Contents

About This Guide  xix
Typographical conventions  xxi
  Note to Windows users  xxii
How to use this guide  xxiii
Assumptions  xxv

1  IDE Concepts  1
What is an IDE?  3
  An IDE for building embedded systems  3
Starting the IDE  4
  Starting the IDE for the first time  5
  Starting the IDE after an update  8
  Starting the IDE from the command line  9
Workbench  10
Help system  11
  Opening the IDE Help  11
  Navigating the Help  11
  Help bookmarks  13
  Tips and tricks  13
Perspectives  13
Views and editors  14
  Views  14
  Editors  15
Projects and workspace  16
  Specifying a workspace location  16
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>How the IDE looks at projects</td>
<td>17</td>
</tr>
<tr>
<td>Host and target machines</td>
<td>18</td>
</tr>
<tr>
<td>Target agent (the qconn daemon)</td>
<td>18</td>
</tr>
<tr>
<td>Launcher</td>
<td>19</td>
</tr>
<tr>
<td>Resources</td>
<td>19</td>
</tr>
<tr>
<td>Wizards</td>
<td>19</td>
</tr>
<tr>
<td>Keyboard shortcuts</td>
<td>19</td>
</tr>
<tr>
<td>Preferences</td>
<td>20</td>
</tr>
<tr>
<td>Version coexistence</td>
<td>21</td>
</tr>
<tr>
<td>Specifying which OS version to build for</td>
<td>23</td>
</tr>
<tr>
<td>Environment variables</td>
<td>24</td>
</tr>
<tr>
<td>What’s new in the IDE?</td>
<td>24</td>
</tr>
<tr>
<td>What’s new in 6.3.0 SP1?</td>
<td>24</td>
</tr>
<tr>
<td>What’s new in 6.3.0 SP2?</td>
<td>25</td>
</tr>
<tr>
<td>General IDE</td>
<td>25</td>
</tr>
<tr>
<td>C/C++ user interface</td>
<td>33</td>
</tr>
<tr>
<td>C/C++ debug and launch</td>
<td>36</td>
</tr>
<tr>
<td>C/C++ project configuration and build</td>
<td>41</td>
</tr>
<tr>
<td>C/C++ editing and source navigation</td>
<td>45</td>
</tr>
<tr>
<td>QNX Momentics tools</td>
<td>51</td>
</tr>
<tr>
<td>2 Preparing Your Target</td>
<td>59</td>
</tr>
<tr>
<td>Host-target communications</td>
<td>61</td>
</tr>
<tr>
<td>IP communications</td>
<td>61</td>
</tr>
<tr>
<td>Serial communications</td>
<td>62</td>
</tr>
<tr>
<td>Example: Debugging via PPP</td>
<td>63</td>
</tr>
<tr>
<td>Connecting with Phindows</td>
<td>67</td>
</tr>
<tr>
<td>3 Developing C/C++ Programs</td>
<td>69</td>
</tr>
<tr>
<td>The C/C++ Development perspective</td>
<td>71</td>
</tr>
<tr>
<td>Wizards and Launch Configurations</td>
<td>72</td>
</tr>
<tr>
<td>Controlling your projects</td>
<td>72</td>
</tr>
<tr>
<td>Opening files</td>
<td>73</td>
</tr>
</tbody>
</table>
4 Managing Source Code 109
CVS and the IDE 111
Local history feature 112
Project files (.project and .cdtproject) 112
Core Eclipse documentation on using CVS in the IDE  112
Importing existing source code into the IDE  114
   Projects within projects?  115
   Filesystem drag-and-drop  121
   CVS repository  121
   Import wizard  126
   Linked resources  127
Using container projects  128
   Creating a container project  129
   Setting up a build configuration  131
   Editing existing configurations  131
Importing a BSP or other QNX source packages  134
   Step 1: Use File→Import…  134
   Step 2: Select the package  136
   Step 3: Select the source projects  137
   Step 4: Select a working set  138
   Step 5: Build  139
Exporting projects  140
   Using the Export… command  141

5  Debugging Programs  145
Introduction  147
Debugging your program  148
   Building an executable for debugging  148
   Starting your debugging session  149
Controlling your debug session  152
   Using the controls  154
   Debug launch controls  158
   Disassembly mode  159
More debugging features  159
   Inspecting variables  160
   Using breakpoints and watchpoints  164
   Evaluating your expressions  170
6 Building OS and Flash Images  181
Introducing the QNX System Builder  183
   Toolbar buttons  185
   Binary Inspector  186
   Boot script files  187
   QNX System Builder projects  187
   The scope of the QNX System Builder  188
Overview of images  188
   The components of an image, in order of booting  189
   Types of images you can create  191
   Project layout  196
   Overview of workflow  198
Creating a project for an OS image  199
Creating a project for a Flash filesystem image  200
Building an OS image  201
   Create new image  202
   Combine images  204
Downloading an image to your target  207
   Downloading via a serial link  207
   Downloading via TFTP  210
   Downloading using other methods  213
Configuring your QNX System Builder projects  213
   Managing your images  214
   Configuring image properties  217
   Configuring project properties  223
Optimizing your system  227
Optimizing all libraries in your image 227
Optimizing a single library 229
Restoring a slimmed-down library 233
Moving files between the host and target 234
Moving files to the target 235
Moving files from the target to the host 235

7 Developing Photon Applications 237
What is PhAB? 239
PhAB and the IDE 240
Using PhAB 241
Creating a QNX Photon Appbuilder project 241
Closing PhAB 244
Reopening PhAB 245
Editing code 245
Building a QNX Photon Appbuilder project 246
Starting Photon applications 247

8 Profiling an Application 249
Introducing the Application Profiler 251
Types of profiling 252
Profiling your programs 253
Building a program for profiling 254
Running and profiling a process 256
Profiling a running process 258
Postmortem profiling 261
Controlling your profiling sessions 263
Understanding your profiling data 265
Usage by line 265
Usage by function 268
Usage by thread 269
Call counts 269
9 Using Code Coverage  273
   Code coverage in the IDE  275
      Types of code coverage  275
      How the coverage tool works  276
   Enabling code coverage  278
      Enabling code coverage for Standard Make projects  278
      Starting a coverage-enabled program  279
   Controlling your session  282
   Examining data line-by-line  284
   Examining your coverage report  285
   Seeing your coverage at a glance  287

10 Finding Memory Errors  289
   Introduction  291
      Memory management in QNX Neutrino  292
      What the Memory Analysis perspective can reveal  299
   Analyzing your program  301
   Tracing memory events  305
      Memory Problems view  308
      Event Backtrace view  310

11 Getting System Information  313
   Introduction  315
   What the System Information perspective reveals  316
      Key terms  317
      The views in this perspective  319
   Controlling your system information session  320
      Sending a signal  322
      Updating the views  323
      Adding views to the System Information perspective  323
   Examining your target system’s attributes  324
      System Specifications pane  325
      System Memory pane  325
Processes panes 326
Watching your processes 326
Thread Details pane 327
Environment Variables pane 328
Process Properties pane 328
Examining your target’s memory 328
Malloc Information view 329
Virtual address space 332
Stack errors 334
Inefficient heap usage 334
Examining process signals 335
Getting channel information 336
Tracking file descriptors 338
Tracking resource usage 339
System Uptime display 339
General Resources display 340
Memory Resources display 341

12 Analyzing Your System with Kernel Tracing 343
Introducing the QNX System Profiler 345
Before you begin 347
Configuring a target for system profiling 349
Launching the System Profiler Configuration wizard 350
Selecting options in the wizard 350
Capturing instrumentation data in event log files 353
Viewing and interpreting the captured data 355
The System Profiler editor 356
Other views in the System Profiler 362

13 Common Wizards Reference 365
Introduction 367
Creating a C/C++ project 369
Contents

14 Launch Configurations Reference 429
What is a launch configuration? 431
Types of launch configurations 431
Running and debugging the first time 433
   Debugging a program the first time 434
   Running a program the first time 436
Running and debugging subsequent times 437
   Launching a selected program 438
   Launching from a list of favorites 438
   Launching the last-launched program 439
Setting execution options 439
   Main tab 440
   Arguments tab 442
   Environment tab 443
Download tab 444
Debugger tab 445
Source tab 448
Common tab 449
Tools tab 451

A Tutorials 455
Before you start… 457
Tutorial 1: Creating a Standard Make C/C++ project 457
Tutorial 2: Creating a QNX C/C++ project 460
Tutorial 3: Importing an existing project into the IDE 461
Tutorial 4: Importing a QNX BSP into the IDE 463
   Step 1: Use File→Import… 463
   Step 2: Select the package 464
   Step 3: Select the source projects 466
   Step 4: Select a working set 467
   Step 5: Build 468

B Where Files Are Stored 471

C Utilities Used by the IDE 475

D Migrating from Earlier Releases 479
Introduction 481
From 6.3.0 to 6.3.0 Service Pack 2 482
   Migrating your workspace 482
   Migrating your projects 483
   Migration issues 485
From 6.2.1 to 6.3.0 489
   Migrating your workspace 489
   Migrating your projects 491
From 6.2.0 to 6.3.0 493
   Migrating your projects 494
Glossary 497

Index 503
List of Figures

The IDE User’s Guide at a glance.   xxiii
This chapter introduces key concepts used in the IDE.   3
Selecting a workspace directory.   5
The IDE’s welcome screen.   6
The Configuration Changes dialog.   8
The Install/Update dialog tells you to restart the IDE.   9
The first thing you see.   10
The new Progress view showing the progress of a CVS checkout and a Workspace build background operation.   28
This chapter explains how to set up host-target communications.   61
This chapter shows you how to create and manage your C or C++ projects.   71
The C/C++ Editor.   90
The Tasks view lets you track your tasks.   102
This chapter describes managing source code from within the IDE.   111
The Check Out As wizard.   123
This chapter shows you how to work with the debugger.   147
Use the QNX System Builder to create OS and Flash images for your target.   183
Typical boot order.   190
Use the PhAB visual design tool to develop Photon apps.   239
This chapter shows you how to use the application profiler.   251
The QNX Application Profiler perspective.   252
Use the Code Coverage tool to help test your code.   275
Use the QNX Memory Analysis perspective to solve memory problems. 291
Process memory layout on an x86. 294
This chapter shows you how to work with the System Information perspective. 315
Use the System Profiler to analyze your system via instrumentation. 345
This chapter describes the IDE’s wizards. 367
The Import wizard. 403
Importing a container project. 405
Selecting container components to import. 406
Importing an existing project. 407
Importing code from the filesystem. 409
The Select Types dialog lets you filter imported files by selecting one or more extensions. 410
Browsing for a project or folder. 411
Importing GCC coverage data. 413
Referenced projects and comments. 414
Importing a BSP. 415
Selecting source projects from a BSP archive. 416
Selecting a working set from a BSP archive. 417
Building BSP projects. 418
Importing a mkifs .build file. 419
Selecting a destination System Builder project. 420
Importing a QNX Source Package. 421
Building package projects. 422
Importing a Team Project Set. 424
Importing code from a ZIP archive. 425
The Select Types dialog lets you filter imported files by selecting one or more extensions. 426
Browsing for a project or folder. 427
You must set up a Launch Configuration before you can run or debug a program. 431
Here are several tutorials to help you get going with the IDE. 457
This appendix shows you where to find key files used by the IDE. 473
This appendix lists the utilities used by the IDE. 477
You can easily migrate your old workspace and projects to this release. 481
Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications. The following table summarizes our conventions:

<table>
<thead>
<tr>
<th>Reference</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code examples</td>
<td><code>if( stream == NULL )</code></td>
</tr>
<tr>
<td>Command options</td>
<td><code>-lR</code></td>
</tr>
<tr>
<td>Commands</td>
<td><code>make</code></td>
</tr>
<tr>
<td>Environment variables</td>
<td><code>PATH</code></td>
</tr>
<tr>
<td>File and pathnames</td>
<td><code>/dev/null</code></td>
</tr>
<tr>
<td>Function names</td>
<td><code>exit()</code></td>
</tr>
<tr>
<td>Keyboard chords</td>
<td><code>Ctrl – Alt – Delete</code></td>
</tr>
<tr>
<td>Keyboard input</td>
<td><code>something you type</code></td>
</tr>
<tr>
<td>Keyboard keys</td>
<td><code>Enter</code></td>
</tr>
<tr>
<td>Program output</td>
<td><code>login:</code></td>
</tr>
<tr>
<td>Programming constants</td>
<td><code>NULL</code></td>
</tr>
<tr>
<td>Programming data types</td>
<td><code>unsigned short</code></td>
</tr>
<tr>
<td>Programming literals</td>
<td><code>0xFF,&quot;message string&quot;</code></td>
</tr>
<tr>
<td>Variable names</td>
<td><code>stdin</code></td>
</tr>
<tr>
<td>User-interface components</td>
<td><code>Cancel</code></td>
</tr>
</tbody>
</table>

We format single-step instructions like this:

➤ To reload the current page, press Ctrl – R.

We use an arrow (→) in directions for accessing menu items, like this:
Typographical conventions

You’ll find the **Other...** menu item under **Perspective→Show View**.

We use notes, cautions, and warnings to highlight important messages:

- **Notes**: point out something important or useful.

- **CAUTION**: Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

- **WARNING**: Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we use a forward slash (/) as a delimiter in *all* pathnames, including those pointing to Windows files.

We also generally follow POSIX/UNIX filesystem conventions.
How to use this guide

This User’s Guide describes the Integrated Development Environment (IDE), which is part of the QNX Momentics development suite. The guide introduces you to the IDE and shows you how to use it effectively to build your QNX Neutrino-based systems.

The workflow diagram above shows how the guide is structured and suggests how you might use the IDE. Once you understand the basic concepts, you’re ready to begin the typical cycle of setting up your projects, writing code, debugging, testing, and finally fine-tuning your target system.

Each chapter begins with the workflow diagram, but with the chapter’s bubble highlighted to show where you are in the book. Note that in the online version each bubble is a link.
How to use this guide

CAUTION: This release of the IDE is based on Eclipse 3.0. If you have an older version of the IDE, see the Migrating from Earlier Releases appendix in this guide.

The following table may help you find information quickly:

<table>
<thead>
<tr>
<th>To:</th>
<th>Go to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learn about the workspace, perspectives, views, and editors</td>
<td>IDE Concepts</td>
</tr>
<tr>
<td>Use the IDE’s help system</td>
<td>IDE Concepts</td>
</tr>
<tr>
<td>Connect your host and target</td>
<td>Preparing Your Target</td>
</tr>
<tr>
<td>Import existing code into the IDE</td>
<td>Managing Source Code</td>
</tr>
<tr>
<td>Import a QNX BSP source package</td>
<td>Managing Source Code</td>
</tr>
<tr>
<td>Set execution options for your programs</td>
<td>Launch Configurations Reference</td>
</tr>
<tr>
<td>Check code into CVS</td>
<td>Managing Source Code</td>
</tr>
<tr>
<td>Run QNX Neutrino on your target</td>
<td>Building OS and Flash Images</td>
</tr>
<tr>
<td>Examine execution stats (e.g. call counts) in your programs</td>
<td>Profiling an Application</td>
</tr>
</tbody>
</table>

continued...
### Assumptions

This guide assumes the following:

- On your host you’ve already installed the QNX Momentics suite, which includes a complete QNX Neutrino development environment.

- You're familiar with the architecture of the QNX Neutrino RTOS.

- You can write code in C or C++.

---

<table>
<thead>
<tr>
<th>To:</th>
<th>Go to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exercise a test suite</td>
<td>Using Code Coverage</td>
</tr>
<tr>
<td>Find and fix a memory leak in a program</td>
<td>Finding Memory Errors</td>
</tr>
<tr>
<td>See process or thread states, memory allocation, etc.</td>
<td>Getting System Information</td>
</tr>
<tr>
<td>Examine your system’s performance, kernel events, etc.</td>
<td>Analyzing Your System with Kernel Tracing</td>
</tr>
<tr>
<td>Look up a keyboard shortcut</td>
<td>IDE Concepts</td>
</tr>
<tr>
<td>Find the meaning of a special term used in the IDE</td>
<td>Glossary</td>
</tr>
</tbody>
</table>

---

---

---
Chapter 1
IDE Concepts

In this chapter...

What is an IDE? 3
Starting the IDE 4
Workbench 10
Help system 11
Perspectives 13
Views and editors 14
Projects and workspace 16
Host and target machines 18
Target agent (the qconn daemon) 18
Launcher 19
Resources 19
Wizards 19
Keyboard shortcuts 19
Preferences 20
Version coexistence 21
What’s new in the IDE? 24
What is an IDE?

Welcome to the Integrated Development Environment (IDE), a powerful set of tools in the QNX Momentics Professional Edition development suite. The IDE is based on the Eclipse Platform developed by Eclipse.org, an open consortium of tools vendors (including QNX Software Systems).

The IDE incorporates into the Eclipse framework several QNX-specific plugins designed for building projects for target systems running the QNX Neutrino RTOS. The tools suite provides a single, consistent, integrated environment, regardless of the host platform you’re using (Windows, Linux, Solaris, or QNX Neutrino). Note that all plugins from any vendor work within the Eclipse framework in the same way.

An IDE for building embedded systems

The IDE provides a coherent, easy-to-use work environment for building your applications. If you’ve used an IDE before, then you already have a good idea of the convenience and power this kind of toolset can offer.
Starting the IDE

Through a set of related windows, the IDE presents various ways of viewing and working with all the pieces that comprise your system. In terms of the tasks you can perform, the toolset lets you:

- organize your resources (projects, folders, files)
- edit resources
- collaborate on projects with a team
- compile, run, and debug your programs
- build OS and Flash images for your embedded systems
- analyze and fine-tune your system’s performance

The IDE doesn’t force you to abandon the standard QNX tools and Makefile structure. On the contrary, it relies on those tools. And even if you continue to build your programs at the command line, you can also benefit from the IDE’s unique and powerful tools, such as the QNX System Analysis tool and the QNX System Profiler, which can literally show you, in dynamic, graphical ways, exactly what your system is up to.

Starting the IDE

After you install QNX Momentics, you’ll see — depending on which host you’re using — a desktop icon and/or a menu item labeled “Integrated Development Environment” in the start or launch menu. To start the IDE, simply click the icon or the menu item.

On Solaris, you must start the IDE from the command-line:

```
$QNXHOST/usr/qde/eclipse/qde -vmargs -Xms256m -Xmx512m
```
On Neutrino, don’t start the IDE from the command line if you’ve used the `su` command to switch to a different user. It is unable to attach to your Photon session and fails to start.

**Starting the IDE for the first time**

The first time you start the IDE on Windows, the Workspace Launcher dialog asks you where to store your workspace. All of your IDE projects are stored in this directory.

![Workspace Launcher dialog](image)

**Selecting a workspace directory.**

By default, the IDE offers to put your workspace in your home directory (`$HOME/workspace` on Neutrino, Linux, and Solaris), or the path specified in the QNX Momentics IDE shortcut (C:\QNX630\workspace) on Windows. To store your workspace in another location:

- Click the **Browse**... button and select a directory for your workspace.

To continue loading the IDE, click the **OK** button.
Starting the IDE

Check the **Use this as the default and do not ask again** box to always use this workspace when launching the IDE.

To change the default workspace location on QNX, Linux, and Solaris, launch `qde` with the `-data workspace_path` option.

**The IDE welcomes you**

After you choose a workspace location, the IDE displays a welcome screen with several options that help to introduce you to the IDE:

![The IDE's welcome screen.](image)

The icons on the welcome screen are:
Starting the IDE

**Icon** | **Description**
--- | ---
| Takes you to the workbench screen and your workspace. |
| Provides links to overviews of the IDE: the Documentation Roadmap, Team Support (an important topic if you use CVS), Workbench Basics, and C/C++ Online Docs. |
| Provides links to the *Quickstart Guide — 10 Steps to Your First QNX Program*, and the C/C++ tutorials included with the C Development Toolkit (CDT). |
| Provides links to documents describing new features: the new features in this release and information about migrating from a previous release. |

You can return to this welcome screen at any time by choosing Help→Welcome.
Starting the IDE

Starting the IDE after an update

After you’ve updated one or more IDE components, such as the CDT or an Eclipse add-on, you might be prompted to process these updates the next time you launch the IDE:

![Configuration Changes dialog](image)

This doesn’t apply to the QNX Momentics 6.3.0 Service Pack 2 update, although it does apply to the 6.3.0 Service Pack 1 update.

To process IDE updates:

8 Chapter 1 • IDE Concepts
Starting the IDE

1. In the Configuration Changes dialog, click the + symbol to expand the list of updates.
2. If you see an update you don’t want to apply to the IDE, clear its check box.
3. Click the Finish button.
The IDE processes its updates and then displays the Install/Update dialog.

The Install/Update dialog tells you to restart the IDE.

4. Click Yes to restart the IDE with all of the processed updates. Click No to continue using the IDE.

Starting the IDE from the command line

You can also start the IDE by running the qde command:

1. Go to the directory where the qde.exe executable (Windows) or the qde script (all other hosts) resides. For example, C:/QNX630/host/win32/x86/usr/qde/eclipse.
2. Run this command:
   ./qde
Don’t run the `eclipse` command, even though it may seem to work. Always use `qde` instead, because it sets up the proper QNX-specific environment.

You can also direct the IDE at a particular workspace location. For details, see the section “Specifying a workspace location” in this chapter.

For more information on starting the IDE, including advanced execution options for developing or debugging parts of Eclipse itself, see Tasks→Running Eclipse in the Workbench User Guide.

Workbench

When you first run the IDE, you should see the workbench, which looks like this:

![The first thing you see.](image)
Help system

For details about the workbench’s menu, see Reference→User Interface Information→Workbench menus in the Workbench User Guide. For a basic tutorial on using the workbench UI, see Getting Started→Basic tutorial→The Workbench in the Workbench User Guide.

Help system

The IDE contains its own help system, which is an HTML server that runs in its own window “above” the workbench (i.e. the help isn’t a perspective or a view).

