Pins** page.

9. Define the physical pins on the I/O device.
   a. Click the **Add** button to open the **Add Pins** dialog box.
   b. Enter a prefix for the pins in the **Pin prefix** text box.
   c. Enter a start index in the **Start index** box. Each pin contains a sequential number. The start index defines the starting number.
   d. Enter the number of pins for the I/O device in the **Number of pins** text box.
   e. Click the **OK** button to return to the wizard.

10. Click the **Next** button to open the **Assign the Pins to Resources** page.

11. Define the connection between the pins and the resources you created. Typically, a resource can use more than one pin and a pin can be used by more than one resource.
   a. Select the pin(s) you want to assign in the **Pins** list.
   b. Select the resource to which you want to assign the pins from the **Resources** list.
   c. Click the **Add** button to add the pins to the **Pins assigned to resource** list. To remove a pin from a resource, select the resource, select the pin, and click the **Remove** button.

12. Click the **Next** button to open the **Add the Device to Targets** page.

13. Assign the I/O device you are creating to the applicable targets in the **Available targets** list. It is common for several targets that use the same CPU to support the same Elemental I/O device. Any target to which you add the Elemental I/O device uses that Elemental I/O device for the Elemental I/O Nodes. If a target has multiple devices, you can select which device to use in the Elemental I/O Node configuration dialog box.

14. Click the **Finish** button. The **Add Elemental I/O Device** wizard creates the new I/O device in the **Edit Elemental I/O Devices** window.
Adding an Additional Elemental I/O Class

Complete the following steps to add a new elemental I/O class.

2. Under Devices, expand the device to which you want to add the Elemental I/O class.
3. Select the Elemental I/O Classes folder and click the Add button to open the Select Elemental I/O Class dialog box.
4. Select the class you want to add from the Elemental I/O Class drop-down list.
5. Click the OK button to return to the Edit Elemental I/O Devices window. The Elemental I/O Classes folder now contains the new elemental I/O class, the mass-compiled ConfigPane plug-in VI, and the mass-compiled GCode plug-in VI.

Adding a New Resource

Complete the following steps to add a new resource.

2. Under Devices, expand the device to which you want to add the resource.
3. Select the Resources folder and click the Add button to open the Add Resources dialog box.
4. Enter a prefix, AI for analog input for example, for the resource in the Resource prefix text box.
5. Enter a start index in the Start index box. Each resource contains a sequential number. The start index defines the starting number.
6. Enter the number of resources for the I/O device in the Number of resources text box.
7. Select the I/O class from the I/O class drop-down list, which contains the I/O classes you created for the device.
8. Click the OK button to return to the Elemental I/O Devices window. The Resources folder now contains the new resource.
Adding a New Pin

Complete the following steps to add a new pin.
1. Select **Tools»Edit Elemental I/O Devices** to open the **Edit Elemental I/O Devices** window.
2. Under **Devices**, expand the device to which you want to add the resource.
3. Select the **Pins** folder and click the **Add** button to open the **Add Pins** dialog box.
4. Enter a prefix for the pins in the **Pin prefix** text box.
5. Enter a start index in the **Start index** box. Each pin contains a sequential number. The start index defines the starting number.
6. Enter the number of pins for the I/O device in the **Number of pins** text box.
7. Click the **OK** button to return to the **Elemental I/O Devices** window. The **Pins** folder now contains the new pins.

Assigning a Pin to a Resource

Complete the following steps to assign a pin to a resource.
1. Select **Tools»Edit Elemental I/O Devices** to open the **Edit Elemental I/O Devices** window.
2. Under **Devices**, expand the device that contains the resource to which you want to assign a pin.
3. Expand the **Resources** folder and select the resource to which you want to assign a pin.
4. Click the **Add** button to open the **Select Pin** dialog box.
5. Select the pin you want to assign.
6. Click the **OK** button to return to the **Elemental I/O Devices** window.

Assigning an I/O Device to a Target

Complete the following steps to assign a device to a target.
1. Select **Tools»Edit Elemental I/O Devices** to open the **Edit Elemental I/O Devices** window.
2. Expand **Targets** and select the target to which you want to assign a device.
3. Click the **Add** button to open the **Select Device** dialog box.
4. Select the device you want to assign from the Devices list.
5. Click the OK button to return to the Elemental I/O Devices window.

Tip You also can drag a device to a target.

Exporting a Device Configuration

You can export an elemental I/O device for use on another PC through an XML file.

Complete the following steps to export an Elemental I/O device configuration.

2. Expand Devices and select the Elemental I/O device you want to export.
3. Click the Export button.
4. Navigate to the location where you want to save the XML file.
5. Enter a name for the XML file and click the OK button.

Importing a Device Configuration

You can import Elemental I/O devices that you or someone else creates through an XML file.

Complete the following steps to import an Elemental I/O device configuration.

2. Select the Devices folder.
3. Click the Import folder.
4. Navigate to and select the XML that contains the elemental I/O device configuration.
5. Click the OK button. If the device does not already exist in the Edit Elemental I/O Devices window, the Elemental I/O Device Editor imports the new device and creates the new plug-in VIs.

Note You must separately migrate any plug-in VIs that exist when you import a device from another system.
Editing a Pin

Complete the following steps to edit a pin.

1. Select ToolsÆEdit Elemental I/O Devices to open the Edit Elemental I/O Devices window.
2. Under Devices, expand the device that contains the pin you want to edit.
3. Expand the Pins folder and select the pin you want to edit.
4. Click the Edit button to open the Edit Pin Name dialog box.
5. Enter the new name for the pin.
6. Click the OK button to return to the Elemental I/O Devices window.

Note If the pin is assigned to any other resource, the pin name changes in the other resources too.

Editing a Resource

Complete the following steps to edit a resource.

1. Select ToolsÆEdit Elemental I/O Devices to open the Edit Elemental I/O Devices window.
2. Under Devices, expand the device that contains the resource you want to edit.
3. Expand the Resources folder and select the resource you want to edit.
4. Click the Edit button to open the Edit Resources dialog box.
5. Enter the new name for the resource in the Resource name box.
6. Select the I/O class to which the resource belongs from the I/O Class drop-down list. If an I/O class has been removed, you see <1> in the I/O class. Select another I/O class.
7. Click the OK button to return to the Elemental I/O Devices window.