On Linux, the IDE tries to start the Mozilla web browser to display the online help system. Red Hat Enterprise Linux 4 now ships with Firefox instead of Mozilla, so you’ll have to change the help browser setting:

1. Open the Preferences dialog (Window→Preferences...).
2. In the left-hand panel, select the Help item.
3. Change the Custom Browser command to firefox %1.
4. Click OK to close the Preferences dialog and save your changes.

Opening the IDE Help

➤ From the main menu, select Help→Help Contents.

Navigating the Help

The left pane of the Help window is the bookshelf, which has links to the various documentation sets. Click one of the links to view a document. Note that you can return to the bookshelf at any time by clicking the Table of Contents button ( ).

The Contents pane includes at least the following titles:
Workbench User Guide

Written by Eclipse.org, the book explains Eclipse concepts and core IDE functionality, and includes tutorials on using the workbench. Although some of the workbench topics are covered lightly here in this IDE User’s Guide, you can find full documentation on using the workbench in the Eclipse Workbench User Guide.

QNX Momentics Professional Edition

The QNX documentation set, which includes several titles:

- A Roadmap to the QNX Momentics Professional Edition
- Dinkum library documentation
- High Availability Toolkit
- Getting Started with Neutrino 2
- Integrated Development Environment (featuring this User’s Guide)
- Phindows for QNX Neutrino
- Photon Multilingual Input
- Photon microGUI for QNX Neutrino
- Power Management
- System Analysis Toolkit
- Documentation for DDKs, and much, much more

Some title pages have content on them, some don’t. If you click a title, and the right side of the window remains blank, you’ve hit a “placeholder” title page. Simply expand the title entry to see its contents.
Help bookmarks

You can create a bookmark for any help page:

1. On the Help browser’s toolbar, click the Bookmark Document button ( ).

2. To see your bookmarks, click the Bookmarks ( ) tab at the bottom of the contents pane.

To learn more about the IDE’s Help system, follow these links in the Eclipse Workbench User Guide: Concepts → Help system.

Tips and tricks

When you select the Tips and tricks item from the Help menu, you’ll see a list of tips and tricks pages. Select the page for the Eclipse platform, which covers several topics:

- workbench (fast views, opening an editor with drag-and-drop, navigation, global find/replace, etc.)
- help (help bookmarks, help working sets)
- CVS (CVS working sets, restoring deleted files, quick sync, etc.)

Perspectives

A perspective is a task-oriented arrangement of the workbench window.

For example, if you’re debugging, you can use the preconfigured Debug perspective, which sets up the IDE to show all the tools related to debugging. If you wanted to work with the elements and tools related to profiling, you’d open the QNX Application Profiler perspective.

You can customize a perspective by adding or removing elements. For example, if you wanted to have certain profiling tools available whenever you’re debugging, you could add those elements to the Debug perspective.
Perspectives generally consist of these components:

- toolbars
- views
- editors

Perspectives govern which views appear on your workbench. For example, when you’re in the Debug perspective, the following main views are available (in the default configuration):

- Debug
- Breakpoints
- Variables
- Console
- Outline
- Tasks

## Views and editors

### Views

*Views* organize information in various convenient ways. For example, the Outline view shows you a list of all the function names when you’re editing a C file in the C/C++ editor. The Outline view is dynamic; if you declare a function called `mynewfunc()`, the Outline view immediately lists it.

Views give you different presentations of your resources. For example, the Navigator view shows the resources (projects, folders, files) you’re working on. Like individual panes in a large window, views let you see different aspects of your entire set of resources.

Views provide:

- insight into editor contents (e.g. Outline view)
• navigation (e.g. Navigator view)
• information (e.g. Tasks view)
• control (e.g. Debug view)

Editors

You use editors to browse or change the content of your files. Each editor in the IDE is designed for working with a specific type of file. The editor that you’ll likely use the most is the C/C++ editor.

The editor area is a section of the workbench window reserved for editors. Note that views can be anywhere on the workbench except in the editor area.

The IDE lets you rearrange views and editors so they’re beside each other (tiled) or stacked on top of each other (tabbed).

If you wish to use a different text editor than the one that’s built into the IDE, you can do so, but you’ll lose the integration of the various views and perspectives. For example, within the IDE’s text editor, you can set breakpoints and then see them in the Breakpoints view, or put “to-do” markers on particular lines and see them in the Tasks view, or get context-sensitive help as you pause your cursor over a function name in your code, and much, much more.

But if you want to use your own editor, we recommend that you:

1. Edit your files outside of the IDE.
2. Make sure that you save your files in the right place, e.g. on Windows:
   
   ```plaintext
   C:/QNX630/workspace/project_name
   ```

3. From within the IDE, use the Refresh command (right-click menu in the Navigator view or the C/C++ Projects view).
Projects and workspace

Projects and workspace

Projects are generic containers for your source code, Makefiles, and binaries. Before you do any work in the IDE, you must first create projects to store your work. One of the more common projects is a QNX C/C++ Project.

Throughout this guide, we use the term “C/C++” as shorthand to cover both C and C++ projects. The titles of elements within the IDE itself are often explicit (e.g. “QNX C Project,” “QNX C++ Project,” etc.).

When you create a file within a project, the IDE also creates a record (local history) of every time you changed that file and how you changed it.

Your workspace is a folder where you keep your projects. For the exact location of your workspace folder on your particular host, see the appendix Where Files Are Stored in this guide.

Specifying a workspace location

You can redirect the IDE to point at different workspaces:

- From the directory where the qde.exe executable (Windows) or the qde script (all other hosts) resides, run this command:
  ```
  ./qde -data path_to_workspace
  ```

This command launches the IDE and specifies where you want the IDE to create (or look for) the workspace folder.
Don’t use spaces when naming a project or file — they can cause problems with some tools, such as the `make` utility.

Also, don’t use case alone to distinguish files and projects. On Unix-style hosts (i.e. Solaris, Linux, QNX Neutrino), filenames are case-sensitive, but in Windows they’re not. For example, `Hello.c` and `hello.c` would refer to the same file in Windows, but would be separate filenames in Unix-style systems.

How the IDE looks at projects

The IDE associates projects with *natures* that define the characteristics of a given project. For example, a Standard Make C Project has a “C nature,” whereas a QNX C Project has a C nature as well as a QNX C nature, and so on. Note that QNX C or C++ projects assume the QNX recursive *Makefile* hierarchy to support multiple target architectures; Standard Make C/C++ projects don’t.

For more on the QNX recursive *Makefile* hierarchy, see the Conventions for Makefiles and Directories appendix in the *Neutrino Programmer’s Guide*.

The natures tell the IDE what can and can’t be done with each project. The IDE also uses the natures to filter out projects that would be irrelevant in certain contexts (e.g. a list of QNX System Builder projects won’t contain any C++ library projects).

Here are the most common projects and their associated natures:

<table>
<thead>
<tr>
<th>Project</th>
<th>Associated natures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Project</td>
<td>n/a</td>
</tr>
<tr>
<td>Standard Make C Project</td>
<td>C</td>
</tr>
<tr>
<td>Standard Make C++ Project</td>
<td>C, C++</td>
</tr>
</tbody>
</table>

*continued…*
The IDE saves these natures and other information in `.project` and `.cdtproject` files in each project. To ensure that these natures persist in CVS, include these files when you commit your project.

The IDE doesn’t directly support nested projects; each project must be organized as a discrete entity. However, the IDE does support project dependencies by allowing a project to reference other projects that reside in your workspace. Container projects also let you logically nest projects by collecting several projects together.

### Host and target machines

The *host* is the machine where the IDE resides (e.g. Windows). The *target* is the machine where QNX Neutrino and your program actually run.

### Target agent (the `qconn` daemon)

The `qconn` daemon is the target agent written specifically to support the IDE. It facilitates communication between the host and target machines.

If you’re running the IDE on a QNX Neutrino PC (self-hosted), your target machine may also be the host machine. In this case, you must still run `qconn`, even though your host machine is “connecting to itself.”
For more information about connection methods, see the Launch Configurations Reference chapter in this guide.

**Launcher**

Before you can run a program, you must tell the IDE’s launcher what program to run, what target to run it on, what arguments to pass to the program, and so on.

If you want to run the program on another target or run with different options (e.g. with profiling enabled), you must create a new *launch configuration* or copy a previous one and modify it.

**Resources**

*Resources* is a collective term for your projects, folders, and files.

**Wizards**

*Wizards* guide you through a sequence of tasks. For example, to create a QNX C Project, you run a wizard that takes you through all the steps and gathers all the necessary information before creating the project. For more information, see the Common Wizards Reference chapter in this guide.

**Keyboard shortcuts**

You’ll find many keyboard shortcuts for various UI tasks throughout the IDE. You can easily create your own shortcuts. For instructions, follow these links in the *Workbench User Guide*:

**Reference**→**Preferences**→**Keys**
Preferences

Some existing shortcuts and some commands that can be assigned to shortcuts apply only to Java code and projects. For example, the “Search for Declaration in Workspace” command, which is bound to Ctrl – G works only with Java code.

Preferences

The Preferences dialog (under the Window menu) lets you customize the behavior of your environment — when to build your projects, how to open new perspectives, which target processors to build for, etc.

Besides global preferences, you can also set preferences on a per-project basis via the Properties item in right-click menus.
Version coexistence

The QNX Momentics 6.3.0 development suite lets you install and work with multiple versions of Neutrino (from 6.2.1 and later) — you can choose which version of the OS to build programs for.

When you install QNX Momentics, you get a set of configuration files that indicate where you’ve installed the software. The QNX_CONFIGURATION environment variable stores the location of the configuration files for the installed versions of Neutrino; on a self-hosted Neutrino machine, the default is /etc/qnx.

QWinCfg for Windows hosts

On Windows hosts, you’ll find a configuration program (QWinCfg) for switching between versions of QNX Momentics.

You launch QWinCfg via the start menu (e.g. All Programs→QNX Momentics 6.3.0→Configuration).

For details on using QWinCfg, see its entry in the Utilities Reference.

qconfig utility for non-Windows hosts

The qconfig utility lets you configure your machine to use a specific version of Neutrino:

- If you run it without any options, qconfig lists the versions that are installed on your machine.

- If you specify the -e option, you can set up the environment for building software for a specific version of the OS. For example, if you’re using the Korn shell (ksh), you can configure your machine like this:

```
eval `qconfig -n "QNX 6.3.0 Install" -e`
```
Version coexistence

In the above command, you must use the “back tick” character (‘), not the single quote character (’).

When you start the IDE, it uses your current qconfig choice as the default version of the OS; if you haven’t chosen a version, the IDE chooses an entry from the directory identified by QNX_CONFIGURATION. If you want to override the IDE’s choice, you can choose the appropriate build target.

Coexistence and PhAB

If you’re going to create Photon applications for QNX 6.3.0 and 6.2.1 using PhAB, you need to use the older version of PhAB to create your application resources.

To ensure that you’re always using the older version of PhAB to create your resources:

1  Choose Window→Preferences from the menu to display the Preferences dialog.

2  Expand the QNX item in the list, then choose Appbuilder to display the Appbuilder preferences:
Version coexistence

3 Un-check the **Use default** check box.

4 Change the **Path to Photon Appbuilder** to
   \texttt{C:/QNXsdk/host/win32/x86/usr/bin/appbuilder.bat}.

5 Click **OK** to save your changes and close the Preferences dialog.

**Specifying which OS version to build for**

To specify which version of Neutrino you want the IDE to build for:

1 Open the Preferences dialog (**Window\rightarrow Preferences**).

2 Select **QNX**.

3 Using the dropdown list in the **Select Install** field, choose the
   OS version you want to build for.

4 Click **Apply**, then **OK**.
What's new in the IDE?

Environment variables

Neutrino uses these environment variables to locate files on the host machine:

- **QNX_HOST**  
  The location of host-specific files.

- **QNX_TARGET**  
  The location of target backends on the host machine.

- **QNX_CONFIGURATION**  
  The location of the qconfig configuration files.

- **MAKEFLAGS**  
  The location of included *.mk files.

- **TMPDIR**  
  The directory to use for temporary files. The gcc compiler uses temporary files to hold the output of one stage of compilation used as input to the next stage: for example, the output of the preprocessor, which is the input to the compiler proper.

The qconfig utility sets these variables according to the version of QNX Momentics that you specified.

What’s new in the IDE?

Each update to the Momentics IDE adds new abilities and features.

What’s new in 6.3.0 SP1?

Here are some of the more interesting or significant changes made to the QNX Momentics IDE since the release of QNX Momentics 6.3.0:

- Improved documentation, including more extensive code importing procedures, etc.

- The System Builder perspective now supports projects with more than one build file, and the perspective’s icons have been improved.
What's new in the IDE?

- The Application Profiler, System Builder, System Information, and System Profiler perspectives have been improved.
- The stability and usability of the self-hosted IDE have been improved.
- Support for Intel's C compiler (icc) has been added.
- The Code Coverage perspective now works with gcc 3.3.1 or later.

What's new in 6.3.0 SP2?

The following sections describe some of the more interesting or significant changes made to the QNX Momentics IDE since the release of QNX Momentics 6.3.0 SP1:

- General IDE
- C/C++ user interface
- C/C++ debug and launch
- C/C++ project configuration and build
- C/C++ editing and source navigation
- QNX Momentics tools

General IDE

The QNX Momentics 6.3.0 SP2 IDE sports many useful new features:

- New look and feel
- Responsive UI
- Editor management enhancements
- Themes
- Background workspace auto-refresh
- Regular expressions in Find/Replace dialog
What's new in the IDE?

- New text editor functions
- New editor functions
- Opening external files

New look and feel

The look and feel of the workbench has evolved. Here are some of the things you might notice:

- Title bars and tabs for views and editors look different.
- Title bars and tabs for views and editors let you maximize and restore.
- Views include a button for collapsing (minimizing).
- Perspective switching/opening toolbar support has changed:
  - it can be docked on the top right (default), top left or left, and
  - perspective buttons include text for quickly identifying the current perspective.
- The Fast View bar can be on the bottom (default), left or right.
- Fast View bar size is reduced when there are no fast views.
- Title bars and tabs have been merged to save space.
- Drag-and-drop has been improved (better feedback while dragging).
- Detached views are supported (Windows and Linux GTK only, due to platform limitations).
- Editor management has changed.
- The view-specific toolbars are now next to the view’s tab to save space when possible.
- Different tab folder styles and uses of color have been employed to help indicate active and selected views and editors more clearly.
What's new in the IDE?

- Other minor items such as status bar style, border widths, shading, etc.

Responsive UI

A number of changes have occurred in the UI to support a higher level of responsiveness. This includes support for running jobs in the background instead of tying up the UI and making you wait.

The IDE now features a:

- Progress view
- status line entry showing what’s running in the background
- dialog for showing operations that can be run in the background
What's new in the IDE?

The new Progress view showing the progress of a CVS checkout and a Workspace build background operation.

Many user operations can now be run in the background. When you see the progress dialog with the Run In Background button you can select it to get back to work.
What's new in the IDE?

This dialog also shows you the details of other currently running operations in the workspace and informs you when one operation is blocked waiting for another to complete.

Editor management enhancements

A number of changes and enhancements have gone into the editor management in the QNX Momentics IDE.

The IDE now provides:

- support for single and multiple editor tabs; single is especially useful for those who tend to have many files open or who like using the keyboard to navigate editors

- support for editor pinning. When limiting the number of editors that can be opened at once, some editors that should not be closed can be pinned. An indicator is displayed when an editor is pinned.

- chevrons to handle the overflow of editors with an extra indication of how many more editors are open than there are tabs on the screen.

- new menu options, keyboard shortcuts and key bindings for editor management:
  - Close Others — close all editors but the current.
  - Close All — menu option available.
  - Ctrl+E — dropdown list of editors supports type ahead.
What's new in the IDE?

Themes

The QNX Momentics IDE now contains basic support for themes. This currently goes as far as allowing customization of colors and fonts used in the workbench.

Background workspace auto-refresh

Changes made in the local file system can now be automatically refreshed in the workspace. This saves you from having to do a manual File→Refresh every time you modify files with an external editor or tool. This feature is currently disabled by default, but can be turned on from the Workbench preference page.

Regular expressions in Find/Replace dialog

The Find/Replace dialog for text editors now supports searching and replacing using regular expressions. Press F1 to get an overview of
the regular expression syntax, and press Ctrl – Space to get Content Assist for inserting regular expression constructs.

When the cursor is placed in a dialog field that can provide Content Assist, a small lightbulb appears at the upper-left corner of the field.

![Find/Replace dialog box]

**New text editor functions**

You can now customize the displayed width of tabs and the text selection foreground and background colors in the text editor. See the Workbench→Editors→Text Editor page:
What's new in the IDE?

New editor functions

All text editors based on the QNX Momentics IDE editor framework support new editing functions, including moving lines up or down (Alt – Up Arrow and Alt – Down Arrow), copying lines (Ctrl – Alt – Up Arrow and Ctrl – Alt – Down Arrow), inserting new a line above or below the current line (Ctrl – Shift – Enter and Shift – Enter), and converting to lowercase or uppercase (Ctrl – Shift – Y and Ctrl – Shift – X).

Double clicking on the line number in the status line is the same as Navigate→Go to Line… (Ctrl – L).

Opening external files

The File menu now includes an Open External File… option that lets you open any file in the workbench without having to import it into a project.
What’s new in the IDE?

C/C++ user interface

The new CDT in the QNX Momentics 6.3.0 SP2 IDE features:

- Outline filters and groups
- New wizard for creating C++ classes
- New wizards for working with C/C++
- Code folding
- Makefile editor

Outline filters and groups

The Outline view now offers users the ability to filter out certain elements such as defines and namespaces as well as the ability to group all include statements together.

New C++ Class wizard

Creating new C++ classes continues to get easier with a number of enhancements to the C++ class-creation wizard.
What's new in the IDE?

New C/C++ wizards

A new toolbar has been created that facilitates the creation of a number of standard C/C++ objects:

- source and header files
- source folders
- C and C++ projects
What’s new in the IDE?

Code folding

The C/C++ editor supports code folding for functions, methods, classes, structures and macros.

Makefile editor

The Makefile editor has a whole new set of preferences and now supports code folding.
What's new in the IDE?

C/C++ debug and launch

Debugging support and application launching in the CDT has been improved, as described in the following sections:

- Thread-specific breakpoints
- Breakpoint filtering
What's new in the IDE?

- Workspace variable support
- Mixed source/assembly
- Global variables
- Debug console
- Automatic refresh options
- Finer-grained Launch configurations

**Thread-specific breakpoints**

The C/C++ Debugger now supports thread-specific breakpoints. After placing a breakpoint, look at its **Properties** to see which threads or processes it is active for.

![Properties for C/C++ breakpoint](image)

**Breakpoint filtering**

The Breakpoints view now lets you filter out all of the irrelevant breakpoints based on the specific process that you’re debugging.

![Breakpoints view](image)

**Workspace variable support**

C/C++ launch configurations now include support for workspace variables in the **Environment**, **Argument**, and **Working Directory** tabs.
What's new in the IDE?

Mixed source/assembly

Gone are the days of toggling the C/C++ editor to show the assembly of a program. Instead, use the Disassembly view to see both assembly code and source mixed:

```
int main(int argc, char **argv) {
  0x00400000 <main>: push %ebp
  0x00400000 <main+1>: mov %esp,%ebp
  0x00400000 <main+4>: sub $0x18,%esp
  int x = 0;
  0x00400000 <main+6>: movl $0x0,0xffffffff(%ebp)
    struct mystruct s;
    x++;
  0x00400000 <main+10>: incl 0xffffffff(%ebp)
    // This will optimize out
    s.next = s.next;
    printf("Hello World \d", x);
  0x00400000 <main+16>: add $0xffffffff,$esp
  0x00400000 <main+18>: mov 0xffffffff(%ebp),%eax
```

Global variables

You can now add global variables can now be added to the Variables view instead of having to add them as separate expressions.
What's new in the IDE?

**Debug console**

The Debug Console has moved to being a proper console selection of its own in the generic Console view.

**Automatic refresh options**

You can now configure the default behavior for the automatic retrieval of shared library and register information in the C/C++ debugger.

You can specify whether to refresh register values automatically or manually from the Launch configuration dialog with the Advanced button of the Debug tab.
What's new in the IDE?

Finer-grained Launch configurations

You can now maintain separate Run and Debug launch configurations for debugging core files, attaching to a running process, attaching to your target with `pdebug` (serial debugging), and attaching to your target with `qconn` (TCP/IP debugging).
What's new in the IDE?

C/C++ project configuration and build

Project configuration and building has been improved:

- Automatic project settings discovery
- Include paths and symbols
- Source folders
- C/C++ file types
- C/C++ working set

Automatic project settings discovery

Automatically generate project defines and include path settings from the C/C++ Standard Make project’s Discovery Options project settings.

Note that this is for projects being built with one of the platform-specific nto*-gcc drivers and a custom Makefile.
What's new in the IDE?

Include paths & symbols

Use the C/C++ Include Paths and Symbols to set up the project settings appropriately for searching, indexing and other source navigation functionality.

Source folders

Use the C/C++ Project Paths project properties to determine those files and directories that should be specifically considered as containing source, output or library content. Performance can be improved by limiting the directories and files of large projects.
C/C++ file types

Define the types of specific files, especially C++ headers without extensions, using the C/C++ File Types global preference or project property.
What's new in the IDE?

C/C++ working set

You can now create working sets containing only C/C++ projects and resources by creating a C/C++ Working Set definition.
What’s new in the IDE?

C/C++ editing and source navigation

Editing and navigating your C/C++ source files is now easier with:

- C/C++ content assist
- Rename refactoring
- Open type
- C/C++ Browsing perspective
- Makefile editor
- Search enhancements
- Hyperlink navigation
- C/C++ Browsing perspective

C/C++ content assist

Editing code just got easier with a more fully featured content assist. Completions are now provided in the C/C++ editor for:

- classes and structure members
- local and global variables
- functions
- preprocessor defines
- preprocessor commands
What's new in the IDE?

Configure completion options in the global C/C++ Editor Preferences.
What's new in the IDE?

Rename refactoring

Use the Outline view or the C/C++ editor’s Refactor→Rename context menu to refactor class and type names, methods, functions and member names.

Open type

Use Navigate→Open type (Ctrl – Shift – T) to open up the declaration of C/C++ classes, structures, unions, typedefs, enumerations and namespaces.
What's new in the IDE?

C/C++ Browsing perspective

Use the C/C++ Browsing perspective to navigate the class and structure members of a particular project.
What's new in the IDE?

Makefile editor

The Makefile editor now provides syntax highlighting, code completion, and content outlining capabilities.
What's new in the IDE?

**Search enhancements**

The C/C++ Search dialog provides context sensitive searches from the Outline view as well as resource selection-restricted searches in the **C/C++ Search** dialog.

**Hyperlink navigation**

The C/C++ Editor supports hyperlink navigation if enabled via **Window → Preferences → C/C++ → C/C++ Editor Preferences**. Then you can use Ctrl – Click to jump to the declaration of an item directly in the C/C++ editor.

**Index error markers**

Enable C/C++ indexing and indexer error reporting in the **C/C++ Indexer** properties. This helps identify projects missing path-configuration information.

Configure the indexer from the **C/C++ Indexer** project settings:
What's new in the IDE?

QNX Momentics tools

These exclusive QNX Momentics tools have also been updated and improved:

- Memory Analysis
- System Profiler
- Code Coverage
- System Information
- System Builder

Memory Analysis

The following new features have been added to the Memory Analysis perspective:

- streamlined user interface
- support for memory leak detection in real time and when a program exits
- deeper, configurable, backtrace information, configured separately for allocation tracing and error detection
What's new in the IDE?

- timestamp tracking of allocations and errors
- thread tracking of allocations and errors
- an optional command-line interface
- the ability to dump trace information to a file
- external runtime control (start, stop for tracing and leak detection)

The Memory Information and Malloc Information views are now part of the System Information perspective.
What's new in the IDE?

System Profiler

New features added to the System Profiler in 6.3.0 SP1:

- improved scalability and performance
- improved graphic timeline display for events.

- additional filters: state activity, IPC activity, CPU usage

New features added to the System Profiler in 6.3.0 SP2:

- new interrupt handler element, with its own timeline, CPU usage, and calling process
- CPU usage and process activity for threads takes into account the time spent in Interrupts.
- Several new graph types including 3D perspectives:
What's new in the IDE?

**Code Coverage**

Improved reporting output and export capabilities.

![Code Coverage Report]

**System Information**

The System Information perspective has been rewritten with a new update control mechanism, and simplified Process Information and Signal Information views.

The new Process Information view:
What's new in the IDE?

The new Signal Information view:

The Memory Information and Malloc Information views (formerly found in the Memory Analysis perspective) are now part of the System Information perspective.
What's new in the IDE?
What's new in the IDE?

System Builder

The following new features have been added to the System Builder perspective:

- You can now create projects with multiple IFS images. Each IFS image can be combined with one or more EFS images while building the project.
- There are now more ways to add a filesystem image to the existing project.
- You can build each IFS or EFS component separately.
- Image combining can be done as a separate step. The images to combine can be defined at that point, and you can dynamically change the combination parameters for each component.
- System Builder now displays the filesystem layout for each IFS or EFS image.
What's new in the IDE?

- The System Optimization component is more flexible.
Chapter 2
Preparing Your Target

In this chapter...

Host-target communications 61
Connecting with Phindows 67
This chapter explains how to set up host-target communications.

Host-target communications

Regardless of whether you’re connecting to a remote or a local target, you have to prepare your target machine so that the IDE can interact with the QNX Neutrino image running on the target.

The IDE supports host-target communications using either an IP or a serial connection. We recommend both. If you have only a serial link, you’ll be able to debug a program, but you’ll need an IP link in order to use any of the advanced diagnostic tools in the IDE.

Target systems need to run the target agent (qconn). See “Target agent (the qconn daemon)” in the IDE Concepts chapter for more information.

IP communications

Before you can configure your target for IP communications, you must connect the target and host machines to the same network. You must already have TCP/IP networking functioning between the host and target systems.
Host-target communications

To configure your target for IP communications, you must launch `qconn` on the target, either from a command-line shell or the target’s boot script.

The version of QNX Momentics on your host must be the same or newer than the version of QNX Neutrino on your target, or unexpected behavior may occur. Newer features won’t be supported by an older target.

When you set up a launch configuration, select **C/C++ QNX QConn (IP)**. (See the Launch Configurations Reference chapter in this guide for more information.)

The `pdebug` command must be present on the target system in `/usr/bin` for all debugging sessions. `qconn` launches it as needed. The `devc-pty` manager must also be running on the target to support the Debug perspective’s Terminal view.

Serial communications

Before you can configure your target for serial communications, you must establish a working serial connection between your host and target machines.

On Linux, disable and stop `mgetty` before configuring your target for serial communications.

To configure your target for serial communications:

1. If it’s not already running, start the serial device driver that’s appropriate for your target. Intel x86-based machines usually use the `devc-ser8250` driver.

2. Once the serial driver is running, you’ll see a serial device listed in the `/dev` directory. To confirm it’s running, enter:
   ```
   ls /dev/ser*
   ```
   You’ll see an entry such as `/dev/ser1` or `/dev/ser2`.
3 Start the pseudo-terminal communications manager (devc-pty):
   devc-pty &

4 Start the debug agent by entering this command (assuming you’re using the first serial port on your target):
   pdebug /dev/ser1 &

The target is now fully configured.

5 Determine the serial port parameters by entering this command (again assuming the first serial port):
   stty </dev/ser1

This command gives a lot of output. Look for the baud=baudrate entry; you’ll need this information to properly configure the host side of the connection.

When you set up a launch configuration, select C/C++ QNX PDebug (Serial). (See the Launch Configurations Reference chapter in this guide for more information.)

Example: Debugging via PPP

This example shows you how to prepare your target and host for debugging using a PPP connection.

Before you begin, make sure the serial ports on both the host and target systems are configured properly and can talk to each other through a null-modem serial cable.

Setting up your target

To configure your target for PPP:

1 Create a /etc/ppp/options file containing the following:
   debug
   57600
   /dev/ser1
   10.0.0.1:10.0.0.0
Host-target communications

You may need to try a different baud rate if you have any problems at 57600.

2 If it’s not already running, start io-net with this command:
   io-net -ptcpip -pppmgr

3 Now start the PPP daemon:
   pppd

4 Finally, start the qconn target agent:
   qconn

QNX Neutrino host

To configure your QNX Neutrino host for PPP:

1 Create a /etc/ppp/options file containing the following:
   debug
   57600
   /dev/ser1
   10.0.0.1:10.0.0.0

You may need to try a different baud rate if you have any problems at 57600.

2 If it’s not already running, start io-net with this command:
   io-net -ptcpip -pppmgr

3 Start the PPP daemon with the passive option:
   pppd passive
Windows host

To configure your Windows XP host for serial communication using PPP:

The names of menu items and other details differ slightly on other supported versions of Windows.

1. In the Control Panel window, select Network Connections.

2. In the New Connection Wizard dialog, click Set up an advanced connection, then click Next:

3. Select Connect directly to another computer, then click Next.

4. When prompted for the role of your target, choose Guest:
5 Name your connection (e.g. “ppp_biscayne”).

6 When prompted to select a device, choose **Communications Port (COM1)**, then click **Next**.

7 When prompted to specify whether you want this connection to be for your use only, or for anyone’s, select **Anyone’s use**.

8 If you want Windows to create a desktop shortcut, click the option on the last page of the wizard. If not, simply click **Finish**.

9 In the **Connect name of target** dialog, enter your user ID and password, the select **Properties**.

10 Select the **Options** tab.

11 Turn off the option **Prompt for name and password, certificate, etc.**, then click **OK**.
Connecting with Phindows

The IDE lets you connect to a Photon session on a target from a Windows host machine and interact with the remote Photon system as if you were sitting in front of the target machine.

To prepare your target for a Phindows connection:

1. Open a terminal window and log in as root.
2. Edit the /etc/inetd.conf file and add the following line (or uncomment it if it’s already there):
   
   ```
   phrelay stream tcp nowait root /usr/bin/phrelay phrelay -x
   ```

3. Save the file and exit the editor.
4. If it’s running, kill the inetd daemon:
   
   ```
   slay inetd
   ```

5. Now restart inetd:
   
   ```
   inetd
   ```

   The inetd daemon starts and you can connect to your target using Phindows.

For details on using Phindows, see the Phindows Connectivity User’s Guide in your QNX Momentics documentation set.
Chapter 3
Developing C/C++ Programs

In this chapter...

The C/C++ Development perspective 71
Controlling your projects 72
Creating projects 76
Building projects 78
Running projects 85
Deleting projects 89
Writing code 89
More development features 98
The C/C++ Development perspective

This chapter shows you how to create and manage your C or C++ projects.

The C/C++ Development perspective

The C/C++ Development perspective is where you develop and build your projects. As mentioned in the Concepts chapter, a project is a container for organizing and storing your files.

Besides writing code and building your projects, you may also debug and analyze your programs from the C/C++ Development perspective.

You’ll find complete documentation on the C/C++ Development perspective, including several tutorials to help you get started, in the core Eclipse platform docset: Help→Help Contents→C/C++ Development User Guide.

The views in the C/C++ Development perspective are driven primarily by selections you make in the C/C++ editor and the C/C++ Projects view, which is a specialized version of the Navigator view.

Since the Navigator view is part of the core Eclipse platform, you’ll find full documentation on the Navigator view in the Workbench User Guide.
Controlling your projects

For information on the Navigator’s:

See these sections in the Workbench User Guide:

<table>
<thead>
<tr>
<th>toolbar and icons</th>
<th>Concepts → Views → Navigator View</th>
</tr>
</thead>
<tbody>
<tr>
<td>right-click context menu</td>
<td>Reference → User interface information → Views and editors → Navigator View</td>
</tr>
</tbody>
</table>

Wizards and Launch Configurations

To create and run your first program, you’ll use two major facilities within the IDE:

- wizards — for quickly creating a new project
- launch configurations — for setting up how your program should run

Once you’ve used these parts of the IDE for the first time, you’ll be able to create, build, and run your programs very quickly. For details, see the Common Wizards Reference and Launch Configurations Reference chapters in this guide.

Controlling your projects

The C/C++ Development perspective’s C/C++ Projects view is perhaps the most important view in the IDE because you can control your projects with it. The selections you make in the C/C++ Projects view greatly affect what information the other views display.

The C/C++ Projects view gives a “virtual” or filtered presentation of all the executables, source, and shared objects that comprise your project. You can set filters for the types of files you want shown in this view.

The C/C++ Projects view has many of the same features as the Navigator view, but is configured specifically for C and C++
Controlling your projects

development. At first glance, the two may seem identical, but the C/C++ Projects view:

- shows only open C/C++ projects
- presents the project’s executables as if they reside in a subdirectory called `bin`
- for a library project, presents the project’s libraries as if they reside in subdirectory called `lib`
- hides certain files
- includes Build Project and related commands in its right-click menu
- gives an outline of `*.c`, `*.cc`, `*.cpp`, `*.h`, and binary files

Opening files

To open files and display them in the editor area:

➤ In the C/C++ Projects view, double-click the file to be opened.

The file opens in the editor area.

Opening projects

Since the C/C++ Projects view hides closed projects, you must use the Navigator view to open them.

➤ In the Navigator view, right-click your project, then select Open Project.

The project opens — you can see it in the C/C++ Projects view.
Filtering files

To hide certain files from the C/C++ Projects view:

1. In the C/C++ Projects view, click the menu dropdown button (▼).

2. Select Filters... The C Element Filters dialog appears:

   ![C Element Filters dialog]

3. In the filter pane, select the types of files you wish to hide. For example, if you select .*, then all files that start with a period...
Controlling your projects

(e.g. .cdtproject, .project, etc.) won’t appear in the C/C++ Projects view.

4 Click OK. The C/C++ Projects view automatically refreshes and shows only the files you haven’t filtered.

Outlines of source and executable files

The C/C++ Projects view shows you an outline of the .c, .cc, and .h files in your project:

Note that you can also use the Outline view to see the structure of your projects. (For more on the Outline view, see the “Code synopsis” section in this chapter.)

The C/C++ Projects view shows you the outlines of executables as well. You can examine the structure of executables to see the functions that you declared and used in the file, as well as the
elements that were called indirectly, such as `malloc()`, `init()`, and `errno`:

Creating projects

If you’re creating an application from scratch, you’ll probably want to create a QNX C Project or QNX C++ Project, which relies on the QNX recursive Makefile hierarchy to support multiple CPU targets. For more on the QNX recursive Makefile hierarchy, see the Conventions for Makefiles and Directories appendix in the Programmer’s Guide.
Creating projects

If you want to import an existing project, see the section “Importing existing source code into the IDE” in the Managing Source Code chapter in this guide.

You use the New Project wizard whenever you create a new project in the IDE. Here are the steps to create a simple “hello world” type of program:

1. In the C/C++ Development perspective, click the **New C/C++ Project** button in the toolbar:
   
   ![New Project wizard](image)

   The New Project wizard appears.

2. Name your project, then select the **type**:
   
   - Application
   - Static library
   - Shared library
   - Shared + Static library
   - Static + Static shared library
   - Shared library without export

   Even though the wizard allows it, don’t use any of the following characters in your project name (they’ll cause problems later): `| ! $ ( ) & ` : ; \ ' * ? [ ]` # `="<`>`{ }`

3. Click **Next** – but don’t press **Enter**! (Pressing **Enter** at this point amounts to clicking the Finish button, which causes the IDE to create the project for all CPU variants, which you may not want.)
Building projects

4 In the Build Variants tab, check the build variant that matches your target type, such as X86 (Little Endian), PPC (Big Endian), etc.
   Also, check **Build debug version** and **Build release version**.

5 Click **Finish**. The IDE creates your project and displays the source file in the editor.

**Building projects**

Once you’ve created your project, you’ll want to **build** it. Note that the IDE uses the same **make** utility and **Makefiles** that are used on the command line.

The IDE can build projects automatically (i.e. whenever you change your source) or let you build them manually. When you do manual builds, you can also decide on the scope of the build.

You can watch a build’s progress and see output from the build command in the Console view. If a build generates any errors or warnings, you can see them in the Problems view.

**Build terminology**

The IDE uses a number of terms to describe the scope of the build:

- **Build** Build only the components affected by modified files in that particular project (i.e. **make all**).
- **Clean** Delete all the built components (i.e. `.o`, `.so`, `.exe`, and so on) without building anything (i.e. **make clean**).
- **Rebuild** Delete all the built components, then build each one from scratch. A Rebuild is really a Clean followed by a Build (i.e. **make clean; make all**).
Turning off the autobuild feature

By default, the IDE automatically rebuilds your project every time you change a file or other resource in any way (e.g. delete, copy, save, etc.). This feature is handy if you have only a few open projects and if they’re small. But for large projects, you might want to turn this feature off.

To turn off autobuilding:

1. From the main menu, select **Window → Preferences**.
2. In the left pane, select **Workbench**.
3. In the right pane, disable the **Build automatically** option.
4. Click **OK** to save and apply your preferences.
   The IDE now builds your projects only when you ask it to.

Existing C/C++ projects (not QNX C/C++ projects) have their own autobuild setting. To turn this off:

1. Right-click the C/C++ project, then choose **Properties** from the menu.
2. Select **C/C++ Make Project** in the list on the left.
3. Select the **Make Builder** tab.
4 Disable the **Build on resource save (Auto Build)** option in the **Workbench Build Behavior** section.

5 Click **OK** to close the project properties dialog and return to the workbench.

**Building everything**

The IDE lets you manually choose to rebuild all your open projects. Depending on the number of projects, the size of the projects, and the number of target platforms, this could take a significant amount of time.

To rebuild all your open projects:

➤ From the main menu, select **Project → Build All**.
Building selected projects

To rebuild a single project:

➤ In the C/C++ Projects view, right-click a project and select Rebuild Project.

Autosave before building

To have the IDE automatically save all your changed resources before you do a manual build:

1. From the main menu, select Window→Preferences.
2. In the left pane, select Workbench.
3. In the right pane, check the Save automatically before build option.
4. Click OK to save and apply your preferences.

The IDE now saves your resources before it builds your project.
Building projects

Configuring project build order

You can tell the IDE to build certain projects before others. And if a given project refers to another project, the IDE builds that project first.

Setting the build order doesn’t necessarily cause the IDE to rebuild all projects that depend on a given project. You must rebuild all projects to ensure that all dependencies are resolved.

To manually configure the project build order:

1. From the main menu, select Window ➔ Preferences.
2. In the left pane, select Build Order.
3. Disable the Use default build order option.
4. Select a project in the list, then use the Up or Down buttons to position the project where you want in the list.
5. When you’re done, click Apply, then OK.

82  Chapter 3 • Developing C/C++ Programs
Creating personal build options

In this section, the term “targets” refers to operations that the make command executes during a build, not to target machines.

A make target is an action called by the make utility to perform a build-related task. For example, QNX Makefiles support a target named clean, which gets called as make clean. The IDE lets you set up your own make targets (e.g. myMakeStuff). You can also use a make target to pass options such as CPULIST=x86, which causes the make utility to build only for x86. Of course, such an option would work only if it’s already defined in the Makefile.

To add your own custom make target to the C/C++ Project view’s right-click menu:

1. In the C/C++ Projects view, right-click a project and select Create Make Target…
2. Type the name of your make target (e.g. myMakeStuff).
3. Click Create.

You’ll see your target option listed in the Build Targets dialog, which appears when you select the Build Make Target… item of the right-click menu of the C/C++ Projects view. Your targets also appear in the Make Targets view.

To build your project with a custom make target:

1. In the C/C++ Projects view, right-click a project.
2. In the context menu, select Build Make Target… item. The Build Targets dialog appears.
3. Select your custom target, then click Build.
To remove a `make` target:

1. Open the Make Targets view (Window → Show View → Make Targets). Expand your project to see your `make` targets.

2. Right-click the target you want to remove, then select **Delete Make Target**.

---

### Adding a `use` message

Adding a helpful “use” message to your application lets people get an instant online reminder for command-line arguments and basic usage simply by typing `use app_name`.

Usage messages are plain text files, typically named `app_name.use`, located in the root of your application’s project directory. For example, if you had the `nodetime` project open, its usage message might be in `nodetime.use`. This convention lets the recursive **Makefile** system automatically find your usage message data.

For information about writing usage messages, please refer to the **usemsg** documentation.

To add a usage message to your application when using a QNX C/C++ Project:

1. In the C/C++ Projects or Navigator view, open your project’s `common.mk` file. This file specifies common options used for building all of your active variants.

2. Find the **USEFILE** entry in `common.mk`.

3. If your usage message is in `app_name.use`, where `app_name` is your executable name, add a `#` character at the start of the **USEFILE** line. This lets the recursive **Makefile** system automatically pick up your usage message.

   If your usage message is in a file with a different name, or you want to explicitly specify your usage message’s file name, change the **USAGE** line:
Running projects

```
USAGE=$(PROJECT_ROOT)/usage_message.use
```

Where `usage_message.use` is the name of the file containing your usage message. This also assumes that your usage message file is in the root of the project directory. If the usage message file is located in another directory, include it instead of `$(PROJECT_ROOT)`.

4 Build your project as usual to include the usage message.

To add a usage message to your application when using a Standard C/C++ Project:

1 In the C/C++ Projects or Navigator view, open your project’s `Makefile`.

2 Find the rule you use to link your application’s various `.o` files into the final executable.

3 Add the following to the rule after the link command:

```
usemsg $@
usage_message.use
```

Where `usage_message.use` is the name of the file containing your usage message.

4 Build your project as usual to include the usage message.

**Running projects**

Before running an application, you must prepare your target. If it isn’t already prepared, you must do so now. See the previous chapter (Preparing Your Target) in this guide.

Once you’ve built your project, you’re ready to run it. The IDE lets you run or debug your executables on either a local or a remote QNX Neutrino target machine. (For a description of local and remote targets, see the IDE Concepts chapter.)

To run or debug your program, you must create both of the following:

- a QNX Target System Project, which specifies how the IDE communicates with your target; once you’ve created a QNX Target...
Running projects

System Project, you can reuse it for every program that runs on that particular target.

- a Launch Configuration, which describes how the program runs on your target; you’ll need to set this up only once for that particular program.

For a complete description of how to create a QNX Target System Project, see the Common Wizards Reference chapter in this guide.

For a complete description of the Launch Configurations dialog and its available options, see the Launch Configurations Reference chapter in this guide.

To create a QNX Target System Project:

1. From the menu, select File→New→Other…
2. In the list, expand QNX.
3. Select QNX Target System Project.
4. Click Next.
5 Name your target.

6 Enter your target’s Hostname or IP address.

7 Click **Finish**.

You’ll see your new QNX Target System Project in the Navigator view.

To create a launch configuration so you can run your “hello world” QNX C Application Project:
Running projects

Make sure you build your project first before you create a launch configuration for it. See “Building projects” above.

1. In the C/C++ Projects view, select your project.
2. From the Run workbench menu, click the Run... menu item.
3. In the Launch Configurations dialog, select C/C++ QNX QConn (IP) in the left pane.
4. Click New.
5. In the Name field, give your launch configuration a name.
6. Click the Search button beside the C/C++ Application field. The Program Selection dialog appears.
7. Select a program to run; the _g indicates it was compiled for debugging.
8. Click OK.
Deleting projects

To delete a project:

1. In the C/C++ Projects view, right-click a project and select Delete from the context menu. The IDE then prompts you to confirm, like this:

   ![Confirm Project Delete dialog]

2. Decide whether you want to delete just the project framework, or its contents as well.

   When you delete a project in the IDE, any launch configurations for that project are not deleted. This feature lets you delete and recreate a project without also having to repeat that operation for any corresponding launch configurations you may have created.