Editing the Elemental I/O Plug-in VIs

Complete the following steps to edit the ConfigPane, GCode, or SDUtilityPlugin VIs.
1. Select ToolsÆEdit Elemental I/O Devices to open the Edit Elemental I/O Devices window.
2. Under Devices, expand the device that contains the plug-in VIs you want to edit.
3. Expand the Plug-Ins folder and select the VI you want to edit.
4. Click the **Edit** button. You can now edit and save the plug-in VI the same way you edit and save any other LabVIEW VI.

**Determining Which Targets Use an Elemental I/O Device**

Complete the following steps to determine which targets use a particular Elemental I/O device.

1. Select **Tools»Edit Elemental I/O Devices** to open the **Edit Elemental I/O Devices** window.
2. Expand the **Devices** folder and select the device.
3. Click the **Where Used** button to see the targets that use this Elemental I/O device.
4. Click the **Close** button to return to the **Edit Elemental I/O Devices** window.

**Determining Which Resources Use a Pin**

Complete the following steps to determine which resources use a particular pin.

1. Select **Tools»Edit Elemental I/O Devices** to open the **Edit Elemental I/O Devices** window.
2. Under **Devices**, expand the device that contains the pin.
3. Expand the **Pins** folder and select the pin.
4. Click the **Where Used** button to see the resources that use this pin.
5. Click the **Close** button to return to the **Edit Elemental I/O Devices** window.

**Implementing the Elemental I/O Plug-In VIs**

**GetMethod Plug-In VI**

The GetMethod VI determines which indicators and controls the GCode plug-in VI uses for the input and output to an Elemental I/O Node.

LabVIEW runs the GetMethod VI for the Elemental I/O device to get information on the different classes of Elemental I/O. The input to the GetMethod plug-in VI is the I/O class. The output of the GetMethod plug-in VI is an array of clusters. Modify the version of the GetMethod VI the Elemental I/O Device Editor creates. You must account for every Elemental I/O class in a Case Structure. Modify the cluster in every case of the Case Structure to change the properties of the Elemental I/O Node for
that I/O class. Ignore the **Postfix Name** in the output cluster. Enable the **ShowName?** Boolean to have the Elemental I/O Node draw the contents of the **MethodName** string on the top of the Elemental I/O Node. Figure 11-2 shows an Analog Input Elemental I/O Node that does not have **ShowName?** enabled.

![Analog Input](image)

**Figure 11-2.** Elemental I/O Node Without ShowName? Enabled

Figure 11-3 shows a PWM Output Elemental I/O Node that has **ShowName?** enabled. The **MethodName** is **PWM Output**.

![PWM Output](image)

**Figure 11-3.** Elemental I/O Node With ShowName? Enabled

The **ParameterList** array in the output cluster provides the names, types of inputs, and types of outputs of the Elemental I/O Node. For the Analog Input Elemental I/O Node shown in Figure 11-2, the **ParameterList** array contains one element, with **Name** as **Analog Input**, **Direction** as **Read**, and **Type** as **U16**. For the PWM output Elemental I/O Node shown in Figure 11-3, the array has two elements, both having their **Direction** as **Write**.

The **TerminalList** array contains clusters with the items **Label**, **IO Number**, and **IO Type**. **Label** must refer to the exact name of a control or indicator on the GCode VI of the input Elemental I/O class type. **IO Number** specifies an index of the **ParameterList** array, starting at zero. This index links the control or indicator on the GCode VI with an input or output parameter of the Elemental I/O Node.

Figure 11-4 shows the Analog Input GCode VI for the Axiom CMD565. The LabVIEW code in the GCode VI is what runs in place of the Elemental I/O Node on the embedded target. The output of the GCode VI is called **Output**. The **TerminalList** in the GetMethod VI for the analog input I/O Class case contains one element, with **Label** set to **Output**, **IO Type** as **Output**, and **IO Index** as 0. This 0 refers to the first element of the **ParameterList** array.
System Description Utility Plug-In VI

Note You do not need to modify this VI unless you want to create resource contention rules.

The System Description Utility plug-in VI responds to custom events to permit or deny operations in the System Description Utility, which users use to attach a resource to an Elemental I/O Node. For example, a user might try to use two resources that share a pin in a mutually exclusive fashion in the same embedded application. Because the pins are mutually exclusive, users cannot use two resources that share the same pin. The SDUtilityPlugIn VI can detect and deny that operation.

When a user clicks the Add or Remove button in the System Description Utility, the SDUtilityPlugin VI is called to determine if the action is possible. This plug-in can enforce rules and provide error messages if an action is not allowed.

GCode Plug-In VI

Wherever an Elemental I/O Node appears on the block diagram of an embedded VI, the GCode plug-in VI for the I/O class for an I/O device replaces the Elemental I/O Node. The GCode VI must be located in a subdirectory of the I/O class for the Elemental I/O device.

All GCode VIs must have a single, top-level Sequence Structure. All controls and indicators in the GCode VI must be outside of and wired to the Sequence Structure. Wires from indicators connecting to the outer Sequence Structure must be connected to something inside of the Sequence Structure.

When a user builds an embedded VI into an embedded application, the Sequence Structure in the GCode VI replaces all occurrences of the Elemental I/O Node with which it’s associated. The controls and indicators of the GCode VI are removed, and wires connecting to and from the
Elemental I/O Node become connected to the Sequence Structure in place of the controls and indicators. The GetMethod plug-in VI makes these connections. In Figure 11-3, the output of the Analog Input Elemental I/O Node is **Analog Input**, which is bound to the **Output** indicator in the GCode VI. Thus, when the Sequence Structure of the GCode VI replaces the Elemental I/O Node, the wire once connected to **Output** is now connected to what was wired to **Analog Input**. All controls and indicators must be on the GCode VI connector pane.

The GCode plug-in VI does not run as a subVI. Instead, the GCode plug-in VI is run in a sequence that is part of the calling VI.

**Configuration Pane Plug-In VI**

The ConfigurationPane plug-in VI is where you implement the user interface for the Elemental I/O Node configuration dialog box. This VI also is responsible for validating and storing configuration data in the embedded project (LEP) using the utility VIs.