   For more on launch configurations, see the Launch Configurations Reference chapter in this guide.

Writing code

The C/C++ editor is where you write and modify your code. As you work in the editor, the IDE dynamically updates many of the other views (even if you haven’t saved your file).
C/C++ editor layout

The C/C++ editor has a gray border on each side. The border on the left margin might contain icons that indicate errors or other problems detected by the IDE, as well as icons for any bookmarks, breakpoints, or tasks (from the Tasks view). The icons in the left margin correspond to the line of code.

The border on the right margin displays red and yellow bars that correspond to the errors and warnings from the Problems view. Unlike the left margin, the right margin displays the icons for the entire length of the file.

The C/C++ Editor.
If you use the `cpptest.cc` sample show above, you must add the math library to your project, or you get link errors. For more information on adding libraries to your build, see the Linker tab section of the Common Wizards Reference chapter.

Finishing function names

The Content Assist feature can help you finish the names of functions if they’re long or if you can’t remember the exact spelling.

To use Content Assist:

1. In the C/C++ editor, type one or two letters of a function’s name.
2. Press Ctrl – Space. (Or, right-click near the cursor and select Content Assist.) A menu with the available functions appears:

3. You may do one of the following:
   - Continue typing. The list shortens.
   - Scroll with the up and down arrows. Press Enter to select the function.
Writing code

- Scroll with your mouse. Double-click a function to insert it.
- To close the Content Assist window, press Esc.

**Inserting code snippets**

The IDE has another code-completion feature that can insert canned snippets of code such as an empty `do-while` structure. If you’ve already used the Content Assist feature, you may have already noticed the Code Templates feature; you access it the same way.

To use Code Templates:

1. As with Content Assist, start typing, then press Ctrl – Space. (Or, right-click near the cursor and select **Content Assist**).
2. Any code templates that match the letters you’ve typed appear first in the list:

![Code Templates](image)

The IDE lets you enable as many of these templates as you like, edit them as you see fit, create your own templates, and so on.

To edit a template or add one of your own:

1. From the main menu, select **Window**→**Preferences**.
2. In the left pane, select **C/C++**→**Code Templates**.
3 To edit a template, select it, then click **Edit**.

4 To add your own template, click **New**. A dialog for adding new templates appears:
Writing code

[Image of a C/C++ editor interface with a dialog box for adding templates]

Adding `#include` directives

To insert the appropriate `#include` directive for any documented QNX Neutrino function:

1. In the C/C++ editor, double-click the function name, but don’t highlight the parentheses or any leading tabs or spaces.

2. Right-click and select Add Include. The IDE automatically adds the `#include` statement to the top of the file, if it isn’t already there.

Hover help

The IDE’s hover help feature gives you the synopsis for a function while you’re coding. To use hover help:

- In the C/C++ editor, pause your pointer over a function. You’ll see a text box showing the function’s summary and synopsis:
Commenting-out code

You can easily add comments using either the C or C++ style, even to large sections of code. You can add // characters to the beginning of lines, letting you comment out large sections, even if they have /* */ comments.

When you uncomment lines, the editor removes the leading // characters from all lines that have them, so be careful not to accidentally uncomment sections. Also, the editor can comment or uncomment selected lines — if you highlight a partial line, the editor comments out the entire line, not just the highlighted section.

To comment or uncomment a block of code:
Writing code

1. In the C/C++ editor, highlight a section of code to be commented or uncommented. For one line, place your cursor on that line.

2. Right-click and select Comment or Uncomment.

Customizing the C/C++ editor

You can change the font, set the background color, show line numbers, and control many other visual aspects of the C/C++ editor. You can also configure context highlighting and change how the Code Assist feature works. You do all this in the C/C++ editor preferences dialog:

To access the C/C++ editor preferences dialog:

1. Select Window→Preferences.

2. In the left pane, select C/C++→Editor.
Using other editors

If you wish to use a different text editor than the one that’s built into the IDE, you can do so, but you’ll lose the integration of the various views and perspectives. For example, within the C/C++ editor, you can set breakpoints and then see them in the Breakpoints view, or put “to-do” markers on particular lines and see them in the Tasks view, or get hover help as you pause your cursor over a function name in your code, and so on.

If you want to use other editors, you can do so either outside or inside the IDE.

Outside the IDE

You can edit your code with an editor started outside of the IDE (e.g. from the command line). When you’re done editing, you’ll have to synchronize the IDE with the changes.

To synchronize the IDE with changes you’ve made using an editor outside of the IDE:

➤ In the C/C++ Projects view, right-click the tree pane and select Refresh. The IDE updates the display to reflect any changes you’ve made (such as creating new files).

Within the IDE

You can specify file associations that determine the editor you want to use for each file type. For example, you can tell the IDE to use an external program such as WordPad to edit all .h files. Once that preference is set, you can double-click a file in the C/C++ Projects view, and the IDE automatically opens the file in your selected program.

If the IDE doesn’t have an association set for a certain file type, it uses the host OS defaults. For example, on a Windows host, if you double-click a .doc file, Word or WordPad automatically launches and opens the file.
Creating files from scratch

By default, the IDE creates a simple “hello world” C/C++ source file for you, which you may or may not want to use as a template for your own code.

To create a new C/C++ file:

1. Highlight the project that contains the new file you’re creating.
2. Click the **New C/C++ Source File** button on the toolbar:

   ![New C/C++ Source File](image)

3. Enter (or select) the name of the folder where the file resides.
4. Name your file, then click Finish.

You should now see an empty text editor window, ready for you to begin working on your new file. Notice your filename highlighted in blue in the title bar above the editor.

More development features

Besides the features already described above, the IDE has several other helpful facilities worth exploring.

Tracking remaining work

The Problems view gives you a list of errors and warnings related to your projects. These are typically syntax errors, typos, and other programming errors found by the compiler:
More development features

Error markers

The IDE also shows corresponding markers in several other locations:

- C/C++ Projects view — on both the file that contained compile errors and on the project itself
- Outline view — in the method (e.g. main())
- C/C++ editor — on the left side, beside the offending line of code

Jumping to errors

To quickly go to the source of an error (if the IDE can determine where it is):

➤ In the Problems view, double-click the error marker ( ) or warning marker ( ). The file opens in the editor area, with the cursor on the offending line.

To jump to errors sequentially:

➤ Click the Jump to next error marker button ( ) or the Jump to previous error marker button ( ).

Filtering errors

Depending on the complexity and stage of your program, the IDE can generate an overwhelming number of errors. But you can customize the Problems view so you’ll see only the errors you want to see.

To access the error-filtering dialog:
More development features

➤ In the Problems view, click the Filter icon ( ).

The Filters dialog lets you adjust the scope of the errors shown in the Problems view. The more boxes checked, the more errors you’ll see.
More development features

![Filters](image)

<table>
<thead>
<tr>
<th>Type</th>
<th>Parent types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant Buildfile Problem</td>
<td>Problem</td>
</tr>
<tr>
<td>Build Path Problem</td>
<td>Problem; Text</td>
</tr>
<tr>
<td>C/C++ Problem</td>
<td>Problem; Text</td>
</tr>
<tr>
<td>Indexer Marker</td>
<td>Problem; Text</td>
</tr>
<tr>
<td>Java Problem</td>
<td>Problem; Text</td>
</tr>
<tr>
<td>Memory Error Marker</td>
<td>Task; Problem</td>
</tr>
</tbody>
</table>

- Enabled
- Limit visible items to: 100

- Select All
- Deselect All

- On any resource
- On any resource in same project
- On selected resource only
- On selected resource and its children
- On working set: <no working set selected>
- Select...

- Where description contains
- Where severity is:
  - Error
  - Warning
  - Info

- Restore Defaults

OK  Cancel
More development features

Tracking Tasks

The Tasks view is part of the core Eclipse platform. For more information about this view, follow these links in the Workbench User Guide: Reference → User interface information → View and Editors → Tasks view.

The Tasks view lets you track your tasks.

Setting reminders

The Tasks view lets you create your own tasks for the unfinished function you’re writing, the error-handling routine you want to check, or whatever.

You use the New Tasks dialog to add a personal task:

1. In the Tasks view, right-click the tasks pane and select Add Task or click the Add Task button in the Tasks view.

2. Complete the dialog for your task:
To remove a personal task:

➤ In the Tasks view, right-click the task and select **Delete**.

**Code synopsis**

The Outline view gives you a structural view of your C/C++ source code:

The view shows the elements in the source file in the order they occur, including functions, libraries, and variables. You may also sort the list alphabetically, or hide certain items (fields, static members, and nonpublic members).

If you click an entry in the Outline view, the editor’s cursor moves to the start of the item selected.
Checking your build

The Console view displays the output from the make utility:

```
make -k all --file=C:/QWin32/workspacetoptest/qWin32files3727.tmp
make -j 1 -cQ8x -fMakefile all
make[1]: Entering directory 'C:/QWin32/workspacetoptest/x86'
make -j 1 -c0-g -fMakefile all
make[2]: Entering directory 'C:/QWin32/workspacetoptest/x86/o-g'
C/C++ has 55 files 3886675 lines 3577598 of which 1779975 are source
centric

C:/QWin32/workspacetoptest/x86/o-g -ICl/C:/QWin32/workspacetoptest/x86/o-g
-ICl/C:/QWin32/workspacetoptest/x86/o-g
make[2]: Leaving directory 'C:/QWin32/workspacetoptest/x86/o-g'
```

Customizing the Console view

You can choose to clear the Console view before each new build or let the output of each subsequent build grow in the display. You can also have the Console view appear on top of the other stacked views whenever you build.

To set the preferences for the Console view:

1. From the main menu, select **Window → Preferences**.
2. In the left pane, select **C/C++ → Build Console**.
More development features

Accessing source files for functions

While editing source code in the editor, you can select a function name, press F3, and the editor immediately jumps to the source file for that function (if the file is also in your project).

For more information on the C/C++ Development perspective, go to: Help→Help Contents→C/C++ Development User Guide.

Opening headers

You can select a header (such as stdio.h) in the C/C++ editor and press Ctrl – Shift – o to open the header file in the editor. You can also right-click the header file’s name in the Outline view, then choose Open.

Many of the enhanced source navigation and code development accelerators available in the C/C++ editor are extracted from the source code. To provide the most accurate data representation, the
More development features

project must be properly configured with the include paths and defines used to compile the source.

For QNX projects, the standard include paths and defines are set automatically based on the compiler and architecture. Additional values can be set using the project’s properties.

For Standard C/C++ Make projects, you must define the values yourself. These values can be set manually using the Paths and Symbols tab of the project’s properties, or they can be set automatically using the Set QNX Build Environment... item in the project’s context menu.

To set the include paths and defines for a Standard C/C++ Make project:

1. In the C/C++ Projects view, right-click your project and select Set QNX Build Environment....

   The Set QNX Build Environment wizard appears.
More development features

2 Select one or more Standard C/C++ Make projects to update and click Next.

The Compiler/Architecture Selection panel appears.
Select the appropriate Compiler, Language, and Architecture for your project, and click Finish.
Chapter 4
Managing Source Code

In this chapter...

CVS and the IDE 111
Importing existing source code into the IDE 114
Using container projects 128
Importing a BSP or other QNX source packages 134
Exporting projects 140
This chapter describes managing source code from within the IDE.

CVS and the IDE

CVS is the default source-management system in the IDE. Other systems (e.g. ClearCase) are also supported.

The CVS Repository Exploring perspective lets you bring code from CVS into your workspace. If another developer changes the source in CVS while you’re working on it, the IDE helps you synchronize with CVS and resolve any conflicts. You can also choose to automatically notify the CVS server whenever you start working on a file. The CVS server then notifies other developers who work on that file as well. Finally, the CVS Repository Exploring perspective lets you check your modified code back into CVS.

The IDE connects to CVS repositories that reside only on remote servers — you can’t have a local CVS repository (i.e. one that resides on your host computer) unless it’s set up to allow CVS pserver, ext, or extssh connections.
Local history feature

The IDE lets you “undo” changes with its local history. While you’re working on your code, the IDE automatically keeps track of the changes you make to your file; it lets you roll back to an earlier version of a file that you saved but didn’t commit to CVS.

For more on the IDE’s local history feature, follow these links in the Workbench User Guide: Reference→User interface information→Development environment→Local history.

Project files (.project and .cdtproject)

For each project, the IDE stores important information in these two files:

- .project
- .cdtproject

You must include both files with your project when you commit it to CVS.

Core Eclipse documentation on using CVS in the IDE

Since the CVS Repository Exploring perspective is a core Eclipse feature, you’ll find complete documentation in the Eclipse Workbench User Guide. Follow these links:

- Tips and Tricks, scroll down to the Team - CVS section
- Tasks→Working in the team environment with CVS

This table may help you find information quickly in the Workbench User Guide:
<table>
<thead>
<tr>
<th>If you want to:</th>
<th>Go to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect to a CVS repository</td>
<td>Tasks→Working in the team environment with CVS→Working with a CVS repository→Creating a CVS repository location</td>
</tr>
<tr>
<td>Check code out of CVS</td>
<td>Tasks→Working in the team environment with CVS→Working with projects shared with CVS→Checking out a project from a CVS repository</td>
</tr>
<tr>
<td>Synchronize with a CVS repository</td>
<td>Tasks→Working in the team environment with CVS→Synchronizing with the repository, especially the Updating section</td>
</tr>
<tr>
<td>See who’s also working on a file</td>
<td>Tasks→Working in the team environment with CVS→Finding out who’s working on what: watch/edit</td>
</tr>
<tr>
<td>Resolve CVS conflicts</td>
<td>Tasks→Working in the team environment with CVS→Synchronizing with the repository→Resolving conflicts</td>
</tr>
<tr>
<td>Prevent certain files from being committed to CVS</td>
<td>Tasks→Working in the team environment with CVS→Synchronizing with the repository→Version control life cycle: adding and ignoring resources</td>
</tr>
</tbody>
</table>

*continued…*
Importing existing source code into the IDE

<table>
<thead>
<tr>
<th>If you want to:</th>
<th>Go to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create and apply a patch</td>
<td>Tasks→Working in the team environment with CVS→Working with patches</td>
</tr>
<tr>
<td>Track code changes that haven’t been</td>
<td>Tasks→Working with local history, especially the Comparing resources with the local history section</td>
</tr>
<tr>
<td>been committed to CVS</td>
<td>Reference→Team Support→CVS</td>
</tr>
<tr>
<td>View an online FAQ about the CVS Repository Exploring perspective</td>
<td>Reference→Team Support→CVS</td>
</tr>
</tbody>
</table>

Importing existing source code into the IDE

As with many tasks within the IDE, there’s more than one way to bring existing source files into your workspace:

- *filesystem drag-and-drop* — from a Windows host, you can drag-and-drop (or copy and paste) individual files from the filesystem into your project in your workspace.

- *CVS repository* — you can use the CVS Repositories view to connect to a CVS repository and check out projects, folders, or files into your workspace.

- *Import wizard* — this IDE wizard lets you import existing projects, files, and even files from ZIP archives into your workspace.

- *linked resources* — this lets you work with files and folders that reside in the filesystem outside your project’s location in the workspace. You might use linked resources, for example, if you have a source tree that’s handled by some other source-management tool outside of the IDE. (For more on linked resources, follow these links in the Workbench User Guide: Concepts→Workbench→Linked resources.)
Importing existing source code into the IDE

Whatever method you use, you always need to set up an IDE project in your workspace in order to work with the resources you’re importing.

If you’re importing code that uses an existing build system, you may need to provide a Makefile with all: and clean: targets that call your existing build system.

For example, if you’re using the jam tool to build your application, your IDE project Makefile might look like this:

```makefile
all:
    jam -fbuild.jam

clean:
    jam -fbuild.jam clean
```

Projects within projects?

Suppose you have an existing source hierarchy that looks something like this:
Importing existing source code into the IDE

In order to work efficiently with this source in the IDE, each component and subcomponent should be a “subproject” within the one main project. (You could keep an entire hierarchy as a single project if you wish, but you’d probably find it cumbersome to build and work with such a monolith.)

Unfortunately, the current version of Eclipse (3.0) in QNX Momentics 6.3 doesn’t support nesting projects as such. So how would you import such a source tree into Eclipse 3.0?

Step 1

First, in your workspace create a single project that reflects all the components that reside in your existing source tree:

1. Select \texttt{File}→\texttt{New→Project}…
2. Select the type of project (e.g. Standard Make C project).
3 Name your project (e.g. `EntireSourceProjectA`).

4 Unselect **Use Default Location**, because we need to tell the IDE where the resources reside in the filesystem (since they don’t reside in your workspace).

5 In the **Location:** field, type in the path to your source (or click **Browse**...).

6 Click **Finish**. You should now have a project that looks something like this in the C/C++ Projects view:

![C/C++ Projects](image.png)

**Step 2**

Now we’ll create an individual project (via **File**→**New**→**Project**...) for each of the existing projects (or components) in your source tree.
Importing existing source code into the IDE

In this example, we’ll create a separate project for each of the following source components:

- **ComponentA**
- **ComponentB**
- **SubcomponentC**
- **SubcomponentD**

1. Select **File → New → Project...**
2. Select the type of project (e.g. Standard Make C project).
3. Name your project (e.g. `Project_ComponentA`).
4. Check **Use default location**, because we want the IDE to create a project in your workspace for this and all the other components that comprise your `EntireSourceProjectA`. In the next step, we’ll be linking each project to the actual location of the directories in your source tree.
5. Click **Finish**, and you’ll see `Project_ComponentA` in the C/C++ Projects view.

**Step 3**

Next we’ll link each individual project in the IDE to its corresponding directory in the source tree:

1. Select **File → New → Folder**.
2. Make sure your new project (`Project_ComponentA`) is selected as the parent folder.
3. Name the folder (e.g. `ComponentA`).
4. Click the **Advanced>>** button.
5 Check **Link to folder in the file system**.

6 Enter the path to that folder in your source tree (or use **Browse...**).
Importing existing source code into the IDE

7 Click Finish. Your Project_ComponentA project should now show a folder called ComponentA, the contents of which actually reside in your source tree.

Step 4

Now we’ll need to tell the IDE to build Project_ComponentA in the ComponentA linked folder that you just created in your workspace:

1 In the C/C++ Projects view, right-click Project_ComponentA, then select Properties from the context menu.

2 Select C/C++ Make Project.

3 In the Make Builder tab, set the Build Directory to ComponentA in your workspace.

Now when you go to build Project_ComponentA, the IDE builds it in the ComponentA folder in your workspace (even though the source actually resides in a folder outside your workspace).
CAUTION: Linked resources let you overlap files in your workspace, so files from one project can appear in another project. But keep in mind that if you change a file or other resource in one place, the duplicate resource is also affected. If you delete a duplicate resource, its original is also deleted!

Special rules apply when working with linked resources. Since a linked resource must reside directly below a project, you can’t copy or move a linked resource into other folders. If you delete a linked resource from your project, this does not cause the corresponding resource in the filesystem to also be deleted. But if you delete child resources of linked folders, this does delete those child resources from the filesystem!

Filesystem drag-and-drop

On Windows hosts, you can select files or folders and drop them into projects in the Navigator view:

1. Create a new project. If your existing code has an existing build procedure, use a Standard Make C/C++ Project. If not, you can use a QNX C/C++ Project or a Standard Make C/C++ Project.

2. Switch to the Navigator view.

3. Select one or more source files or folders in the Windows Explorer, then drag them into the project. The files are copied into your project workspace.

You can also use Cut, Copy, and Paste to move or copy files into a project from Windows Explorer.

CVS repository

Using the CVS Repository Exploring perspective, you can check out modules or directories into existing projects, or to create new projects.
Importing existing source code into the IDE

Bringing code into the IDE from CVS differs slightly depending on what you’re importing:

- an existing C/C++ project
- existing C/C++ code that isn’t part of a project
- existing C/C++ code that needs to be added to an existing project

**Importing a C/C++ project from CVS**

To check out an existing C/C++ project (either a QNX project or a Standard Make C/C++ project) from the CVS repository into your workspace:

1. Right-click the project in the CVS Repositories view and choose **Check Out** from the menu.

   The IDE creates a project with the same name as the CVS module in your workspace. The project is automatically recognized as a Standard Make C/C++ or QNX C/C++ project (if the project has `.project` and `.cdtproject` files).

2. If the project is a QNX project:
   1. Right-click the new project in the Navigator or C/C++ Projects view and choose **Properties**.
   2. Click the **Build Variants** tab, which displays a warning:

      ![Warning: At least one platform should be selected]

   3. Select one or more of the build variants, then click **OK**.

**Importing C/C++ code from CVS**

To check out existing C/C++ code that isn’t part of a project:

1. Right-click the module or directory in the CVS Repositories view and choose **Check Out As...** from the menu.

   The IDE displays the Check Out As wizard.
Importing existing source code into the IDE

The Check Out As wizard.

2 Choose how to check out this project:

- as a project configured using the New Project wizard
  or:
- as a new project in the workspace
  or:
- **Standard Make C/C++ Project** – Use a Standard Make C/C++ project if you need to create your own Makefile to integrate with an existing build process.

Choose the workspace location for this project, then the CVS tag to check out. Click **Finish** to exit the **Check Out As** dialog.
Importing existing source code into the IDE

Click Next to continue.

3 If you’re creating or checking out a QNX project:
   3a Right-click the new project in the Navigator or C/C++
      Projects view and choose Properties.
   3b Click the Build Variants tab, which displays a warning:
      At least one platform should be selected
   3c Select one or more of the build variants, then click OK.