Use the Elemental I/O Device Editor to create default ConfigurationPane plug-in VIs, which are located in the I/O class subdirectories of the Elemental I/O device. If the GCode VIs you create and implement need additional user-defined parameters, such as timing, use the ConfigurationPane VI to create the user interface. The contents of the ConfigurationPane VI appear on the **Configuration** tab of Elemental I/O Node configuration dialog box. The Elemental I/O Node automatically saves any controls that have control references that connect to the **Persistent Controls** input of the CPane - Init VI connector pane.

Use the ConfigurationPane VI to send the configuration data to the GCode VI. In the **<Stop Trigger>: Value Change** case of the event structure in the ConfigurationPane VI, add event cases for controls in the GCode VI. The case selector labels must exactly match the labels of the controls on the GCode VI. You should only have cases for GCode controls that are not mentioned in the GetMethod VI. Convert the data you wish to send to the GCode control to a variant, and wire it to the CPane - Update Terminals & Backup UI VI. Note that the controls on the GCode VI must be on the connector pane.
Showing the Implementation

By default, the implementation of Elemental I/O Nodes is hidden from users. Showing the implementation can be useful when implementing Elemental I/O and debugging. You view the implementation by right-clicking an Elemental I/O Node and selecting **Show Implementation** from the shortcut menu.

To make the **Show Implementation** menu item visible, add the following token to `LabVIEW.ini`:

```
EIO.ShowImplementation=TRUE
```
Static Memory Allocation

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

If a user builds an embedded application using the static memory model, the generated code and LabVIEW embedded runtime 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 C code generator cannot estimate the maximum amount of memory required if a block diagram contains either of the following:

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

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

To support the static memory model, all the C files must be compiled with the `CStatic` compiler directive defined and active syntax checking must be enabled.

The `CStatic` compiler directive must be defined for all source files in the embedded project and all runtime library files. When `CStatic` is defined dynamic memory management calls, such as `malloc`, `realloc`, and `free`, are defined to be no-ops. Additional function arguments are produced that are pointers to global variables the C code generator declares. The functions then use the global data in these extra function arguments instead of calling `malloc`. This is done with macros.
If you want to have both static and dynamic memory allocation in the same target you must implement both static memory and dynamic memory versions of the pre-built run-time libraries and link with the correct one at build time. An embedded application must be completely static or dynamic and cannot be both at the same time.

To enable active target syntax checking you must call the SetSMAState utility VI every time the embedded project changes state, for example, when a user opens or closes a project, switches execution target, changes build settings, and so on. The SetSMAState utility VI is located in the following directory:

```
labview embedded\resource\plugins\EmbProject\Sma
```

Most targets contain the LEP_x_OnAction VI, which reads the current memory model by accessing the project configuration data, and calls the SetSMAState utility VI to dynamically update the target syntax checker built into the LabVIEW executable. In LEPPlugins.ini the OnOpen, OnClose and OnSwitchTarget configuration tokens define the VIs that execute when the user performs these actions. Define these tokens to contain the relative path to the LEP_x_OnAction VI, and then this VI will be called every time the project is opened, closed, or the execution target is changed. You must also call the SetSMAState VI from the LEP_x_BuildSettingDlg VI when configuration changes are saved in the project.

## Limitations of the Static Memory Model

### Limitations for End-Users

- The String data type is not supported.
- Only fixed size array constants are supported. To specify the type and size of an array constant, right-click the array constant and select *Set Dimension Size* from the shortcut menu.
- You cannot access the String element of the error cluster
- The Insert Into Array function is not supported.
- You can index tunnels in a For Loop, but you must define the number of iterations. The target syntax checker does not support constant folding, so you must wire a constant to the count (N) terminal of the For Loop.
• You cannot wire the name of the Timed Loop external source to the Source input because the string data type is unsupported. Enter the name of the external timing source in the Source name text box in the Loop Configuration dialog box. Open the Loop Configuration dialog box by right-clicking the Timed Loop on the block diagram and selecting Configure Timed Loop from the shortcut menu. Remove any checkmarks from the Use terminal checkbox. Enter 1 in the Period box.

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

• You cannot wire string constants to the CreateExternalTSource VI, FireExternalTSource VI, or DeleteExternalTSource 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 function uses a string input to specify the name of the RTFIFO instance. You specify the name using a fixed size array of bytes, which is the same way you specify the external timing source names. Refer to the previous bullet for more information.

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

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

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

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

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

• TCP/UDP functions are not supported.

• The LabVIEW analysis library is not supported.

**Limitations for OEMs**

- The NODE_MAX macro limits the number of external timing sources.
- The FIFO_MAX macro defines the number of RT FIFO instances you can use and does not support array elements. The FIFO_SIZE_MAX macro defines the maximum number of RT FIFO elements.
Chapter 12 Static Memory Allocation

- 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.
Implementing Instrumented Debugging

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

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

Implementing the Instrumented Debugging Communication Layer for LabVIEW

Implementing the instrumented debugging communications layer in LabVIEW Embedded Development Module requires providing four plug-in VIs for an embedded target: open, close, read, and write. You must specify the location of these VIs in LEPPlugins.ini, which is located in the target folder in the following directory:

```
labview embedded\resource\LabVIEW Targets\Embedded
```

You must modify the debug plug-in VIs to implement support to initiate the debug connection. The debug plug-in VIs only implement a communications layer. No parsing or interpretation of the data is necessary because LabVIEW interprets the debug data on the host computer and the run-time code interprets the debug data on the embedded target.
LEP_x_Debug.vi

The LEP_x_Debug.vi plug-in VI must call nitargetStartDebug.vi, which is located in the following directory:

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

nitargetStartDebug.vi has one input on its connector pane, VI Path. Wire the top-level VI of the VI hierarchy for the VI you are building to this input to begin an instrumented debugging session.

Modifying the LEPPlugins.ini File

When the instrumented debugging session starts, LabVIEW uses the target’s LEPPlugins.ini file to identify the four debug plug-in VIs that must be present to communicate with the embedded target.