4 If you’re creating a Standard Make C/C++ project, create a new
   Makefile with appropriate all: and clean: targets.

Importing C/C++ code into an existing project

To import a directory full of C/C++ code into an existing project:

1 Right-click the module or directory in the CVS Repositories
   view and choose Check Out As... from the menu.
   The IDE displays the Check Out As dialog.
Importing existing source code into the IDE

Click **Next** to continue. The IDE displays the Check Out Into dialog.
Importing existing source code into the IDE

2 Select an existing project from the list, then click Finish to add the code from CVS to the selected project.

Import wizard

Use the Import wizard to bring files or folders into an existing project from a variety of different sources, such as:

- an existing container project
- an existing project
- another directory
- a QNX Board Support Package
- a QNX `mkifs` Buildfile
Importing existing source code into the IDE

- a QNX Source Package
- a Team Project Set
- a Zip file

For details, see “Importing projects” in the Common Wizards Reference chapter.

Linked resources

As an alternative to dragging-and-dropping, you can link files and folders into a project. This lets you include files in your project, even if they need to reside in a specific place on your filesystem (because of a restrictive source control system, for example).

To add a linked resource to a project in the C/C++ Project or Navigator view:

1. Right-click on a project, then choose File or Folder from the New menu.
   The New File or New Folder dialog appears.

2. Enter the new file or folder name in the File Name field.

3. Click the Advanced >> button, and check the Link to file in the file system or Link to folder in the file system check box.

4. Enter the full path to the file or folder, or click the Browse… button to select a file or folder.

5. Use the Variables… button to define path variables for use in the file or folder path:
Using container projects

6 Click **Finish** to link the file or folder into your project.

See Concepts→Workbench→Linked resources in the Workbench User Guide for more information about linked resources.

**Using container projects**

A *container* is a project that creates a logical grouping of subprojects. Containers can ease the building of large multiproject systems. You can have containers practically anywhere you want on the filesystem, with one exception: containers can’t appear in the parent folders of other projects (because this would create a projects-in-projects problem).

Containers let you specify just about any number of build configurations (which are analogous to build variants in C/C++ projects). Each build configuration contains a list of subprojects and specifies which variant to be built for each of those projects. Note that each build configuration may contain a different list and mix of subprojects (e.g. QNX C/C++ projects, Standard Make C/C++ projects, or other container projects.)
Creating a container project

In order to create a container, you must have at least one subproject that you want to contain.

To create a container project:

1. Select File → New → Project…, then QNX → C/C++ Container Project.
2. Name the container.
3. Click Next.
4. Click Add Project…
5. Now select all the projects (which could be other containers) that you want included in this container:

![New Project dialog box](image)
Using container projects

Each subproject has a **make** targets entry under the Target field. The “Default” entry means “don’t pass any targets to the **make** command.” QNX C/C++ projects interpret this as “rebuild”. If a subproject is also a container project, this field represents the build configuration for that container.

You can set the default for QNX C/C++ projects by opening the Preferences dialog box (**Window** → **Preferences** in the menu), then choosing **QNX** → **Container properties**.

6 Select the build variant for each project you wish to build. You can choose **All** (for **every** variant that has already been created in the project’s folder) or **All Enabled** (for just the variants you’ve selected). Note that the concept of variants makes sense only for QNX C/C++ projects.

The **Stop on error** column controls whether the build process for the container stops at the first subproject to have an error or continues to build all the remaining subprojects.

7 If you want to reduce clutter in the C/C++ Projects view, then create a **working set** for your container. The working set contains all the projects initially added to the container.

To select a working set, click the down-arrow at the top of the C/C++ Projects view pane and select the working set you want. Note that the working set created when the container was created has the same name as the container.

If you add or remove elements to a container project later, the working set is *not* updated automatically.

8 Click **Finish**. The IDE creates your container project.
Setting up a build configuration

Just as QNX C/C++ projects have build variants, container projects have build configurations. Each configuration can be entirely distinct from other configurations in the same container. For example, you could have two separate configurations, say Development and Released, in your top-level container. The Development configuration would build the Development configuration of any subcontainers, as well as the appropriate build variant for any subprojects. The Released configuration would be identical, except that it would build the Released variants of subprojects.

Note that the default configuration is the first configuration that was created when the container project was created.

To create a build configuration for a container:

1. In the C/C++ Projects view, right-click the container.
2. Select Create Container Configuration….
3. In the Container Build Configuration dialog, name the configuration.
4. Click Add Project, then select all the projects to be included in this configuration.
5. Change the Variant and Stop on error entries for each included project, as appropriate.

If you want to change the build order, use the Shift Up or Shift Down buttons.

6. Click OK.

Editing existing configurations

There are two ways to change existing configurations for a container project, both of which appear in the right-click menu:
Using container projects

- Properties
- Build Container Configuration

Although you can use either method to edit a configuration, you might find changing the Properties easier because it shows you a tree-view of your entire container project.

Note also that you can edit only those configurations that are immediate children of the root container.

Editing via project Properties

You can use the container project’s Properties to:

- add new configurations
- add projects to existing configurations
- specify which variant of a subproject should be built

To edit a configuration:

1. Right-click the container project and select Properties.
2. In the left pane, select Container Build Configurations.
3. Expand the project in the tree-view on the right.
4. Select the configuration you want to edit. Configurations are listed as children of the container.
5. Click the Edit button at the right of the dialog. This opens the familiar Container Build Configuration dialog (from the New Container wizard), which you used when you created the container.
6. Make any necessary changes — add, delete, reorder projects, or change which make target or variant you want built for any given project.
Using container projects

While editing a configuration, you can include or exclude a component from the build just by checking or unchecking the component. Note that if you exclude a component from being built, it’s not removed from your container.

7 Click OK, then click OK again (to close the Properties dialog).

Editing via the Build Container Configuration... item

You can access the Container Build Configuration dialog from the container project’s right-click menu.

Note that this method doesn’t show you a tree-view of your container.

To edit the configuration:

1 Right-click the container project, then select Build Container Configuration... .

2 Select the configuration you want to edit from the list.

3 Click the Edit button. This opens the familiar Container Build Configuration dialog (from the New Container wizard), which you used when you created the container.

4 Make any necessary changes — add, delete, reorder projects, or change which make target or variant you want built for any given project.

5 Click OK, then click OK again (to save your changes and close the dialog).

Building a container project

Once you’ve finished setting up your container project and its configurations, it’s very simple to build your container:

1 In the C/C++ Projects view, right-click your container project.

2 Select Build Container Configuration... .

3 Choose the appropriate configuration from the dialog.
Importing a BSP or other QNX source packages

4 Click **Build**.

A project’s build variant selected in the container configuration is built, regardless of whether the variant is selected in the C/C++ project’s properties. In other words, the container project overrides the individual project’s build-variant setting during the build.

The one exception to this is the **All Enabled** variant in the container configuration. If the container configuration is set to build all enabled variants of a project, then only those variants that you’ve selected in the project’s build-variant properties is built.

To build the default container configuration, you can also use the **Build** item in the right-click menu.

**Importing a BSP or other QNX source packages**

QNX BSPs and other source packages (e.g. DDKs) are distributed as .zip archives. The IDE lets you import these packages into the IDE:

<table>
<thead>
<tr>
<th>When you import:</th>
<th>The IDE creates:</th>
</tr>
</thead>
<tbody>
<tr>
<td>QNX BSP source package</td>
<td>A System Builder project</td>
</tr>
<tr>
<td>QNX C/C++ source package</td>
<td>A C or C++ application or library project</td>
</tr>
</tbody>
</table>

**Step 1: Use File→Import…**

You import a QNX source archive using the standard Eclipse import dialog:
Importing a BSP or other QNX source packages

If you’re importing a BSP, select **QNX Board Support Package**. If you’re importing a DDK, select **QNX Source Package**.

As you can see, you can choose to import either a QNX BSP or a “source package.” Although a BSP is, in fact, a package that contains source code, the two types are structured differently and generates different types of projects. If you try to import a BSP archive as a QNX Source Package, the IDE won’t create a System Builder project.
Importing a BSP or other QNX source packages

Step 2: Select the package

After you choose the type of package you’re importing, the wizard presents you with a list of the packages found in

$QNX\_TARGET/usr/src/archives on your host:

Notice that as you highlight a package in the list, a description for that package is displayed.

To add more packages to the list:

1. Click the Select Package... button.
Importing a BSP or other QNX source packages

2 Select the .zip source archive you want to add.

Step 3: Select the source projects

Each source package contains several components (or projects, in IDE terms). For the package you selected, the wizard gives you a list of each source project contained in the archive:

You can decide to import only certain parts of the source package; simply uncheck the entries you don’t want (they’re all selected by default). Again, as you highlight a component, you’ll see its description in the bottom pane.
Importing a BSP or other QNX source packages

**Step 4: Select a working set**

The last page of the import wizard lets you name your source projects. You can specify:

- Working Set Name — to group all related imported projects together as a set
- Project Name Prefix — for BSPs, this becomes the name of the System Builder project; for other source projects, this prefix lets you import the same source several times without any conflicts.

![Import QNX BSP window](image-url)
Importing a BSP or other QNX source packages

If you plan to import a source BSP and a binary BSP into the IDE, remember to give each project a different name.

If you import dual-endian BSP’s, the wizards displays this informational message:

If you add build variants, you need to copy the CPU-specific files to the new variant’s build directories.

Step 5: Build

When you finish with the wizard, it creates all the projects and brings in the source from the archive. The wizard then asks if you want to build all the projects you’ve just imported.

If you answer Yes, the IDE begins the build process, which may take several minutes (depending on how much source you’ve imported).

If you decide not to build now, you can always do a Rebuild All from the main toolbar’s Project menu at a later time.

If you didn’t import all the components from a BSP package, you can bring in the rest of them by selecting the System Builder project and opening the import wizard (right-click the project, then select Import...). The IDE detects your selection and then extends the existing BSP (rather than making a new one).
Exporting projects

QNX BSP perspective

When you import a QNX Board Support Package, the IDE opens the QNX BSP perspective. This perspective combines the minimum elements from both the C/C++ Development perspective and the System Builder perspective:

Exporting projects

You can export projects to your filesystem or to .zip files by doing one of the following:

- Drag a file or folder from a project to your filesystem.

  Press Alt while dragging to copy the file or folder instead of moving it out of the project.

- Use the Copy (to copy) or Cut (to move) context-menu items, then Paste the file into your filesystem.

- Export to the filesystem using the Export... command.
Exporting projects

- Export to a .zip file using the Export... command.

Using the Export... command

The Export wizard helps you export entire projects to your filesystem or a .zip file.

To export one or more projects:

1. Choose File→Export... (or Export... from the Navigator context menu).

The Export wizard appears.
Exporting projects

2 To export your project to the filesystem, choose **File system**. To export your project to a **.zip** file, choose **Zip file**. Click **Next**. The Export wizard’s next panel appears.

3 Select the projects you want to export. You can also select or deselect specific files in each project.

To select files based on their extensions, click the **Select Types**... button. The Select Types dialog box appears.
Exporting projects

When you're done selecting projects and files, click Finish.
Exporting projects

If you export more than one project, and someone imports from the resulting filesystem or `.zip` file, they’ll get one project containing all of the projects you exported.
Chapter 5
Debugging Programs

In this chapter...

Introduction 147
Debugging your program 148
Controlling your debug session 152
More debugging features 159
This chapter shows you how to work with the debugger.

Introduction

One of the most frequently used tools in the traditional design-develop-debug cycle is the source-level debugger. In the IDE, this powerful tool provides an intuitive debugging environment that’s completely integrated with the other workbench tools, giving you the flexibility you need to best address the problems at hand.

Have you ever had to debug several programs simultaneously? Did you have to use separate tools when the programs were written in different languages or for different processors? The IDE’s source debugger provides a unified environment for multiprocess and multithreaded debugging of programs written in C, C++, Embedded C++, or Java. You can debug such programs concurrently on one or multiple remote target systems, or locally if you’re doing Neutrino self-hosted development.

In order to use the full power of the Debug perspective, you must use executables compiled for debugging. These executables contain additional debug information that lets the debugger make direct associations between the source code and the binaries generated from
Debugging your program

that original source. With QNX Make projects, an executable compiled for debugging has “.g” appended to its filename.

The IDE debugger uses GDB as the underlying debug engine. It translates each GUI action into a sequence of GDB commands, then processes the output from GDB to display the current state of the program being debugged.

The IDE updates the views in the Debug perspective only when the program is suspended.

Editing your source after compiling causes the line numbering to be out of step because the debug information is tied directly to the source. Similarly, debugging an optimized binary can also cause unexpected jumps in the execution trace.

Debugging your program

Building an executable for debugging

Although you can debug a regular executable, you’ll get far more control by building debug variants of the executables. When you created your project, you may have already set the option to cause the IDE to build an executable that’s ready for debugging. If so, you should have executables with .g appended to the filename. If not, you must tell the IDE to build debug variants:

1. In the C/C++ Projects view (or the Navigator view), right-click the project and select Properties.
2. In the left pane, select QNX C/C++ Project.
3. In the right pane, select the Build Variants tab.
4. Under your selected build variants, make sure Debug is enabled:
5 Click **Apply**.

6 Click **OK**.

7 Rebuild your project (unless you’re using the IDE’s autobuild feature).

For more information about setting project options, see the Common Wizards Reference chapter.

**Starting your debugging session**

For a full description of starting your programs and the launch configuration options, see the Launch Configurations Reference chapter.

After building a debug-enabled executable, your next step is to create a launch configuration for that executable so you can run and debug it:
1. From the main menu, select Run→Debug→. The launch configurations dialog appears.

2. Create a launch configuration as you normally would, but don’t click OK.

3. Select the Debugger tab.
4 Make sure **Stop at main() on startup** is set.

5 Click **Apply**.

6 Click **Debug**.

By default, the IDE automatically changes to the Debug perspective when you debug a program. If the default is no longer set, or if you wish to change to a different perspective when you debug, see the “Setting execution options” section in the Launch Configurations Reference chapter.

If launching a debugging session doesn’t work when connected to the target with **qconn**, make sure **pdebug** is on the target in `/usr/bin`. 
Debugging sessions stay in the Debug perspective until you remove them. These consume resources on your development host and your debugging target. You can automatically delete the completed debug session by checking the **Remove terminated launches when a new launch is created** box on the Run/Debug→Launching pane of the Preferences dialog.

**Controlling your debug session**

The contents of all views in the Debug perspective are driven by the selections you make in the Debug view.

The Debug view lets you manage the debugging or running of a program in the workbench. This view displays the stack frame for each suspended threads for each target you’re debugging. Each thread in your program appears as a node in the tree. The view displays the process for each program you’re running.

The Debug view shows the target information in a tree hierarchy as follows (shown here with a sample of the possible icons):
Controlling your debug session

<table>
<thead>
<tr>
<th>Session item</th>
<th>Description</th>
<th>Possible icons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Launch instance</td>
<td>Launch configuration name and type (e.g. Stack Builder [C/C++ QNX QConn (IP)])</td>
<td></td>
</tr>
<tr>
<td>Debugger instance</td>
<td>Debugger name and state (e.g. QNX GDB Debugger (Breakpoint hit))</td>
<td></td>
</tr>
<tr>
<td>Thread instance</td>
<td>Thread number and state (e.g. Thread[1] (Suspended))</td>
<td></td>
</tr>
<tr>
<td>Stack frame instance</td>
<td>Stack frame number, function, filename, and line number</td>
<td></td>
</tr>
</tbody>
</table>

The number beside the thread label is a reference counter for the IDE, not a thread ID (TID) number.

The IDE displays stack frames as child elements, and gives the reason for the suspension (e.g. end of stepping range, breakpoint hit, signal received, and so on). When a program exits, the IDE displays the exit code.

The label includes the thread’s state. In the example above, the thread was suspended because the program hit a breakpoint. You can’t suspend only one thread in a process; suspension affects all threads.

The Debug view also drives the C/C++ editor; as you step through your program, the C/C++ editor highlights the location of the execution pointer.
Controlling your debug session

Using the controls

After you start the debugger, it stops (by default) in main() and waits for your input. (For information about changing this setting, see the “Debugger tab” section in the Launch Configurations Reference chapter.)

The debugging controls appear in the following places (but not all together in any one place):

- at the top of the Debug view as buttons
- in the Debug view’s right-click context menu
- in the main menu under Run (with hotkeys)
- in the C/C++ editor

The controls are superseded by breakpoints. For example, if you ask the program to step over a function (i.e. run until it finishes that function) and the program hits a breakpoint, the program pauses on that breakpoint, even though it hasn’t finished the function.

The icons and menu items are context-sensitive. For example, you can use the Terminate action to kill a process, but not a stack frame.

<table>
<thead>
<tr>
<th>Action</th>
<th>Icon</th>
<th>Hotkey</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resume</td>
<td>🎁</td>
<td>F8</td>
<td>Run the process freely from current point.</td>
</tr>
<tr>
<td>Suspend</td>
<td>🕹️</td>
<td></td>
<td>Regain control of the running process.</td>
</tr>
<tr>
<td>Terminate</td>
<td>🕵️</td>
<td></td>
<td>Kill the process.</td>
</tr>
<tr>
<td>Restart</td>
<td>🔄️</td>
<td></td>
<td>Rerun the process from the beginning.</td>
</tr>
</tbody>
</table>

continued...
Controlling your debug session

<table>
<thead>
<tr>
<th>Action</th>
<th>Icon</th>
<th>Hotkey</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resume without signal</td>
<td>![Resume Icon]</td>
<td>F0</td>
<td>Resume the execution of a process without delivering pending signals.</td>
</tr>
<tr>
<td>Step Into</td>
<td>![Step Into Icon]</td>
<td>F5</td>
<td>Step forward one line, going into function calls.</td>
</tr>
<tr>
<td>Step Over</td>
<td>![Step Over Icon]</td>
<td>F6</td>
<td>Step forward one line without going into function calls.</td>
</tr>
<tr>
<td>Run to return</td>
<td>![Run to Return Icon]</td>
<td>F7</td>
<td>Finish this function.</td>
</tr>
<tr>
<td>Resume at line</td>
<td>![Resume at Line Icon]</td>
<td></td>
<td>Resume the execution of the process at the specified line. Using this action to change into a different function may cause unpredictable results.</td>
</tr>
</tbody>
</table>

You can control your debug session in various ways:

- from the Debug view
- using hotkeys
- from the C/C++ editor

**From the Debug view**

You’ll probably use the Debug view primarily to control your program flow.

To control your debug execution:
Controlling your debug session

1. In the Debug view, select the thread you wish to control.
2. Click one of the stepping icons (e.g. Step Into) in the Debug view’s toolbar. Repeat as desired.
3. Finish the debug session by choosing one of the debug launch controls (e.g. Disconnect). For details, see the section “Debug launch controls” in this chapter.

Using hotkeys

Even if you’re running your debug session without the Debug view showing, you can use the hotkeys (or the Run menu) to step through your program. You can enable the debug hotkeys in any perspective.

You can easily customize these keys through the Preferences dialog:

To customize the debug hotkeys:

1. Choose Window → Preferences from the menu. The Preferences dialog is displayed.
Controlling your debug session

2 Choose **Workbench→Keys** in the list on the left.

3 Choose **Run/Debug** in the **Category** drop-down. The commands for running and stepping through a program are displayed.

4 Select a command in the **Name** drop-down. Its scope, configuration, and current key sequence (if any) are listed in the **Assignments** box.

5 To assign this command to a new hotkey, click in the **Key sequence** box, then press the key(s) for your new hotkey.

6 Click the **Add** button to assign the newly created hotkey to the selected command.

7 Click **OK** to activate your new hotkeys.

**From the C/C++ editor**

You can control your debug session using the C/C++ editor by having the program run until it hits the line your cursor is sitting on (i.e. the **gdb until** command). If the program never hits that line, the program runs until it finishes.

You can also use the C/C++ editor’s context menu to resume execution at a specific line, or to add a watch expression.

To use the C/C++ editor to debug a program:

1 In the editor, select a file associated with the process being debugged.

2 Left-click to insert the cursor where you want to interrupt the execution.

3 Right-click near the cursor and select **Run To Line**, **Resume at line** or **Add watch expression**.
Controlling your debug session

Note that Run To Line works only in the current stack frame. That is, you can use Run to Line within the currently executing function.

Debug launch controls

In addition to controlling the individual stepping of your programs, you can also control the debug session itself (e.g. terminate the session, stop the program, and so on) using the debug launch controls available in the Debug view (or in the view’s right-click menu).

As with the other debug controls, these are context-sensitive; some are disabled depending on whether you’ve selected a thread, a process, and so on, in the Debug view.

<table>
<thead>
<tr>
<th>Action</th>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminate</td>
<td></td>
<td>Kill the selected process.</td>
</tr>
<tr>
<td>Terminate &amp; Remove</td>
<td></td>
<td>Kill the selected process and remove it from the Debug view.</td>
</tr>
<tr>
<td>Terminate All</td>
<td></td>
<td>Kill all active processes in the Debug view.</td>
</tr>
<tr>
<td>Disconnect</td>
<td></td>
<td>Detach the debugger (i.e. gdb) from the selected process (useful for debugging attached processes).</td>
</tr>
</tbody>
</table>

continued...
More debugging features

<table>
<thead>
<tr>
<th>Action</th>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove All Terminated Launches</td>
<td></td>
<td>Clear all the killed processes from the Debug view.</td>
</tr>
<tr>
<td>Relaunch</td>
<td></td>
<td>Restart the process.</td>
</tr>
</tbody>
</table>

Disassembly mode

You can also examine your program as it steps into functions that you don’t have source code for, such as `printf()`. Normally, the debugger steps over these functions, even when you click Step Into. When the instruction pointer enters functions for which it doesn’t have the source, the IDE shows the function in the Disassembly view.

To show the Disassembly view:

➤ From the menu, choose Window→Show View→Disassembly. The workbench adds the Disassembly view to the Debug perspective.