Add the new keys Read, Write, Open, and Close to the LEPPlugins.ini file. The keys are paths you set to the corresponding VIs, relative to the LEPPlugins.ini file. For example, the Axiom CMD565, VxWorks Module target has the following entries in its LEPPlugins.ini file:

Read=vxworks_cmd565_LEP_TargetPlugin\LEP_vxworks_cmd565_Read.vi
Write=vxworks_cmd565_LEP_TargetPlugin\LEP_vxworks_cmd565_Write.vi
Open=vxworks_cmd565_LEP_TargetPlugin\LEP_vxworks_cmd565_Open.vi
Close=vxworks_cmd565_LEP_TargetPlugin\LEP_vxworks_cmd565_Close.vi

Implementing the Host Plug-In VIs

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

- Open VI
- Close VI
- Read VI
- Write VI

You can use an existing target example as a starting place. Refer to the LabVIEW Embedded Development Module Example Targets section to
help you determine which example target to use. Place these VIs in the target embedded project directory.

labview embedded\resource\LabVIEW Targets\Embedded\x

LabVIEW provides examples of serial and TCP/IP implementations of host side communication. The examples are located in the following directories:

- **Serial implementation**—`labview embedded\vi.lib\LabVIEW\Targets\TargetShared\serial`
- **TCP/IP implementation**—`labview embedded\vi.lib\LabVIEW\Targets\TargetShared\tcp`

To use these VIs, you must still implement the following target plug-in VIs, but the implementation of these VIs can call the serial and TCP/IP implementations.

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

**Open VI**

Establishes a connection with the embedded target.

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

- **error in (no error)**—An error cluster control.
- **error out**—An error cluster indicator.
- **connection data**—A variant indicator. Place data for the debug connection in this variant. This variant is passed in to all of the other debugging plug-in VIs, and is modifiable so you can communicate between plug-in VIs you call at different times.

**Close VI**

Closes the connection with the embedded target.

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

- **error in (no error)**—An error cluster control.
- **error out**—An error cluster indicator
- **connection data**—A variant control. The contents of this variant are user-defined, and can be modified so you can communicate between plug-in VIs you call at different times.
Read VI
Reads data from the embedded target.

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

- **error in (no error)** — An error cluster control.
- **error out** — An error cluster indicator.
- **connection data in** — A variant control. The contents of this variant come from the last invocation of an instrumented debugger plug-in VI.
- **connection data out** — A variant control. The contents of this variant are user-defined, and can be modified so you can communicate between plug-in VIs you call at different times.
- **bytes to read** — An I32 numeric control. This is the number of bytes read from the embedded target.
- **connection ID** — An I32 numeric control.

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

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

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

Write VI
Writes data from LabVIEW to the embedded target. The write must be blocking—when the VI finishes, LabVIEW expects the data to be sent to the embedded application.

Write VI must have the following five items on the connector pane:

- **error in (no error)** — An error cluster control.
- **error out** — An error cluster indicator
- **connection data in** — A variant control. The contents of this variant come from the last invocation of an instrumented debugger plug-in VI.
- **connection data out** — A variant control. The contents of this variant are user-defined, and may be modified so you can communicate between plug-in VIs you call at different times.
data to send—A string control. The contents of this string are sent to
the embedded target.

Implementing the Instrumented Debugging Communication Layer
on the Embedded Target

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

- Connect
- Disconnect
- Write
- Read
- Bytes Available

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

Tip Give your functions new, unique names to avoid linking errors.

LabVIEW Embedded Development Module provides examples of serial
and TCP/IP implementations of target side communication. The examples
are located in the following c files:

- **Serial implementation**—labview
  embedded\CCodeGen\comms\SerialMessaging.c
- **TCP/IP implementation**—labview
  embedded\CCodeGen\comms\TCPMessaging.c

To use these C functions, you must still implement the following target
plug-in VIs, but the implementation of these VIs can call the TCP and serial
implementations.

**Connect**

eRunStatus DebugConnect(void)

Opens the connection to LabVIEW Embedded Edition. It should establish
a connection to the Open debugging plug-in VI. Return eFail if there is an
error. Otherwise, return eFinished.
**Disconnected**

eRunStatus DebugDisconnect(void)

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

**Write**

eRunStatus DebugWrite(const char* pData, uInt32 size)

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

**Read**

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

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

**Bytes Available**

eRunStatus DebugBytes(uInt32* bytes)

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

**Using the Instrumented Debugging Method**

To build a LabVIEW Embedded application that uses instrumented debugging, you must give the appropriate arguments to the C Code Generator in LabVIEW Embedded Edition. Refer to the Defining the Code Generation Options chapter for more information.

**Assigning to Function Pointers**

Assign the target side instrumented debugging functions to the global LabVIEW Embedded run-time function pointers in the run-time source file labview embedded\CCodeGen\libsrc\comms\DebugComm.c, in the function PDADebugInitializeComm(). Add a new C preprocessor
identifier that describes the debugger connection mechanism. Pre-existing
testifiers include UsesTCPDebugger, UsesSerialDebugger, and
UsesCANDebugger. When you script the compiler to build an embedded
application, verify you are defining the new identifier if you are
building a debuggable application. Make a new #elif case in
PDADebugInitializeComm(), and assign the follow function pointers:

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

Troubleshooting

You can configure LabVIEW to create an embedded debugger diagnostic
file. To create this diagnostic file, add the line LogPDAMessages=True to
the LabVIEW.ini file, located in labview embedded\LabVIEW
creates a file called labview embedded\pdmsglog.txt, which
contains various debugging diagnostic and error messages that occur as a
result of instrumented debugging connections. Use this file to help you
implement and troubleshoot the connection implementation.

Implementing On Chip Debugging

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

Users enable on chip debugging in the Build Options 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.

The On Chip debug 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) debug mode affects the execution of the embedded application because this mode adds extra checkpoint code for all possible LabVIEW checkpoints. You can use the On Chip (Release) debug mode with hardware breakpoints to debug embedded applications running in ROM because the number of hardware breakpoints is limited. Use the On Chip (Release) debug mode if your C compiler cannot generate lists files you can use to map lines of generated C code to the relative symbol offsets.

Note Not all targets support on chip debugging. On chip debugging might require special debugging hardware, such as JTAG, BDM, or a Nexus emulator.
This section uses C code and the .lvm file of simple_math.vi shown in Figure 13-2.

**.lvm File**

The .lvm file is an ASCII text file the debug database generates. It contains a map of where all possible signals, breakpoints, controls, and indicators in the physical memory of the target device. LabVIEW assigns each object an object ID. The .lvm file allows each object ID to be resolved to a physical memory address on the target device. Also, you can map physical memory addresses back to the associated LabVIEW object IDs. This allows for real-time interaction between the LabVIEW development environment and the embedded application.

When you select On Chip debugging in the project build settings, the LabVIEW C Code Generator adds a set of comments and macros to the generated C code to notate where potential probes, breakpoints, controls, and indicators might be placed as shown below:

```c
GVB[0x10]/ * n: MultiplyConstant */ = 5.00000000000000000000E+2;
GVB[0x10]/ * n: DivideConstant */ = 1.00000000000000000000E+2;
GVB[0x10]/ * n: NumberIn */ = NumberIn_ESS;
/**
 * Multiply */
/**)
OCX_CHECK_POINT(simple_math_OC3F1Flag[0] = 0x1, simple_math_OC3F1Flag, 0/* OCD::BreakPoint::Node[1, 2, 3]
GVB[0x0]/ * n: Multiply: x*y /* = GVB[0x0]/ * n: NumberIn */ = 5.00000000000000000000E+2;
/**
 * Divide */
/**)
OCX_CHECK_POINT(simple_math_OC3F1Flag[0] = 0x2, simple_math_OC3F1Flag, 1/* OCD::BreakPoint::Node[0, 3, 3]
GVB[0x0]/ * n: Divide: x/y */ = (float64)GVB[0x0]/ * n: Multiply: x*y */(float64)1.00000000000000000000E+2;
if (NumberOut__l1520) {
  *NumberOut__l1520 = GVB[0x0]/ * n: Divide: x/y */;
}
return eFinished;
```
.c Parser

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

```
[simple math.vi]
Node(1, 2, 3}@63=0xA3E94
Node(2, 3, 3}@68=0xA3E94
SID_2360=0xB0E70
SID_2212=0xB0E78
SID_3104=0xB0E80
SID_1988=0xB0E88
SID_1332=0xB0E90
```

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

```
Node(a, b, c}@x=...
```

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

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

```
CID_i=...
SID_i=...
```

where CID represents Control ID, SID represents Signal ID, and i represents the unique LabVIEW-generated object ID.

C Compiler

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

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

_lst Parser_

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

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

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

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

```
[simple math.vi]
Node(1, 2, 3)@64=simple_math_Run+0x58
Node(2, 3, 3)@68=simple_math_Run+0x6C
```
The list parser also populates the debug database with information about the compiler byte packing. These descriptions are based on compiler-specific information, and is necessary so that the debug daemon knows what bytes correspond to the actual binary data value of the control or signal. If you are sure that the compiler alignment options are fixed and you are not providing that level of control to users, this information can be generated as a static string. The bold portion in the example below 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 embedded\CCodeGen\include\frontpanel
```

The members of the C data structure shown below mirror those listed in the .lvm file.