More debugging features

Besides the Debug view, you’ll find several other useful views in the Debug perspective:

<table>
<thead>
<tr>
<th>To:</th>
<th>Use this view:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inspect variables</td>
<td>Variables</td>
</tr>
<tr>
<td>Use breakpoints and watchpoints</td>
<td>Breakpoints</td>
</tr>
<tr>
<td>Evaluate expressions</td>
<td>Expressions</td>
</tr>
<tr>
<td>Inspect registers</td>
<td>Registers</td>
</tr>
</tbody>
</table>

continued…
More debugging features

<table>
<thead>
<tr>
<th>To:</th>
<th>Use this view:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inspect a process’s memory</td>
<td>Memory</td>
</tr>
<tr>
<td>Inspect shared library usage</td>
<td>Shared Libraries</td>
</tr>
<tr>
<td>Monitor signal handling</td>
<td>Signals</td>
</tr>
<tr>
<td>View your output</td>
<td>Console</td>
</tr>
<tr>
<td>Debug with GDB</td>
<td>Console</td>
</tr>
</tbody>
</table>

Inspecting variables

The Variables view displays information about the variables in the currently selected stack frame:

At the bottom of the view, the Detail pane displays the value of the selected variable (as evaluated by `gdb`).

If you happen to have multiple variables of the same name, the one most in scope is evaluated.

When the execution stops, the changed values are highlighted in red (by default). Like the other debug-related views, the Variables view doesn’t try to keep up with the execution of a running program; it updates the display only when execution stops.
More debugging features

You can decide whether or not to display the variable type (e.g. `int`) by clicking the **Show Type Names** toggle button (ackle).

You can also control whether or not the IDE tracks your program’s variables. See the “Debugger tab” section in the Launch Configurations Reference chapter.

**Inspecting global variables**

By default, global variables aren’t displayed in the Variables view. To add global variables to the view:

1. In the Variables view, click the **Add Global Variables** button (ackle).

2. Select one or more symbols in the Selection Needed dialog.
More debugging features

Changing variable values

While debugging a program, you may wish to manually change the value of a variable to test how your program handles the setting or to speed through a loop.

To change a variable value while debugging:

1. In the Variables view, right-click the variable and select the Change Value… item.

3. Click OK to add the selected globals to the Variables view.
More debugging features

You can also change a variable’s value in the Detail pane at the bottom of the Variables view. Click the value, change it, then press Ctrl – S to save the new value.

Controlling the display of variables

You can prevent the debugger from reading the value of variables from the target. You might use this feature for variables that are either very sensitive or specified as volatile.

To enable or disable a variable:

1. In the Variables view, right-click the variable and select either Enable or Disable. (You can disable all the variables in your launch configuration. See the “Debugger tab” section in the Launch Configurations Reference chapter.)

To change a variable to a different type:

1. In the Variables view, right-click the variable.

2. Select one of the following:
More debugging features

Cast To Type...

Cast the variable to the type you specify in the field (e.g. `int`).

![Cast To Type dialog](image)

Restore Original Type

Cancel your Cast To Type command.

Format, followed by a type

Display the variable in a different format (e.g. hexadecimal).

Display As Array

Display the variable as an array with a length and start index that you specify. This option is available only for pointers.

Using breakpoints and watchpoints

The Breakpoints view lists all the breakpoints and watchpoints you’ve set in your open projects:
A breakpoint makes your program stop whenever a certain point in the program is reached. For each breakpoint, you can add conditions to better control whether or not your program stops.

A watchpoint is a special breakpoint that stops the program’s execution whenever the value of an expression changes, without specifying where this may happen. Unlike breakpoints, which are line-specific, watchpoints are event-specific and take effect whenever a specified condition is true, regardless of when or where it occurred.

<table>
<thead>
<tr>
<th>Object</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoint</td>
<td></td>
</tr>
<tr>
<td>Watchpoint (read)</td>
<td></td>
</tr>
<tr>
<td>Watchpoint (write)</td>
<td></td>
</tr>
<tr>
<td>Watchpoint (read and write)</td>
<td></td>
</tr>
</tbody>
</table>

If the breakpoint or watchpoint is for a connected target, the IDE places a check mark (✔) on the icon.

The rest of this section describes how to:

- add breakpoints
- add watchpoints
- set properties of breakpoints and watchpoints
- disable/enable breakpoints and watchpoints

### Adding breakpoints

You set breakpoints on an executable line of a program. When you debug the program, the execution suspends before that line of code executes.

To add a breakpoint:

1. In the editor area, open the file that you want to add the breakpoint to.
More debugging features

2 Notice that the left edge of the C/C++ editor has a blank space called a marker bar.

3 With your pointer, hover over the marker bar beside the exact line of code where you want to add a breakpoint. Right-click the marker bar and select Toggle Breakpoint.

A dot appears, indicating the breakpoint:

![Breakpoint in C/C++ editor](image)

A corresponding dot also appears in the Breakpoints view, along with the name of the file in which you set the breakpoint:

To add a breakpoint at the entry of a function:

▶ In either the Outline or C/C++ Projects view, right-click a function and select Toggle Breakpoint.

Adding watchpoints

To add a watchpoint:

1 Right-click in the Breakpoints view and choose the Add Watchpoint (C/C++)... item.
2 Enter an expression in the field. The expression may be anything that can be evaluated inside an if statement. (e.g. y==1)

3 If you want the program to stop when it reads the watch expression, check Read; to have the program stop when it writes the expression, check Write.

4 Click OK. The watchpoint appears in the Breakpoints view list.

**Setting properties of breakpoints and watchpoints**

After you’ve set your breakpoint or watchpoint, the IDE unconditionally halts the program when:

- it reaches a line of code that the breakpoint is set on
- the expression specified by the watchpoint becomes true.

To set the properties for a breakpoint or watchpoint:

1 In the Breakpoints view, right-click the breakpoint or watchpoint and select the **Breakpoint Properties**… item. (For breakpoints only, in the C/C++ editor, right-click the breakpoint and select **Breakpoint Properties**.)
More debugging features

2 To restrict the breakpoint to a specific threads, make sure they are selected in the Filtering panel.

3 Use the Common panel to modify the watchpoint’s behavior.
In the **Condition** field, enter the Boolean expression to evaluate. The expression may be anything that can be evaluated inside an *if* statement (e.g. \( x > y \)). The default is TRUE.

In the **Ignore Count** field, enter the number of times the breakpoint or watchpoint may be hit before it begins to take effect (not the number of times the condition is true). The default is 0.

4  Click **OK**. When in debug mode, your program stops when it meets the conditions you’ve set for the breakpoint or watchpoint.

**Disabling/enabling breakpoints and watchpoints**

You may wish to temporarily deactivate a breakpoint or watchpoint without losing the information it contains.

To disable or enable a breakpoint or watchpoint:

➤ In the Breakpoints view, right-click the breakpoint or watchpoint and select **Disable** or **Enable**. Clicking the check
More debugging features

box in the **Breakpoints** view (so the breakpoint is no longer selected) also disables the breakpoint.

For breakpoints only, right-click the breakpoint in the editor area and select **Disable Breakpoint** or **Enable Breakpoint**.

To disable or enable multiple breakpoints or watchpoints:

1. In the Breakpoints view, use any of the following methods to select the breakpoints:
   - Select breakpoints and watchpoints while holding down the Ctrl key.
   - Select a range of breakpoints and watchpoints while holding down the Shift key.
   - From the main menu, select **Edit → Select All**.
   - Right-click in the Breakpoints view and select **Select All**.

2. Right-click the highlighted breakpoints/watchpoints and select **Disable** or **Enable**.

Removing breakpoints and watchpoints

To remove one or more breakpoints/watchpoints:

- Select the breakpoint or watchpoint, right-click, then select **Remove** or **Remove All**.

Evaluating your expressions

The Expressions view lets you evaluate and examine the value of expressions:
More debugging features

The Expressions view is similar to the Variables view; for more information, see the “Inspecting variables” section in this chapter.

To evaluate an expression:

1. Right-click the Expressions view, then choose Add Watch Expression.

2. Enter the expression you want to evaluate (e.g. \((x-5) * 3\)).

3. Click OK. The expression and its value appear in the Expressions view. When the debugger suspends the program’s
More debugging features

execution, it reevaluates all expressions and highlights the changed values.

Inspecting your registers

The Registers view displays information about the registers in the currently selected stack frame. When the execution stops, the changed values are highlighted.

The Registers view is similar to the Variables view; for more information, see the “Inspecting variables” section in this chapter.

You can also customize the colors in the Registers view and change the default value of the Show Type Names option.

Inspecting a process’s memory

The Memory view lets you inspect and change your process’s memory. The view consists of four tabs that let you inspect multiple sections of memory:
More debugging features

QNX Neutrino uses a virtual-addressing model wherein each process has its own range of valid virtual addresses. This means that the address you enter into the Memory view must be a virtual address that’s valid for the process you’re debugging (e.g. the address of any variable). For more on QNX Neutrino’s memory management, see the Process Manager chapter in the *System Architecture* guide.

**Inspecting memory**

The Memory view supports the same addressing as the C language. You can address memory using expressions such as `0x0847d3c`, `(y)+1024`, and `*ptr`.

To inspect the memory of a process:

1. In the Debug view, select a process. Selecting a thread automatically selects its associated process.
2. In the Memory view, select one of the four tabs (labeled Memory 1, Memory 2, etc.).
3. In the **Address** field, type the address, then press Enter.

**Changing memory**

To change the memory of a process:

1. Follow the procedure for inspecting a process’s memory.
2. In the memory pane, type the new value for the memory. The Memory view works in “type-over” mode; use the Tab and arrow keys to jump from byte to byte.
More debugging features

The changed memory appears in red.

**CAUTION:** Changing a process’s memory can make your program crash.

Configuring output format

You can configure your output to display hexadecimal or decimal. You can also set the number of display columns and the memory unit size. You can configure each memory tab independently.

To configure the output format:

1. In the Memory view, select one of the four tabs labeled *Memory 1, Memory 2*, and so on.
2. Right-click the pane and select the item you want to configure (*Format, Memory Unit Size*, or *Number of Columns*).
3. Choose your desired format; the output reflects your selection. Note that some output formats look best in a monospaced font such as Courier.

Customizing the Memory view

You can customize the Memory view’s colors and fonts. You can also customize some of its behavior.

To access the view’s customization dialog:

1. From the menu, select *Window* → *Preferences*.
2. In the left pane, select *Run/Debug* → *Memory View*.
3. You can now change the colors, font, and behavior of the Memory view. When you’re done, click *Apply*, then *OK*. 
More debugging features

Inspecting shared-library usage

The Shared Libraries view shows you information about the shared libraries for the session you select in the Debug view. The view shows the name, start address, and end address of each library.

To load a library’s symbols:

- Right-click a library and select Load Symbols (or Load Symbols for All for all your libraries).

Monitoring signal handling

The Signals view provides a summary of how your debugger handles signals that are intercepted before they’re received by your program.

The view contains the following fields:
More debugging features

Name The name of the signal

Pass The debugger can filter out signals. If the signal is set to “no”, the debugger prevents it from reaching your program.

Suspend Upon receipt of a signal, the debugger can suspend your program as if it reached a breakpoint. Thus, you can step through your code and watch how your program handles the signal.

To change how the debugger handles a signal:

1 In the Signals view, select a signal (e.g. SIGINT) in the Name column.

2 Right-click the signal’s name, then choose Signal Properties… from the menu.

3 In the signal’s Properties dialog, check Pass this signal to the program to pass the selected signal to the program. Uncheck it to block this signal.
More debugging features

Check **Suspend the program when this signal happens** to suspend the program when it receives this signal. Uncheck it to let the program handle the signal as it normally would.

To send a signal to a suspended program:

1. In the Signals view, select a signal.

2. If the program isn’t suspended, click the **Suspend** button ( 

3. In the Signals view, right-click your desired signal and select **Resume With Signal**. Your program resumes and the debugger immediately sends the signal.

You can see a thread-by-thread summary of how your program handles signals using the Signal Information view. To learn more, see the “Mapping process signals” section in the Getting System Information chapter.

Viewing your output

The Console view shows you the output of the execution of your program and lets you supply input to your program:

![Console view](image)

The console shows three different kinds of text, each in a different default color:

- standard output (blue)
- standard error (red)
More debugging features

- standard input (green)

You can choose different colors for these kinds of text on the preferences pages.

To access the Console view’s customization dialog:

1. From the menu, select Window→Preferences.
2. In the left pane, select Run/Debug→Console.

Debugging with GDB

The IDE lets you use a subset of the commands that the `gdb` utility offers:

![GDB Output]

To learn more about the `gdb` utility, see its entry in the Utilities Reference and the Using GDB appendix of the Neutrino Programmer’s Guide.

Enabling the QNX GDB Console view

The QNX GDB Console view is part of the regular Console perspective but isn’t accessible until you toggle to it. Once you do, GDB output appears in place of the regular Console view output.

To enable the QNX GDB Console view:

1. In the Debug view, select a debug session.
2. Click the Display selected console button ( ). The Console view changes to the QNX GDB Console view.
More debugging features

Using the QNX GDB Console view

The QNX GDB Console view lets you bypass the IDE and talk directly to GDB; the IDE is unaware of anything done in the QNX GDB Console view. Items such as breakpoints that you set from the QNX GDB Console view don’t appear in the C/C++ editor.

You can’t use the Tab key for line completion because the commands are sent to GDB only when you press Enter.

To use the QNX GDB Console view:

➤ In the QNX GDB Console view, enter a command (e.g. nexti to step one instruction):
Chapter 6
Building OS and Flash Images

In this chapter...

Introducing the QNX System Builder 183
Overview of images 188
Creating a project for an OS image 199
Creating a project for a Flash filesystem image 200
Building an OS image 201
Downloading an image to your target 207
Configuring your QNX System Builder projects 213
Optimizing your system 227
Moving files between the host and target 234
Introducing the QNX System Builder

One of the more distinctive tools within the IDE is the QNX System Builder perspective, which simplifies the job of building OS images for your embedded systems. Besides generating images intended for your target board’s RAM or Flash, the QNX System Builder can also help reduce the size of your images (e.g. by reducing the size of shared libraries). The Builder also takes care of tracking library dependencies for you, prompting you for any missing components.

The QNX System Builder contains a Serial Terminal view for interacting with your board’s ROM monitor or QNX Initial Program Loader (IPL) and for transferring images (using the QNX sendnto protocol). The QNX System Builder also has an integrated TFTP Server that lets you transfer your images to network-aware targets that can boot via the TFTP protocol.

When you open the QNX System Builder to create a project, you have the choice of importing/customizing an existing buildfile to generate an image or of creating one from scratch. The QNX System Builder editor lets you select which components (binaries, DLLs, libs) you want to incorporate into your system image. As you add a component,
Introducing the QNX System Builder

the QNX System Builder automatically adds any shared libraries required for runtime loading. For example, if you add the telnet application to a project, then the QNX System Builder knows to add libsocket.so in order to ensure that telnet can run. And when you select a binary, you’ll see relevant information for that item, including its usage message, in the Binary Inspector view.

Using standard QNX embedding utilities (mkifs, mkefs), the QNX System Builder can generate configuration files for these tools that can be used outside of the IDE for scripted/automated system building. As you do a build, a Console view displays the output from the underlying build command. You can use the mksbp utility to build a QNX System Builder project.bld from the command-line; mksbp automatically calls mkifs or mkefs, depending on the kind of image being built.

Here’s what the QNX System Builder perspective looks like:

One of the main areas in the QNX System Builder is the editor, which presents two panes side by side:
Introducing the QNX System Builder

Images
Shows all the images you’re building. You can add or remove binaries and other components, view their properties, etc.

Filesystem
Shows the components of your image arranged in a hierarchy, as they would appear in a filesystem on your target.

Toolbar buttons
Above the Images and Filesystem panes in the editor you’ll find several buttons for working with your image:

- Add a new binary.
- Add a new shared library.
- Add a new DLL.
Introducing the QNX System Builder

- Add a new symbolic link.
- Add a new file.
- Add a new directory.
- Add a new image.
- Run the System Optimizer.
- Rebuild the current project.
- Merge two or more images into a single image.

**Binary Inspector**

Below the Images and Filesystem panes is the QNX Binary Inspector view, which shows the usage message for any binary you select:

![Binary Inspector](image)

The Binary Inspector also has a **Use Info** tab that gives the selected binary’s name, a brief description, the date it was built, and so on.
Introducing the QNX System Builder

Boot script files

All QNX BSPs ship with a buildfile, which is a type of “control” file that gives instructions to the mkifs command-line utility to generate an OS image. The buildfile specifies the particular startup program, environment variables, drivers, etc. to use for creating the image. The boot script portion of a buildfile contains the sequence of commands that the Process Manager executes when your completed image starts up on the target.

For details on the components and grammar of buildfiles, see the section “Configuring an OS image” in the chapter Making an OS Image in Building Embedded Systems as well as the entry for mkifs in the Utilities Reference.

The QNX System Builder perspective provides a convenient graphical alternative to the text-based buildfile method. While it hides most of the “gruesome” details from you, the QNX System Builder perspective also lets you see and work with things like boot scripts.

The QNX System Builder perspective stores the boot script for your project in a .bsh file.

If you double-click a .bsh file in the Navigator or System Builder Projects view, you’ll see its contents in the editor.

QNX System Builder projects

Like other tools within the IDE, the QNX System Builder perspective is project-oriented — it organizes your resources into a project of related items. Whenever you create a project in the QNX System Builder perspective, you’ll see a project.bld file in the Navigator or System Builder Projects view.

The project.bld file drives the System Builder editor; if you select the project.bld, you’ll see your project’s components in the Images and Filesystem panes, where you can add/remove items for the image you’ll be building.
As with most other tools in the IDE, you build your QNX System Builder projects using the standard Eclipse build mechanism via Project→Build Project.

The scope of the QNX System Builder

You can use the QNX System Builder throughout your product-development cycle:

- At the outset — to import a QNX BSP, generate a minimal OS image, and transfer the image to your board, just to make sure everything works.
- During development — to create an image along with your suite of programs and download everything to your eval board.
- For your final product — to strip out the extra utilities you needed during development, reduce libraries to their bare minimum size, and produce your final, customized, optimized embedded system.

For details on importing a BSP, see the section “Importing a BSP or other QNX source packages” in the chapter Managing Source Code in this guide.

Overview of images

Before you use the QNX System Builder to create OS and Flash images for your hardware, let’s briefly describe the concepts involved in building images so you can better understand the QNX System Builder in context.

This section covers the following topics:

- The components of an image, in order of booting
- Types of images you can create
- Project layout
Overview of workflow

The components of an image, in order of booting

Neutrino supports a wide variety of CPUs and hardware configurations. Some boards require more effort than others to embed the OS. For example, x86-based machines usually have a BIOS, which greatly simplifies your job, while other platforms require that you create a complete IPL. Embedded systems can range from a tiny memory-constrained handheld computer that boots from Flash to an industrial robot that boots through a network to an SMP system with lots of memory that boots from a hard disk.

Whatever your particular platform or configuration, the QNX System Builder helps simplify the process of building images and transferring them from your host to your target.

For a complete description of OS and Flash images, see the Building Embedded Systems guide.

The goal of the boot process is to get the system into a state that lets your program run. Initially, the system might not recognize disks, memory, or other hardware, so each section of code needs to perform whatever setup is needed in order to run the subsequent section:

1. The IPL initializes the hardware, makes the OS image accessible, and then jumps into it.
2. The startup code performs further initializations then loads and transfers control to the microkernel/process manager (procundo), the core runtime component of the QNX Neutrino OS.
3. The procunto module then runs the boot script, which performs any final setup required and runs your programs.
Typical boot order.

At reset, a typical processor has only a minimal configuration that lets code be executed from a known linearly addressable device (e.g. Flash, ROM). When your system first powers on, it automatically runs the IPL code at a specific address called the reset vector.

**IPL**

When the IPL loads, the system memory usually isn’t fully accessible. It’s up to the IPL to configure the memory controller, but the method depends on the hardware — some boards need more initialization than others.

When the memory is accessible, the IPL scans the Flash memory for the image filesystem, which contains the startup code (described in the next section). The IPL loads the startup header and startup code into RAM, and then jumps to the startup code.

The IPL is usually board-specific (it contains some assembly code) and as small as possible.

**Startup**

The startup code initializes the hardware by setting up interrupt controllers, cache controllers, and base timers. The code detects system resources such as the processor(s), and puts information about these resources into a centrally accessible area called the system page. The code can also copy and decompress the image filesystem components, if necessary. Finally, the startup code passes control, in virtual memory mode, to the procnto module.
The startup code is board-specific and is generally much larger than the IPL. Although a larger procnto module could do the setup, we separate the startup code so that procnto can be board-independent. Once the startup code sets up the hardware, the system can reuse a part of the memory used by startup because the code won’t be needed again.

**The procnto module**

The procnto module is the core runtime component of the QNX Neutrino OS. It consists of the microkernel, the process manager, and some initialization code that sets up the microkernel and creates the process-manager threads. The procnto module is a required component of all bootable images.

The process manager handles (among other things) processes, memory, and the image filesystem. The process manager lets other processes see the image filesystem’s contents. Once the procnto module is running, the operating system is essentially up and running. One of the process manager’s threads runs the boot script.

Several variants of procnto are available (e.g. procnto-400 for PowerPC 400 series, procnto-smp for x86 SMP machines, etc.).

**Boot script**

If you want your system to load any drivers or to run your program automatically after power up, you should run those utilities and programs from the boot script. For example, you might have the boot script run a devf driver to access a Flash filesystem image and then run your program from that Flash filesystem.

When you build your image, the boot script is converted from text to a tokenized form and saved as /proc/boot/ .script. The process manager runs this tokenized script.

**Types of images you can create**

The IDE lets you create the following images:
Overview of images

OS image (.ifs file)

An image filesystem. A bootable image filesystem holds the procnto module, your boot script, and possibly other components such as drivers and shared objects.

Flash image (.efs file)

A Flash filesystem. (The “e” stands for “embedded.”) You can use your Flash memory like a hard disk to store programs and data.

Combined image

An image created by joining together any combination of components (IPL, OS image, embedded filesystem image) into a single image. You might want to combine an IPL with an OS image, for example, and then download that single image to the board’s memory via a ROM monitor, which you could use to burn the image into Flash. A combined image’s filename extension indicates the file’s format (e.g. .elf, .srec, etc.).

If you plan on debugging applications on the target, you must include pdebug in /usr/bin. If the target has no other forms of storage, include it in the OS image or Flash image.

BSP filename conventions

In our BSP docs, buildfiles, and scripts, we use a certain filename convention that relies on a name’s prefixes and suffixes to distinguish types:

<table>
<thead>
<tr>
<th>Part of filename</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>.bin</td>
<td>Suffix for binary format file.</td>
<td>ifs-artesyn.bin</td>
</tr>
<tr>
<td>.build</td>
<td>Suffix for buildfile.</td>
<td>sandpoint.build</td>
</tr>
</tbody>
</table>

continued...
### Overview of images

<table>
<thead>
<tr>
<th>Part of filename</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>efs-</td>
<td>Prefix for QNX Embedded Filesystem file; generated by mkefs.</td>
<td>efs-sengine.srec</td>
</tr>
<tr>
<td>.elf</td>
<td>Suffix for ELF (Executable and Linking Format) file.</td>
<td>ipl-ifs-mbx800.elf</td>
</tr>
<tr>
<td>ifs-</td>
<td>Prefix for QNX Image Filesystem file; generated by mkifs.</td>
<td>ifs-800fads.elf</td>
</tr>
<tr>
<td>ipl-</td>
<td>Prefix for IPL (Initial Program Loader) file.</td>
<td>ipl-eagle.srec</td>
</tr>
<tr>
<td>.openbios</td>
<td>Suffix for OpenBIOS format file.</td>
<td>ifs-walnut.openbios</td>
</tr>
<tr>
<td>.prepboot</td>
<td>Suffix for Motorola PRePboot format file.</td>
<td>ifs-prpmc800.prepboot</td>
</tr>
<tr>
<td>.srec</td>
<td>Suffix for S-record format file.</td>
<td>ifs-malta.srec</td>
</tr>
</tbody>
</table>

The QNX System Builder uses a somewhat simplified convention. Only a file’s three-letter extension, not its prefix or any other part of the name, determines how the QNX System Builder should handle the file.

For example, an OS image file is always a .ifs file in the QNX System Builder, regardless of its format (ELF, binary, SREC, etc.). To determine a file’s format in the IDE, you’ll need to view the file in an editor.

### OS image (.ifs file)

The OS image is a bootable image filesystem that contains the startup header, startup code, procnto, your boot script, and any drivers needed to minimally configure the operating system:
Generally, we recommend that you keep your OS image as small as possible to realize the following benefits:

- Memory conservation — When the system boots, the entire OS image gets loaded into RAM. This image isn’t unloaded from RAM, so extra programs and data built into the image require more memory than if your system loaded and unloaded them dynamically.

- Faster boot time — Loading a large OS image into RAM can take longer to boot the system, especially if the image must be loaded via a network or serial connection.

- Stability — Having a small OS image provides a more stable boot process. The fewer components you have in your OS image, the lower the probability that it fails to boot. The components that must go in your image (startup, procnto, a Flash driver or network components, and a few shared objects) change rarely, so they’re less subject to errors introduced during the development and maintenance cycles.

If your embedded system has a hard drive or CompactFlash (which behaves like an IDE hard drive), you can access the data on it by including a block-oriented filesystem driver (e.g. devb-eide) in your OS image filesystem and calling the driver from your boot script. For details on the driver, see devb-eide in the Utilities Reference.

If your system has an onboard Flash device, you can use it to store your OS image and even boot the system directly from Flash (if your
board allows this — check your hardware documentation). Note that an OS image is read-only; if you want to use the Flash for read/write storage, you’ll need to create a Flash filesystem image (.efs file).

**Flash filesystem image (.efs file)**

Flash filesystem images are useful for storing your programs, extra data, and any other utilities (e.g. qconn, ls, dumper, and pidin) that you want to access on your embedded system.

If your system has a Flash filesystem image, you should include a devf* driver in your OS image and start the driver in your boot script. While you can mount an image filesystem only at /, you can specify your own mountpoint (e.g. /myFlashStuff) when you set up your .efs image in the IDE. The system recognizes both the .ifs and .efs filesystems simultaneously because the process manager transparently overlays them. To learn more about filesystems, see the Filesystem chapter in the QNX Neutrino System Architecture guide.

**Combined image**

For convenience, the IDE can join together any combination of your IPL, OS image, and .efs files into a single, larger image that you can transfer to your target:

When you create a combined image, you specify the IPL’s path and filename on your host machine. You can either select a precompiled IPL from among the many BSPs that come with QNX Momentics PE, or compile your own IPL from your own assembler and C source.
Overview of images

The QNX System Builder expects the source IPL to be in ELF format.

Padding separates the IPL, .ifs, and .efs files in the combined image.

Padding after the IPL

The IPL can scan the entire combined image for the presence of the startup header, but this slows the boot process. Instead, you can have the IPL scan through a range of only two addresses and place the startup header at the first address.

Specifying a final IPL size that’s larger than the actual IPL lets you modify the IPL (and change its length) without having to modify the scanning addresses with each change. This way, the starting address of the OS image is independent of the IPL size.

You must specify a padding size greater than the total size of the IPL to prevent the rest of the data in the combined image file from partially overwriting your IPL.

Padding before .ifs images

If your combined image includes one or more .efs images, specify an alignment equal to the block size of your system’s onboard Flash. The optimized design of the Flash filesystem driver requires that all .efs images begin at a block boundary. When you build your combined image, the IDE adds padding to align the beginning of the .efs image(s) with the address of the next block boundary.

Project layout

A single QNX System Builder project can contain your .ifs file and multiple .efs files, as well as your startup code and boot script. You can import the IPL from another location or you can store it inside the project directory.
Overview of images

By default, your QNX System Builder project includes the following parts:

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Images directory</strong></td>
<td>The images and generated files that the IDE creates when you build your project.</td>
</tr>
<tr>
<td><strong>Overrides directory</strong></td>
<td>When you build your project, the IDE first looks in this directory for a directory matching the image being built. Any files in that directory are used to satisfy the build requirements before searching the standard locations. You can use the Overrides/image name directory to easily test a change to your build. You must create the image name subdirectory yourself and populate it with the override files your image needs.</td>
</tr>
<tr>
<td><strong>Reductions directory</strong></td>
<td>The IDE lets you reduce your image size by eliminating unused libraries, and shrinking the remaining libraries. The IDE stores the reduced libraries in the Reductions/image name directory (where image name is the name of the image being built).</td>
</tr>
<tr>
<td><strong>.project file</strong></td>
<td>Information about the project, such as its name and type. All IDE projects have a .project file.</td>
</tr>
</tbody>
</table>
Overview of images

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.sysbldr_meta</td>
<td>Information about the properties specific to a QNX System Builder project. This file describes where the IDE looks for files (including the Overrides and Reductions directories), the location of your IPL file, how the IDE includes .efs files, and the final format of your .ifs file.</td>
</tr>
<tr>
<td>project.bld</td>
<td>Information about the structure and contents of your .ifs and .efs files. This file also contains your boot script file.</td>
</tr>
<tr>
<td>.bsh</td>
<td>Contains the boot script for your project.</td>
</tr>
</tbody>
</table>

Overview of workflow

Here are the main steps involved in using the IDE to get Neutrino up and running on your board:

- Creating a QNX System Builder project for an OS or a Flash image for your board. The process is very simple if a BSP exists for your board. If an exact match isn’t available, you may be able to modify an existing BSP to meet your needs.

- Building your project to create the image.

- Transferring the OS image to your board. You might do this initially to verify that the OS image runs on your hardware, and then again (and again) as you optimize your system.

- Configuring your projects.

- Optimizing your system by reducing the size of the libraries.
Creating a project for an OS image

To create a new QNX System Builder Project:

1. From the main menu, select File → New → Project.

2. Expand QNX, then select QNX System Builder Project. Click Next.

3. Name your project and click Next.

4. At this point, you can either import an existing buildfile (as shipped with your QNX BSPs) or select a generic type (e.g. “Generic PPCBE”).

Chapter 6 ● Building OS and Flash Images 199
Creating a project for a Flash filesystem image

We recommend that you select **Import Existing Buildfile**, rather than a generic option. Creating a buildfile requires a working knowledge of
*boot script grammar* (as described in the entry for `mkifs` in the
*Utility Reference* and in the *Building Embedded Systems* manual).

Click the **Browse** button to select an existing buildfile. Refer
to your BSP docs for the proper `.build` file for your board.
You can find buildfiles for all the BSPs installed on your system
in `C:/QNX630/target/qnx6/processor/boot/build/` on
your Windows host. (For Neutrino, Linux, or Solaris hosts, see
the appendix Where Files Are Stored in this guide.)

If you’re creating a generic buildfile, select your desired
platform from the drop-down list.

5 Click **Finish**. The IDE creates your new project, which
includes all the components that make up the OS image.

Creating a project for a Flash filesystem image

To create a Flash filesystem project:

1 From the main menu, select **File**→**New**→**Project**.

2 Expand **QNX**, then select **QNX System Builder EFS Project**
in the right. Click **Next**.

3 Name your project and click **Next**.

4 Specify your target hardware (e.g. “Generic ARMLE”).
5 Click Finish. The IDE creates your new EFS project, which includes a “generic”.efs image; you’ll likely need to specify the block size, image size, and other properties to suit the Flash on your particular board.

Building an OS image

You build your QNX System Builder projects using the standard Eclipse build mechanism:

➢ From the main menu, select Project→Build | Build Project.

You can also build projects using the context menu:

1 In the Navigator or System Builder Projects view, right-click the project.
Building an OS image

2 Select build.

The System Builder Console view shows the output produced when you build your images:

Output can come from any of these utilities:

- \texttt{mkefs}
- \texttt{mkifs}
- \texttt{mkimage}
- \texttt{mkrec}
- \texttt{objcopy}

For more information, see their entries in the \textit{Utilities Reference}.

You can clear the view by clicking the \textbf{Clear Output} button (\texttt{\textcolor{red}{\textbullet}}).

Create new image

You can create a new image for your QNX System Builder project by using the \textbf{Add New Image} icon (\texttt{\textcolor{blue}{\textbullet}}) in the toolbar:

1 Click the \textbf{Add New Image} icon in the toolbar.

The IDE displays the Create New Image dialog box:
Use the Create New Image dialog to:

- **Duplicate Selected Image** — create a duplicate of the currently selected image with the given name.
- **Import Existing Buildfile** — generate the new image using an existing build file.
- **Create Generic IFS image** — create an empty IFS for the specified platform.
- **Create Generic EFS image** — create an empty EFS for the specified platform.

3 Click **OK** to create the new image and add it to your project.
Combine images

As mentioned earlier, the QNX System Builder lets you create a combined image. You use the Combine images icon (iotic) to:

- add an IPL to the start of your image
- append an EFS to your image
- set the final format of your image

Adding an IPL to the start of your image

To add an IPL to the start of your image:

1. In the Image view, select your image.
2. Click the Combine image(s) icon (iotic).
3. In the Create New Image dialog box, check Add IPL.
4 Enter the IPL filename (or select it by clicking the browse icon).

5 In the **Pad IPL to:** field, select padding equal to or greater than the size of your IPL.

**CAUTION:** If the padding is less than the size of the IPL, the image won’t contain the complete IPL.

6 Click **OK**.
Building an OS image

If you get a File Not Found error while building, make sure the Build with profiling option is unchecked in all of the C/C++ projects in the BSP working set, then rebuild all of the projects.

Right-click on a project, then choose Properties and select QNX C/C++ Project to view the Build with profiling setting.

Adding an EFS to your image

To append a Flash filesystem to your image:

1. In the Append EFS section of the Create New Image dialog, check Append EFS.
2. In the Alignment field, select the granularity of the padding. The padding is a multiple of your selected alignment.
3. Click OK.

Setting the final format of your OS image

You use the Final Processing section of the Create New Image dialog to set the final format for your image.

To change the final format of your OS image:

1. In the Final Processing section of the Create New Image dialog, check the Final Processing box.
2. In the Offset field, enter the board-specific offset. This setting is generally used for S-Record images.
3. In the Format field, select the format from the dropdown menu (e.g. SREC, Intel hex records, binary.)
4. Click OK.

For more information of the final processing of an OS image, see mkrec in the Utilities Reference.
Downloading an image to your target

Many boards have a ROM monitor, a simple program that runs when you first power on the board. The ROM monitor lets you communicate with your board via a command-line interface (over a serial or Ethernet link), download images to the board’s system memory, burn images into Flash, etc.

The QNX System Builder has two facilities you can use to communicate with your board:

- serial terminals (up to four)
- TFTP server

If your board doesn’t have a ROM monitor, you probably can’t use the download services in the IDE; you’ll have to get the image onto the board some other way (e.g. JTAG). To learn how to connect to your particular target, consult your hardware and BSP documentation.

Downloading via a serial link

With the QNX System Builder’s built-in serial terminals, you don’t need to leave the IDE and open a serial communications program (e.g. HyperTerminal) in order to talk to your target, download images, etc.

The Terminal view implements a very basic serial terminal, supporting only the following control sequences: 0x00 (NUL), 0x07 (bell), 0x08 (backspace), 0x09 (horizontal tab), 0x0a (newline), and 0x0d (carriage return).

To open a terminal:

▷ From the main menu, select Show View→Other…, then select QNX System Builder→Terminal N (where N is 1 to 4).

The Terminal view lets you set standard communications parameters (baud rate, parity, data bits, stop bits, and flow control), choose a port (COM1 or COM2), send a BREAK command, and so on.
Downloading an image to your target

To communicate with your target over a serial connection:

1. Connect your target and host machine with a serial cable.

2. Specify the device (e.g. COM 2) and the communications settings in the view’s menu:

   ![Menu With Options]

   - Speed
   - Data Bits
   - Stop Bits
   - Parity
   - Flow Control

You can now interact with your target by typing in the view.
Under Solaris, the Terminal view’s **Device** menu may list all available devices twice:

![Device menu list]

and display the following message on the console used to launch **qde**:

```
#unexpected error in javax.comm native code
Please report this bug
SolarisSerial.c, cond_wait(), rv=1 ,errno=0
```

This is a known problem, and can be safely ignored.

When a connection is made, the **Send File** button changes from its disabled state (.disabled) to its enabled state (enabled), indicating that you can now transfer files to the target.

To transfer a file using the Serial Terminal view:

1. Using either the Serial Terminal view or another method (outside the IDE), configure your target so that it’s ready to receive an image. For details, consult your hardware documentation.

2. In the Serial Terminal view, click the **Send File** button (enabled).
Downloading an image to your target

3 In the Select File to Send dialog, enter the name of your file (or click Browse).

4 Select a protocol (e.g. sendto).

The QNX sendto protocol sends a sequence of records (including the start record, data records, and a go record). Each record has a sequence number and a checksum. Your target must be running an IPL (or other software) that understands this protocol.

5 Click OK. The QNX System Builder transmits your file over the serial connection.

You can click the Cancel button to stop the file transfer:

---

Downloading via TFTP

The QNX System Builder’s TFTP server eliminates the need to set up an external server for downloading images (if your target device supports TFTP downloads). The TFTP server knows about all QNX System Builder projects in the system and automatically searches them for system images whenever it receives requests for service.

When you first open the TFTP Server view (in any perspective), the QNX System Builder starts its internal TFTP server. For the remainder of the current IDE session, the TFTP server listens for incoming TFTP transfer requests and automatically fulfill them.

Currently, the QNX System Builder’s internal server supports only TFTP read requests; you can’t use the server to write files from the target to the host development machine.

The TFTP Server view provides status and feedback for current and past TFTP transfers. As the internal TFTP server handles requests, the view provides visual feedback:
Downloading an image to your target

Each entry in the view shows:

- TFTP client IP address/hostname
- requested filename
- transfer progress bar
- transfer status message

Transferring a file

To transfer a file using the TFTP Server view:

1. Open the TFTP Server view. The internal TFTP server starts.

2. Using either the QNX System Builder’s serial terminal or another method, configure your target to request a file recognized by the TFTP server. (The TFTP Server view displays your host’s IP address.) During the transfer, the view shows your target’s IP address, the requested file, and the transfer status.

You can clear the TFTP Server view of all completed transactions by clicking its clear button ( ).
Downloading an image to your target

The internal TFTP server recognizes files in the **Images** directory of all open QNX System Builder projects; you don’t need to specify the full path.

### Transferring files that aren’t in Images

**CAUTION:** The IDE deletes the content of the **Images** directory during builds — don’t use this directory to transfer files that the QNX System Builder didn’t generate. Instead, configure a new path, as described in the following procedure.

To enable the transfer of files that aren’t in the **Images** directory:

1. From the main menu, select **Window→Preferences**.

2. In the left pane of the Preferences dialog, select **QNX→Connections→TFTP Server**.

3. Check **Other Locations**.
Configuring your QNX System Builder projects

4 Click Add Path, and then select your directory from the Browse For Folder dialog.

5 Click Apply.

6 Click OK. The TFTP server is now aware of the contents of your selected directory.

Downloading using other methods

If your board doesn’t have an integrated ROM monitor, you may not be able to transfer your image over a serial or TFTP connection. You’ll have to use some other method instead, such as:

- CompactFlash — copy the image to a CompactFlash card plugged into your host, then plug the card into your board to access the image.
  Or:

- Flash programmer — manually program your Flash with an external programmer.
  Or:

- JTAG/ICE/emulator — use such a device to program and communicate with your board.

For more information, see the documentation that came with your board.

Configuring your QNX System Builder projects

In order to use the QNX System Builder to produce your final embedded system, you’ll likely need to:

- add/remove image items

- configure the properties of an image and its items

- configure the properties of the project itself
As mentioned earlier, every QNX System Builder project has a `project.bld` file that contains information about your image’s boot script, all the files in your image, and the properties of those files.

If you double-click the `project.bld`, you’ll see your project’s components in the Images and Filesystem panes in the editor area, as well as a Properties view:

![Image of QNX System Builder interface]

### Managing your images

The Images pane shows a tree of all the files in your image, sorted by type:

- binaries
- shared libraries
- symbolic links
- DLLs
- other files
- directories

### Adding files to your image

When you add files, you can either browse your host filesystem or select one or more files from a list of search paths:

**Browse method**  
If you choose files by browsing, you’ll probably want to configure the project to use an *absolute path* so that the IDE always finds the exact file you
Configuring your QNX System Builder projects

specified (provided you keep the file in the same location). Note that other users of your project would also have to reproduce your setup in order for the IDE to locate files.

Select method Select files from a preconfigured list of search locations. We recommend that you use this option because it’s more flexible and lets others easily reproduce your project on their hosts. You can add search paths to the list.

Note that the IDE saves only the filename. When you build your project, the IDE follows your search paths and uses the first file that matches your specified filename. If you specify a file that isn’t in the search path, the build will be incomplete. To learn how to configure your search paths, see the section “Configuring project properties” in this chapter.

To add items to your image:

1. In the Images pane, right-click the image and select Add Item, followed by the type of item you want to add:
   - Binary
   - Shared Library
   - DLL
   - Symbolic Link
   - File
   - Directory
   - Image
Configuring your QNX System Builder projects

If you select **Image**, the QNX System Builder displays the Create New Image dialog.

1. Select an item (e.g. **Binary**) from the list.

2. Select either the **Search using the project's search paths** or the **Use selected absolute path(s)** option. (We recommend using search paths, because other users would be able to recreate your project more easily.)

3. Click **OK**. The QNX System Builder adds the item to your image, as you can see in the **Images** pane.

### Deleting files

- In either the Images or Filesystem pane, right-click your file and select **Delete**.

### Adding directories

To add a directory to your image:

1. In the Filesystem pane, right-click the parent directory and select **Add Item→Directory**.

2. Specify the directory name, path, and image name. Some fields are filled in automatically.

3. Click **Finish**. Your directory appears in the Filesystem pane.

You can also add a directory by specifying the path for an item in the **Location In Image** field in the Properties view. The IDE includes the specified directory as long as the item remains in your image.

### Deleting directories

- In either the Images or Filesystem pane, right-click your directory and select **Delete**.
Configuring your QNX System Builder projects

A deleted directory persists if it still contains items. To completely remove the directory, delete the reference to the directory in the Location In Image field in the Properties view for all the items in the directory.

Configuring image properties

The Properties view lets you see and edit the properties of an image or any of its items:

![Properties view screenshot]

To change the properties of an image or item:

1. In the Images or Filesystem pane, select an image or one of its items.
2. In the Properties view, select an entry in the Value column. The value is highlighted; for some fields (e.g. CPU Type), a dropdown menu appears.
3. Type a new value or select one from the dropdown menu.
4. Press Enter.
5. Save your changes.
6  Rebuild your project.

Different properties appear for images and for the items in an image:

- Image properties
  - Combine
  - Directories
  - General
  - System (.ifs)
  - System (.efs)

- Item properties
  - General
  - Memory
  - Permissions

**Image properties**

**Combine**

These settings control how images are combined with your System Builder project. For example, you can control how the EFS is aligned, what format the resulting image is, the location of the IPL, its image offset, and whether or not the IPL is padded to a certain size or not.

**Directories**

These settings control the default permissions for directories that you add to the image, as well as for any directories that the tools create when you build your system. For example, if you add `/usr/bin/ksh` to your system, the IDE automatically creates the `usr` and `bin` directories. (For more information on permissions, see the Managing User Accounts chapter in the Neutrino User’s Guide and the `chmod` entry in the Utilities Reference.)

Note that the values for permissions are given in *octal* (e.g. 777, which means the read, write, and execute bits are set for the user, group, and other categories).
Configuring your QNX System Builder projects

General

**Boot Script (.ifs only)**
Name of the file that contains the boot script portion of a buildfile. Boot script files have a `.bsh` extension (e.g. `prpmc800.bsh`).

**Compressed (.ifs only)**
If set to something other than `No`, the QNX System Builder compresses the directory structure (image filesystem) section of the image. The directory structure includes `procnto`, the boot script, and files. You might enable compression if you want to save on Flash space or if the BIOS/ROM monitor limits the size of your image.