```
typedef struct {
  uInt8 bStatic;  // 0000 0000
  uInt8 bVal;    // 0000 0070
  uInt8 bInput;  // 0000 0000
  uInt8 padding1;
} BooleanData;
```

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

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

where

- $j$ represents the offset from the beginning of the data structure in bits
- $k$ represents the length of the data member in bits
- $L$ represents the length of the entire data type in bits
Repeat this process for each of the following data types:

- BooleanData
- NumericData
- StringData
- _ControlData
- _OCDI_Alignment
- _PDAStr

In addition, the .lvm file needs information from the LabVIEW Embedded Project to completely decouple the LabVIEW development environment and the debug interface.

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

where x represents 1 or 0 based on the information from the LabVIEW Embedded Project.

**Linker**

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

**.map Parser**

In the first stage, the map parser runs through the map file generated by the linker and creates a structure that maps symbol names to absolute addresses of the built image. The map parser is a linker specific component that you must implement in the event your linker does not produce the map file compatible with the example map parser. In the second stage, the map parser resolves all symbols in the debug database. The map parser is not executed for targets that build a downloadable module. All symbols must be resolved during the module download.
Map file parsing resolves the previously unresolved symbols and converts the relative addresses the list parser generates to physical address for the embedded target. The resolved addresses are shown below in the resulting .lvm file.

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

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

### Debug Database

The debug database is a collection of records for all controls, signals, and breakpoints in the VI hierarchy for the application. The database is initialized before the first .c file is compiled, usually in the setup stage of the build VI. The .c parser inserts records for all signals, controls, and breakpoints. The list parser maps the breakpoint line numbers into relative symbol offsets and updates all breakpoint records. Finally, the map parser resolves all symbols and updates all database records so they contain absolute addresses of the built image. The database snapshot is then saved into the .lvm file locate in the project build folder. The .lvm file is automatically loaded by the debug daemon 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. The debug daemon re-indexes the database every time the .lvm file is loaded instead.

### Debug Daemon

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

**Emulator Plug-in**

The emulator plug-in implements a low-level emulator interface the debug daemon uses. In most cases, the emulator plug-in translates calls of the debug daemon interface into the API of the emulator vendor.

**Adding Support for an Existing Emulator to a New Target**

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

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

2. Call the `EMB_Debug_Map_Gcc_UpdateDD.vi` for every compiled generated source file. The `UpdateDD VI parses the source file and the list file the compiler generates and updates the debug database. This VI should be called only if the OnChip debug mode is enabled in the **Build Options** dialog box. Note that compiler must be configured to generate the list file that the list file parser `EMB_Debug_Map_Gcc_UpdateDD.vi` calls can understand. Refer to the following example batch file, which executes a GCC compiler to compile the C file and generate the list file in stab format.

   ```batch
   labview embedded\resource\LabVIEW Targets\Embedded\vxworks\cmd565\utils\compiled.bat
   ```

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

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

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

   ```batch
   labview embedded\vi.lib\LabVIEW Targets\Embedded\Debug\Map\GCC
   ```
You must implement your new parser for a non-GCC compiler. Refer to the LEP_vxworks_cmd565_ScriptCompiler.vi for information about retrieving configuration data, defining the release debug macro, and calling the update debug database VI to parse source and list files.

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

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

### 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.

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

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

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

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

5. Change the implementation of all VIs in the root directory. Implementation of some methods is optional.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IInit</td>
<td>Establishes the debug connection to the hardware and returns the breakpoint occurrence. The emulator plug-in is responsible for setting the breakpoint occurrences. The plug-in can spawn a background VI that pools low-level emulator API for events and sets breakpoint occurrence as needed.</td>
</tr>
<tr>
<td>IRelease</td>
<td>Shuts down all background VIs and closes the connection to the emulator API.</td>
</tr>
<tr>
<td>ISetBreakpoints</td>
<td>Sets breakpoints on given absolute addresses.</td>
</tr>
<tr>
<td>IClearBreakPoints</td>
<td>Clears breakpoints on given absolute addresses.</td>
</tr>
<tr>
<td>IGetBreakpoint</td>
<td>Retrieve the current breakpoint.</td>
</tr>
<tr>
<td>IContExecution</td>
<td>Resumes execution of the target.</td>
</tr>
<tr>
<td>IResolveSymbols</td>
<td>Resolves given symbols to absolute addresses. You do not need to implement this if you have a plug-in that supports downloadable module, VxWorks for example.</td>
</tr>
<tr>
<td>IMemoryRead</td>
<td>Reads memory blocks from target.</td>
</tr>
<tr>
<td>IMemoryWrite</td>
<td>Writes memory blocks to target.</td>
</tr>
<tr>
<td>IGo</td>
<td>Executes the downloaded image.</td>
</tr>
<tr>
<td>IDownload</td>
<td>Downloads the built application to target memory.</td>
</tr>
<tr>
<td>IConfig</td>
<td>Invokes an emulator-specific configuration dialog box. Called from the OCDI Options dialog box.</td>
</tr>
<tr>
<td>IExit</td>
<td>Called when you close an embedded project (LEP) or switch the execution target. Implementation of this plug-in is optional.</td>
</tr>
<tr>
<td>IStop</td>
<td>Halts the target.</td>
</tr>
</tbody>
</table>
Chapter 13  Debugging

Adding the List Parser for Non-GCC Compiler

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

You must implement the following VIs:

**EMB_Debug_Map_x_UpdateDD.vi**

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

**EMB_Debug_Map_Gcc_ResolveSymbols.vi**

Parses the `.map` file the linker generates and resolves all objects in the debug database.

Examples

The following targets support OCDI debugging:

- Axiom CMD565, VxWorks RAM Image
- Intel IXDP425, VxWorks RAM Image
- Axiom CMD565, VxWorks Module

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

`labview embedded\lvmerce\vi.lib\LabVIEW Targets\Embedded\Debug\Map`

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

`labview embedded\vi.lib\LabVIEW Targets\Embedded\Debug\OCDI`
Customizing the Editing Environment

You can customize the LabVIEW editing environment so only features a target supports is available. To customize the editing environment, you must edit the palette menus and configure the target syntax.

Editing the Palette Menus

You can edit palette menus to remove features from the palettes that the target does not support. For example, you can remove the TCP/IP VIs for targets without Ethernet connectivity. Refer to the LabVIEW Help, available by selecting Help»VI, Function, & How-To Help, for information about editing palette menus.

Configuring the Target Syntax for a New Target

Target syntax determines the supported and unsupported features for a specific target. Configuring the target syntax limits features you or other users can use when developing an embedded application to the features the target can support. It also prevents you or other users from importing VIs from other targets that are unsupported for another target.

Select Tools»Configure Target Syntax to open the Configure Target Syntax dialog box. Use the Configure Target Syntax dialog box to create the XML file LabVIEW uses to check the target syntax.

Note This process overwrites targetsyntax.xml, which LabVIEW uses to determine target syntax. National Instruments recommends that you back up targetsyntax.xml before modifying it.
Complete the following steps to configure the target syntax for a new target.

1. Copy `labview\examples\TargetSyntax` to another location. This directory contains the VIs that determine the target syntax. Features that the C Generator does not support are already included.
   a. `LVEDefaultNumericType.ctl` determines the default data type of front panel controls. For example, if you are creating an integer-only target and want the default data type for numeric controls to be `int32`, create a custom `int32` control.
   b. `LVEUnsupportedNodes.vi` determines which nodes the target does not support.
   c. `LVEUnsupportedFunctions.vi` determines which functions the target does not support.
   d. `LVEUnsupportedTypes.vi` determines which data types the target does not support.
   e. `LVEUnsupportedVIs.vi` determines which VIs the target does not support.

2. Edit the VIs described in step 1 by placing unsupported features on the block diagram of the appropriate VI. Save the VI.

   **Note** For targets with very limited functionality, you can create a "supported" VI that contains only the features that the target can support.

3. Select **Tools»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 `vi.lib`.

8. On the **Default Type** tab, add the default data type.
   a. Navigate to and select `LVEDefaultNumericType.ctl`, which is the custom control that contains the default data type.

9. (Optional) On the **Misc.** tab, adjust the settings for maximum recursion depth, variable support, and so on. You usually can keep the default miscellaneous settings.
   - **Max Type Depth** specifies the maximum recursion depth of supported data types. For example, a cluster of an array of integers has a type depth of 3. A cluster of an array of a cluster of strings has a type depth of 4. A value of –1 ignores the maximum type depth.
   - **Max Array Dimensions** specifies the maximum number of dimensions arrays allow. A value of –1 ignores the maximum array dimension.
   - **Support global variables** specifies if the target supports global variables.
   - **Support local variables** specifies if the target supports local variables.
   - **Errors break the VI** specifies if target syntax errors break the VI.
   - **Errors are considered warnings and do not break the VI** specifies if target syntax errors are considered warnings.
A few situations where you might want to change the Misc. settings include:

- You can set the **Max Type Depth** to prevent nested types from being recursive, which is not permitted by MISRA (The Motor Industry Software Reliability Association) automotive coding standards.
- You can set the **Max Array Dimensions** to 1 to restrict users to only optimized 1D array operations to achieve higher performance.

10. Enter the directory path to the target in the **Location of target** field.
11. Click the **Generate** button to create **TargetSyntax.xml**, which is the file LabVIEW uses to check target syntax, in the target directory you specified in step 10.

## Modifying the Target Syntax for a Target

Use the **Configure Target Syntax** dialog box to modify the XML file LabVIEW uses to check the target syntax.

Complete the following steps to modify the target syntax for a target.

1. Modify the “unsupported” and/or “supported” target syntax VIs. For example, if you want to add support for a VI, remove that VI from the “unsupported” VI.
2. Select **Tools»Configure Target Syntax** to open the **Configure Target Syntax** dialog box.
3. Click the **Open** button.
4. Navigate to and select the **TargetSyntax.xml** file for the target you are modifying.
5. Click the **Generate** button to regenerate the **TargetSyntax.xml** file, which is the file LabVIEW uses to determine target syntax.
Use these tutorials to learn how to create a new embedded target and how to implement Elemental I/O.

Creating a New Target

Use this tutorial to copy an existing target and modify the target to support integer-only embedded applications. The new target you will create can build a VxWorks application for a CMD 565 target that does not use any floating-point operations.

Use the Axiom CMD565, VxWorks Module example target.

labview embedded\resource\LabVIEW Targets\Embedded\vxworks\cmd565

Part 1—Using the Target Creation Utility

Complete the following steps to create a new target from the Axiom CMD565, VxWorks Module target.

1. Select **Tools»Create Embedded Target** to open the **Create Embedded Target** window as shown in Figure 15-1.

![Create Embedded Target Window](image)

**Figure 15-1.** Create Embedded Target Window

2. Navigate to and select the Axiom CMD 565, VxWorks Module target directory in the **Template Target** path control.
3. Enter CMD 565, VxWorks, Integer Only in the Target name text box. This is the name that appears in the Execution Target list.

4. Enter the following path in the Target folder path path control.

5. Click the Create Target button to create the new target. Click the Save button if prompted.

6. Click the OK button and then click the Close button to close the Create Embedded Target window.

7. Navigate to and delete the ram and rom folders. These folders are a result of the target creation process and are unusable targets.

8. Exit and relaunch LabVIEW.

Part 2—Adding a New CGen Step

Complete the following steps to copy some VIs from the parent target to modify for new behavior. All paths in this section are relative to the following directory:

1. Open LEP_vxworks_cmd565int_Build.vi from the following directory:

2. Rename the VIs in the three string constants on the block diagram to have cmd565int in the name instead of cmd565. The three VIs in the Case Structure have already been relinked to their new names by the target creation utility. But LEP_vxworks_Build.vi in the sequence is still coming from the parent target. LEP_vxworks_Build.vi coordinates the build process including the C code generation, compile, and link steps. The CGen step is unwired, which means that the LEP_vxworks_Build.vi will use the default behavior inherited from the parent target. We want a new CGen step that uses a different input for integer only, so we will copy the CGen VI from the parent target into the new target, modify it, and then pass a reference to it into the parent build VI.
3. Copy LEP_vxworks_CGen.vi from vxworks_LEP_TargetPlugin to the following directory:

\...\cmd565int\vxworks_cmd565int_LEP_TargetPlugin

4. Rename LEP_vxworks_CGen.vi you just copied to LEP_vxworks_cmd565int_CGen.vi.

5. Open LEP_vxworks_cmd565int_Build.vi and place an instance of LEP_vxworks_cmd565int_CGen.vi inside the Case Structure with the other VIs.

6. Place another string constant with the text LEP_vxworks_cmd565int_CGen.vi and wire the constant to a new Open VI reference function. The type specifier refnum input of the Open VI Reference should be the following:

labview embedded\resource\LabVIEW Targets\embedded\vxworks\vxworks_LEP_TargetPlugin\LEP_vxworks_CGenRef.vi

7. Wire the opened reference to the CGen step input of the VI inside the Sequence Structure and to a new Close Reference to the right of the sequence as shown in Figure 15-2.

![Figure 15-2. Wiring the Opened Reference to the CGen Step](image)

8. Modify the CGen VI to have the new behavior to generate integer only C code. In LEP_vxworks_cmd565int_CGen.vi, expand the cluster unbundled on the left to include the **Integer only** flag. Expand the cluster bundle on the right to include the **GenerateIntegerOnly** flag.
9. Wire the **Integer only** flag of the unbundled cluster to the **GenerateIntegerOnly** flag of the bundled cluster as shown in Figure 15-3.

![Figure 15-3. Wiring the Integer Only Flag to the GenerateIntegerOnly Flag](image)

**Part 3—Modifying the Build Settings Dialog Box**

Complete the following steps to modify the **Build Settings** dialog box so the user can select the integer only build option. All paths in this section are relative to the following directory:

```
labview embedded\resource\LabVIEW Targets\embedded\vxworks\cmd565int
```

1. Open `LEP_vxworks_cmd565int_BuildSettingsDefault.vi`.
2. Open the **Default** typedef and add a new check box to the end called **Integer only**.
3. Save the typedef as `LEP_vxworks_cmd565int_Config.ctl`. 
4. Add a new array element to the **Attribute name** front panel array control. Name the new array element **GenerateIntegerOnly**.

5. Select **Operate»Make Current Values Default** and save the VI.

6. Open **LEP_vxworks_cmd565int_BuildSettingsDlg.vi**.

7. Modify the front panel to include an integer only checkbox as shown in Figure 15-4.

8. Add a control reference to the block diagram and wire it up to the build array as shown in Figure 15-5. The value of the integer only control is saved in the project data when the end user clicks the **OK** button. When the build process starts, it will be read back into the configuration typedef you edited before and passed to the CGen VI server call.
9. Open `LEP_vxworks_cmd565int_LoadBuildConfig.vi` and replace the typedef named `Config` with the new one named `LEP_vxworks_cmd565int_Config.ctl`.

### Part 4—Modifying the Compile Step

Complete the following steps to modify the compile step. All paths in this section are relative to the following directory:

```
labview embedded\resource\LabVIEW Targets\embedded\vxworks\cmd565int\vxworks_cmd565int_LEP_TargetPlugin
```

1. Open `LEP_vxworks_cmd565int_ScriptCompiler.vi`. Add code to read the `GenerateIntegerOnly` setting from the Project Data as shown in Figure 15-6. Use the value to set a compile flag that is passed to the `.bat` file that actually calls the compiler.
Implementing Elemental I/O

This tutorial shows you how to use the Elemental I/O Device Editor to create an Elemental I/O device, and how to implement and configure the I/O routines that run on an embedded target.

Part 1—Creating a New Elemental I/O Device

Complete the following steps to create a new Elemental I/O Device.

1. Select Tools»Edit Elemental I/O Devices to open the Edit Elemental I/O Devices window as shown in Figure 15-7.
2. Select Devices in the list to expand and view the devices.
3. Click the Add button to open the Add Elemental I/O Device Wizard.
4. Name the device by entering MyDevice in IO device name.
5. Click the Next button.

6. Select the Elemental I/O classes.
   a. Add analog input to MyDevice by selecting AI in the All EIO classes list and clicking the Add button.
   b. Add digital output to MyDevice by selecting DO in the All EIO classes list and clicking the Add button.

7. Click the Next button.
8. Add an analog input resource to the Elemental I/O device.
   a. Click the Add button to open the Add Resources dialog box.
   b. Enter AI in the Resource prefix text box.
   c. Enter 1 in the Start index text box.
   d. Enter 1 in the Number of resources text box.
   e. Select Analog Input from the I/O Class list.
   f. Click the OK button.

![Image of Add Resources dialog box]

Figure 15-10. Adding Resources

9. Add a digital output resource to the Elemental I/O device.
   a. Click the Add button to open the Add Resources dialog box.
   b. Enter DO in the Resource prefix text box.
   c. Enter 1 in the Start index text box.
   d. Enter 1 in the Number of resources text box.
   e. Select Digital Output from the I/O Class list.
   f. Click the OK button.
10. Verify you see the AI1 and DO1 resources in the **Add IO Device Wizard** as shown in Figure 15-11.

![Figure 15-11. Verifying the Resources](image)

11. Click the **Next** button.

12. Add pins.
   a. Click the **Add** button to open the **Add Pins** dialog box.
   b. Enter **PIN** in the **Pin prefix** text box.
   c. Enter **2** in the **Number of pins** text box.
   d. Click the **OK** button.

![Figure 15-12. Adding Pins](image)

13. Click the **Next** button.

14. Assign pin PIN1 to the AI1 resource.
   a. Select **PIN1** in the **Pins** list.
   b. Select **AI1** in the **Resources** drop-down list.
   c. Click the **Add** button. PIN1 is now assigned to the AI1 resource.
15. Assign pin PIN2 to the DO1 resource.
   a. Select PIN2 in the Pins list.
   b. Select DO1 in the Resources drop-down list.
   c. Click the Add button. PIN2 is now assigned to the DO1 resource.

16. Click the Next button.

17. Add the I/O device to the Windows Console Application target by selecting Windows Console Application in the Available targets list.
18. Click the **Finish** button.

19. Under **Devices**, expand **MyDevice** as shown in Figure 15-16.

![Figure 15-15. Adding the Device to Targets](image)

![Figure 15-16. MyDevice](image)
Part 2—Implementing I/O for MyDevice

Complete the following steps to implement I/O for the Elemental I/O device you created in Part 1—Creating a New Elemental I/O Device.

1. Select Tools»Edit Elemental I/O Devices to open the Edit Elemental I/O Devices window if it is not open.

2. Expand Devices, expand Elemental I/O Classes, and then expand AI. Double-click eio_MyDevice_AI_GCode.vi under AI to open the GCode VI for the Analog Input of your Elemental I/O device.


4. Enter `RESULT = 123;` on a new line in the Inline C Node to make the Analog Input Elemental I/O Node always return 123 when the Elemental I/O node executes. Save and close the VI.

5. In the Edit Elemental I/O Devices window, double-click eio_DO_GCode.vi under DO.

6. Add the line `printf("%d\n", INPUT);` to the Inline C Node to have the Digital Output Elemental I/O Node print the value of its input whenever the Elemental I/O node runs on an embedded target. Save and close the VI.
7. Click the **Quit** button to close the **Edit Elemental I/O Devices** window.

8. Select **Operate»Switch Execution Target** and select **Windows Console Application**.

9. Open a blank VI.

10. Place an Analog Input Elemental I/O Node on the block diagram.

11. Double-click the Analog Input Elemental I/O Node to configure it for the Windows Console Application target.
   a. Click the **Edit System Description** button to open the **System Description Utility**.
   b. Add the AI1 and DO1 resources by selecting the resource in the **Resources Available** list and clicking the **Add** button.
   c. Click the **OK** button to return to the **Configure Elemental I/O** dialog box.
   d. On the **General** tab on the **Configure Elemental I/O** dialog box, select AI1 in the **Resource** list.
   e. Click the **OK** button.
12. Place a Digital Output Elemental I/O Node on the block diagram.

   a. On the **General** tab on the **Configure Elemental I/O** dialog box, select DO1 in the **Resource** list.
   b. Click the **OK** button.

14. Create a True Boolean constant and wire the constant to the Digital Output Elemental I/O Node.

15. Place the Number to Decimal String function on the block diagram.

16. Wire the output of the Analog Input Elemental I/O Node to the **Number** input of the Number to Decimal String function.

17. Place a One Button Dialog function on the block diagram.

18. Wire the **decimal integer** output of the Number to Decimal String function to the **message** input of the One Button Dialog function.

19. Click the **Run** button to save the VI and create a new Embedded Project.

20. In the **Embedded Project Manager** window, click the **Build** button to build the VI as an embedded application.

21. In the **Embedded Project** window, click the **Run** button to run the embedded application on the host computer. You should see two values displayed:
   - 123 comes from the **Analog Input** Elemental I/O Node.
   - 1 comes from the `printf` C statement in the Digital Output Elemental I/O Node.

---

**Figure 15-19.** Wiring the Number to Decimal String Function to the One Button Dialog Function
Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Online technical support resources at ni.com/support include the following:
  - **Self-Help Resources**—For answers and solutions, visit the award-winning National Instruments Web site for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on.
  - **Free Technical Support**—All registered users receive free Basic Service, which includes access to hundreds of Application Engineers worldwide in the NI Developer Exchange at ni.com/exchange. National Instruments Application Engineers make sure every question receives an answer.

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

- **Training and Certification**—Visit ni.com/training for self-paced training, eLearning virtual classrooms, interactive CDs, and Certification program information. You also can register for instructor-led, hands-on courses at locations around the world.

- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. To learn more, call your local NI office or visit ni.com/alliance.

If you searched ni.com and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.