**CPU Type**
Your target’s processor (e.g. `armle`).

**Create Image**
If `Yes`, the IDE builds this image when you build your project.

**Image Name**
Name of the `.ifs` file saved in the `Images` directory during a build.

**Page Align Image?**
If `Yes`, files in the image are aligned on page boundaries.

**Remove File Time Stamps?**
If `Yes`, file timestamps are replaced by the current date/time.

**System (.ifs)**

**Auto Link Shared Libs?**
If `Yes`, shared libraries referenced by the image’s binaries are automatically included in the image.

**Boot File**
The image filter that the QNX System Builder uses (e.g. `srec`, `elf`) to perform further processing on the
image file. For example, srec converts the image to the Motorola S-Record format. (For more about image filters, see mkifs in the Utilities Reference.)

**Image Address**

The base address where the image is copied to at boot time. For XIP, set this to the same location as your image file on Flash memory and specify the read/write memory address with the **RAM Address** value, described below.

**Procnto**

Which procnto binary to use (e.g. procnto-600, procnto-600-SMP, etc.).

**Procnto/Startup Symbol Files?**

If **Yes**, include debugging symbol files for procnto and the system startup code.

**Procnto $LD_LIBRARY_PATH**

Path(s) where procnto should look for shared libraries. Separate the paths with a colon (:).

**Procnto $PATH**

Path(s) where procnto should look for executables. Separate the paths with a colon (:).

**Procnto Arguments**

Command-line arguments for procnto.

**RAM Address**

The location of your read/write memory. For XIP, set the address; otherwise, set the value to **Default**. (Note that RAM Address is the *ram* attribute in the mkifs utility.)

**Startup**

Which startup binary to use (e.g. startup-bios, startup-rpx-lite, etc.).
Configuring your QNX System Builder projects

**Startup Arguments**
Command-line arguments for the startup program.

**System (.efs)**
These settings control the format and size of your Flash filesystem image. Unless otherwise specified, the values are in bytes, but you can use the suffixes K, M, or G (e.g. 800, 16K, 2M, 4G). The IDE immediately rejects invalid entries.

**Block Size**
The size of the blocks on your Flash.

**Filesystem Type**
The type of Flash filesystem to create. Use the default (ffs3) unless you specifically need compatibility with older software that requires ffs2 format images.

**Filter**
The filter to use with this image, usually `flashcmp`. (The `mkefs` utility calls `flashcmp`.) You can use any valid command-line argument, such as `flashcmp -t zip`.

**Image Mount Point**
The path where the filesystem is mounted in the filesystem. By default, the location is `/`.

**Maximum Image Size**
The limit for the size of the generated image. If the image exceeds the maximum size, `mkefs` fails and reports an error in the System Builder Console view. The default setting of 4GB accommodates most images.

**Minimum Image Size**
The minimum size of the embedded filesystem. If the size of the filesystem is less than this size after all the specified files have been added, then the...
filesystem is padded to the required size. The default is 0 (i.e. no padding occurs).

Spare Blocks The number of spare blocks to be used by the embedded filesystem. If you want the embedded filesystem to be able to reclaim the space taken up by deleted files, set the number of spare blocks to 1 or more. The default is 1.

Item properties

General

Absolute Location
The offset in the image for this item’s data, in bytes.

Filename
The name of the file for this item (e.g. devc-ser8250).

Image Name
The name of the image in which this item resides.

Include In Image
If Yes, the QNX System Builder includes this item when it builds the image.

Location In Image
The directory where the item lives. If you change this setting, the directory location shown in the Filesystem pane changes as well.

Symbolic links also have a Linked To field for the source file.

Optional Item?
If Yes, this item is considered optional. It’s excluded from the image if the image is too large to fit in the architecture-specific maximum image size.

Strip File
By default, mkifs strips usage messages, debugging information, and Photon resources from
Configuring your QNX System Builder projects

executable files that you include in the image. Doing this helps reduce the size of the image. To keep this information, select No. See mkifs (especially the +raw attribute) and strip in the Utilities Reference.

Set this field to No if your image includes PhAB applications.

Memory

Use these two settings (which apply to .ifs files only) to specify whether a program’s code and data segments should be used directly from the image filesystem (Use In Place) or copied when invoked (Copy). For more information, see the code attribute in the mkifs documentation.

Code Segment

Copy this item’s code segment into main memory, or Use In Place to run the executable directly from the image.

Data Segment

Copy this item’s data segment into main memory, or Use In Place to use it directly from the image.

Permissions

Use these settings to specify the read/write/execute permissions (in octal) assigned to each item, as well as the item’s group and user IDs.

Configuring project properties

The Properties dialog for your QNX System Builder project (right-click the project, then select Properties) lets you view and change the overall properties of your project. For example, you can add dependent projects and configure search paths.

The dialog includes the following sections:

- Info
Configuring your QNX System Builder projects

- External Tools Builders
- Project Preferences
- Search Paths

For information on external tools, follow these links in the Eclipse Workbench User Guide: Tasks→Building resources→Running external tools.

Search Paths

The **Search Paths** pane lets you configure where the IDE looks for the files you specified in your `project.bld` file:

The IDE provides separately configurable search paths for:

- binaries
- shared libraries
- DLLs
- other files
• system files

To add a search path:

1 In the Navigator or System Builder Projects view, right-click your project and select **Properties**.

2 In the left pane, select **Search Paths**.

3 In the right pane, select one of the following tabs:
   - Binaries
   - Shared Libraries
   - DLLs
   - Other Files
   - System Files

4 Click one of the following buttons:
   - **Add Absolute Path** — a hard-coded path
   - **Add QNX_TARGET Path** — a path with a $QNX_TARGET prefix
   - **Add Workspace Path** — a path with a $WORKSPACE prefix
   - **Add Project** — a path with a $WORKSPACE/projectName prefix

The Browse For Folder or Search Projects dialog appears.

5 Select your path or project and click **OK**. The IDE adds your path to the end of the list.

To manage your search paths:

1 In the **Search Path** section of the Properties dialog, select one of the following tabs:
   - Binaries
   - Shared Libraries
Configuring your QNX System Builder projects

- DLLs
- Other Files
- System Files

2 Select a path, then click one of these buttons:
- Move Up
- Move Down
- Remove Selected

CAUTION: The **Overrides** directory must be first in the list. The **Reductions**/**image_name** directory, which is listed in the **Shared Libraries** tab, must be second in the list. Changing the order of the **Overrides** or **Reductions** directories may cause unexpected behavior.

3 Click OK.

Search path variables

You can use any of the following environment variables in your search paths; these are replaced by their values during processing:

- QNX_TARGET
- QNX_TARGET_CPU
- WORKSPACE
- PROJECT
- CPU
- CPUDIR
Optimizing your system

Since “less is better” is the rule of thumb for embedded systems, the QNX System Builder’s System Optimizer and the Dietician help you optimize your final system by:

- reducing the size of shared libraries for your image
- performing system-wide optimizations to remove unnecessary shared libraries, add required shared libraries, and reduce the size of all shared libraries in the system

CAUTION: If you reduce a shared library, and your image subsequently needs to access binaries on a filesystem (disk, network, etc.) that isn’t managed by the QNX System Builder, then the functions required by those unmanaged binaries may not be present. This causes those binaries to fail on execution.

In general, shared-library optimizers such as the Dietician are truly useful only in the case of a finite number of users of the shared libraries, as you would find in a closed system (i.e. a typical embedded system).

If you have only a small number of unmanaged binaries, one workaround is to create a dummy Flash filesystem image and add to this image the binaries you need to access. This dummy image is built with the rest of the images, but it can be ignored. This technique lets the Dietician be aware of the requirements of your runtime system.

Optimizing all libraries in your image

To optimize all the libraries in an image:

1. In the Navigator or System Builder Projects view, double-click your project’s project.bld file.

2. In the toolbar, click the Optimize System button ( ).

3. In the System Optimizer, select the optimizations that you want to make:
Optimizing your system

Remove unused libraries
When you select this option, the Dietician inspects your entire builder project and ensures that all shared libraries in the system are required for proper operation. If the QNX System Builder finds libraries that no component in your project actually needs, you’ll be prompted to remove those libraries from your project.

Add missing libraries
This option causes the Dietician to inspect your entire project for missing libraries. If any binaries, DLLs, or shared libraries haven’t met load-time library requirements, you’ll be prompted to add these libraries to your project.

Apply diet(s) system wide
This option runs the Dietician on all the libraries selected. The diets are applied in the proper order so that runtime dependencies aren’t broken. If you were to do this by hand, it’s possible that the dieting of one shared library could render a previously dieted shared library non-functional. The order of operations is key!

To ensure that your image works and is as efficient as possible, you should select all three options.

4 Click **Next**. You’ll see a list of the libraries scheduled to be removed, added, or put on a diet. Uncheck the libraries that you don’t want included in the operation, then move to the next page.

5 Click **Finish**. The System Optimizer optimizes your libraries; the dieted libraries appear in your project’s `Reductions/image_name` directory.
Optimizing your system

Optimizing a single library

Optimizing a single library doesn’t reduce the library as effectively as optimizing all libraries simultaneously, because the System Optimizer accounts for dependencies.

To reduce a library such as libc using the Dietician, you must iteratively optimize each individual library in your project between two and five times (depending on the number of dependency levels).

You can reduce a single library to its optimum size if it has no dependencies.

To optimize a single library in an image:

1. If your project isn’t already open, double-click its project.bld file in the Navigator or System Builder Projects view.

2. In the QNX System Builder editor, expand the Shared Libraries list and select the library you want to optimize.

3. In the toolbar, click the Optimize System button ( ).
Optimizing your system

4 In the **System Optimizer**, select the **Apply diet(s) system wide** option.

5 Click **Next**. The Dietician shows the unnecessary libraries (if any).
6 Click **Next**. The Dietician shows any additional needed libraries (if any).
Click Next. The Dietician shows the libraries that can be optimized (if any).
8 Click Finish. The Dietician removes unused libraries, adds the additional required libraries, and generates new, reduced libraries. Reduced libraries are added to your project’s Reductions/image_name directory.

Restoring a slimmed-down library

If after reducing a library, you notice that your resulting image is too “slim,” you can manually remove the reduced library from the Reductions directory, and then rebuild your image using a standard, “full-weight” shared library.

To restore a library to its undieted state:

1 In the Navigator or System Builder Projects view, open the Reductions directory in your project. This directory contains the dieted versions of your libraries.
Moving files between the host and target

2 Right-click the library you want to remove and select **Delete**. Click **OK** to confirm your selection. The IDE deletes the unwanted library; when you rebuild your project, the IDE uses the undieted version of the library.

Moving files between the host and target

The IDE’s Target File System Navigator view lets you easily move files between your host and a filesystem residing on your target.

If you haven’t yet created a target system, you can do so right from within the Target File System Navigator view:

- Right-click anywhere in the view, then select **Add New Target**.

The view displays the target and directory tree in the left pane, and the contents of the selected directory in the right pane:

- If the Target File System Navigator view has only one pane, click the dropdown menu button (▼) in the title bar, then select **Show table**. You can also customize the view by selecting **Table Parameters** or **Show files in tree**.

Note that the Target File System Navigator view isn’t part of the default QNX System Builder perspective; you must manually bring the view into your current perspective.

To see the Target File System Navigator view:
Moving files between the host and target

1. From the main menu, select Window→Show View→Other.…
2. Select QNX Targets, then double-click Target File System Navigator.

Moving files to the target

You can move files from your host machine to your target using copy-and-paste or drag-and-drop methods.

To copy files from your host filesystem and paste them on your target’s filesystem:

1. In a file-manager utility on your host (e.g. Windows Explorer), select your files, then select Copy from the context menu.
2. In the left pane of the Target File System Navigator view, right-click your destination directory and select Paste.

To convert files from DOS to Neutrino (or Unix) format, use the textto -l filename command. (For more information, see textto in the Utilities Reference.)

To drag-and-drop files to your target:

- Drag your selected files from any program that supports drag-and-drop (e.g. Windows Explorer), then drop them in the Target File System Navigator view.

This is not supported on Neutrino hosts.

Moving files from the target to the host

To copy files from your target machine and paste them to your host’s filesystem:

1. In the Target File System Navigator view, right-click a file, then select Copy to→File System. The Browse For Folder dialog appears.
Moving files between the host and target

To import files directly into your workspace, select **Copy to → Workspace**. The Select Target Folder dialog appears.

2 Select your desired destination directory and click **OK**.

To move files to the host machine using drag-and-drop:

- Drag your selected files from the Target File System Navigator view and drop them in the Navigator or System Builder Projects view.

This is not supported on Neutrino hosts.
In this chapter...
What is PhAB?

The Photon microGUI includes a powerful development tool called *PhAB* (Photon Application Builder), a visual design tool that generates the underlying C/C++ code to implement your program’s UI.

With PhAB, you can dramatically reduce the amount of programming required to build a Photon application. You can save time not only in writing the UI portion of your code, but also in debugging and testing. PhAB helps you get your applications to market sooner and with more professional results.

PhAB lets you rapidly prototype your applications. You simply select widgets, arrange them as you like, specify their behavior, and interact with them as you design your interface.

PhAB’s opening screen looks like this:
What is PhAB?

PhAB and the IDE

The IDE frequently runs command-line tools such as *gdb* and *mkefs* “behind the scenes,” but PhAB and the IDE are separate applications; each runs in its own window. You can create files, generate code snippets, edit callbacks, test your UI components, etc. in PhAB, while you continue to use the IDE to manage your project as well as debug your code, run diagnostics, etc.

PhAB was originally designed to run under the Photon microGUI on a QNX Neutrino host, but the *phindows* (“Photon in Windows”) utility lets you run PhAB on a Windows host as well. The IDE lets you see, debug, and interact with your target Photon application right from your host machine as if you were sitting in front of your target machine.
Using PhAB

In most respects, using PhAB inside the IDE is the same as running PhAB as a standalone application.

☞ For a full description of PhAB’s functionality, see the Photon Programmer’s Guide.

Creating a QNX Photon Appbuilder project

In order to use PhAB with the IDE, you must create a QNX Photon Appbuilder project to contain your code. This type of project contains tags and other information that let you run PhAB from within the IDE.

To create a PhAB Project:

1  From the workbench menu, select File→New→Project…
2  In the list, select QNX.
3  Select Photon Appbuilder Project.
Using PhAB

4. Click Next.

5. Name your project.
6 Ensure that **Use default** is checked. Don’t use a different location.

7 Click **Next**.

8 Select your target architecture. Be sure to set one of your variants as the default variant (select the variant, then click the **Default** button).
Using PhAB

If you wish to set any other options for this project, click the remaining tabs and fill in the fields. For details on the tabs in this wizard, see “New C/C++ Project wizard tabs” in the Common Wizards Reference chapter.

9 Click Finish.

The IDE creates your project, then launches PhAB. (In Windows, the IDE also creates a Console for PhAB window.)

Closing PhAB

To end a PhAB session:

➤ From PhAB’s main menu, select File→Exit.
In Windows, don’t end a PhAB session by clicking the Close button in the top-right corner of the PhAB window; clicking this button closes the phindows utility session without letting PhAB itself shut down properly. Subsequent attempts to restart PhAB may fail.

To recover from improperly closing PhAB:

1. Close the Console for PhAB window.
2. Reopen your QNX Photon Appbuilder project.

Reopening PhAB

To reopen your QNX Photon Appbuilder project, open the Project menu and click Open Appbuilder.

Editing code

You can edit the code in your QNX Photon Appbuilder project using both PhAB and the IDE. Using PhAB, you can control the widgets and the overall layout of your program; using either PhAB or the IDE, you can edit the code that PhAB generates and specify the behavior of your callbacks.

To use PhAB to edit the code in a QNX Photon Appbuilder project:

1. In the C/C++ Projects view, select a QNX Photon Appbuilder project.
2. Click the Open Appbuilder button in the toolbar ( ). PhAB starts, then opens your project.
If for some reason the **Open Appbuilder** button isn’t in the C/C++ perspective’s toolbar:

1. From the main menu, select **Window→Customize Perspective**.
2. In the left pane, select **Other→Photon Appbuilder Actions**.
3. Check **Photon Appbuilder Actions**.
4. Click **OK**. The **Open Appbuilder** button (📸) appears in the toolbar.

To use the IDE to edit the code in a QNX Photon Appbuilder project:

- In the C/C++ Projects view, double-click the file you want to edit. The file opens in an editor.

If a file that you created with PhAB doesn’t appear in the C/C++ Projects view, right-click your project and select **Refresh**.

Editing files using two applications can increase the risk of accidentally overwriting your changes. To minimize this risk, close the file in one application before editing the file in the other.

**Building a QNX Photon Appbuilder project**

You build a QNX Photon Appbuilder project in exactly the same way as other projects. (For information on building projects, see the “Building projects” section in the Developing Programs chapter.)

To build a QNX Photon Appbuilder project:

- In the C/C++ Projects view, right-click your QNX Photon Appbuilder project and select **Build**. The IDE builds your project.
Starting Photon applications

You can connect to a Photon session from a Windows or QNX Neutrino host machine and run your Photon program as if you were sitting in front of the target machine. Photon appears in a phindows window on your Windows host or in a phditto window on your QNX Neutrino host.

The remote Photon session runs independently of your host. For example, the clipboards don’t interact, and you can’t drag-and-drop files between the two machines. The phindows and phditto utilities transmit your mouse and keyboard input to Photon and display the resulting state of your Photon session as a bitmap on your host machine.

Before you run a remote Photon session on a Windows host, you must first prepare your target machine. (For details, see the “Connecting with Phindows” section in the Preparing Your Target chapter.)

To start a remote Photon session:

➤ In the Target Navigator view, right-click a target and select **Launch Remote Photon**.

Photon appears in a Phindows window.

You can start a Photon application you created in PhAB in exactly the same way that you launch any other program in the IDE. By default, the program opens in the target machine’s main Photon session. (For more on launching, see the Launch Configurations Reference chapter in this guide.)

To run your Photon program in a remote Photon session window:

1. In the remote Photon session, open a command window (e.g. a terminal from the shelf).

2. In the command window, enter:
   ```
   echo $PHOTON
   ```

   The target returns the session, such as `/dev/ph1470499`. The number after `ph` is the process ID (PID).
Starting Photon applications

3 In the IDE, edit the launch configuration for your QNX Photon Appbuilder project.

4 Select the Arguments tab.

5 In the C/C++ Program Arguments field, enter -s followed by the value of $PHOTON. For example, enter -s /dev/ph1470499.

6 Click Apply, then Run or Debug. Your remote Photon program opens in the phindows or phditto window on your host machine.

If you close and reopen a remote Photon session, you must update your launch configuration to reflect the new PID of the new session.
Chapter 8
Profiling an Application

In this chapter...
Introducing the Application Profiler 251
Profiling your programs 253
Controlling your profiling sessions 263
Understanding your profiling data 265
Introducing the Application Profiler

The QNX Application Profiler perspective lets you examine the overall performance of programs, no matter how large or complex, without following the source one line at a time. Whereas a debugger helps you find errors in your code, the QNX Application Profiler helps you pinpoint “sluggish” areas of your code that could run more efficiently.
Introducing the Application Profiler

The QNX Application Profiler perspective.

Types of profiling

The QNX Application Profiler lets you perform:

- statistical profiling
- instrumented profiling
- postmortem profiling (on standard gmon.out files)

Statistical profiling

The QNX Application Profiler takes “snapshots” of your program’s
execution position every millisecond and records the current address
being executed. By sampling the execution position at regular
intervals, the tool quickly builds a summary of where the system is
spending its time in your code.
With statistical profiling, you don’t need to use instrumentation, change your code, or do any special compilation. The tool profiles your programs nonintrusively, so it doesn’t bias the information it’s collecting.

Note, however, that the results are subject to statistical inaccuracy because the tool works by sampling. Therefore, the longer a program runs, the more accurate the results.

**Instrumented profiling**

If you build your executables with profiling enabled, the QNX Application Profiler can provide *call-pair* information (i.e. which functions called which). When you build a program with profiling enabled, the compiler inserts snippets of code into your functions in order to report the addresses of the called and calling functions. As your program runs, the tool produces an exact count for every call pair.

**Postmortem profiling**

The IDE lets you examine profiling information from a *gmon.out* file produced by an instrumented application. The tool gives you all the information you’d get from the traditional *gprof* tool, but in graphical form. You can examine *gmon.out* files created by your programs, whether you built them using the IDE or the `qcc -p` command. For more on the *gprof* utility, go to [www.gnu.org](http://www.gnu.org); for `qcc`, see the *Utilities Reference*.

**Profiling your programs**

Whether you plan to do your profiling in real time or postmortem (using a *gmon.out* file), you should build your programs with profiling enabled before you start a profiling session.

This section includes these topics:

- Building a program for profiling
- Running and profiling a process
Profiling your programs

- Profiling a running process
- Postmortem profiling

If you already have a gmon.out file, you’re ready to start a postmortem profiling session.

Building a program for profiling

Although you can profile any program, you’ll get the most useful results by profiling executables built for debugging and profiling. The debug information lets the IDE correlate executable code and individual lines of source; the profiling information reports call-pair data.

This table shows the application-profiling features that are possible with the various build variants:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Release version</th>
<th>Debug version</th>
<th>Release v. &amp; profiling</th>
<th>Debug v. &amp; profiling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call pairs</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Statistical</td>
<td>Yes (function</td>
<td>Yes</td>
<td>Yes (function</td>
<td>Yes</td>
</tr>
<tr>
<td>sampling</td>
<td>level)</td>
<td></td>
<td>level)</td>
<td></td>
</tr>
<tr>
<td>Line profiling</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Postmortem</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

To build executables with debug information and profiling enabled:

1. In the C/C++ Projects view, right-click your project and select Properties.
2. In the left pane, select QNX C/C++ Project.
3 In the right pane, select the Options tab.

4 Check the Build with Profiling option:

5 Select the Build Variants tab and check the Debug variant for your targets.

The QNX Application Profiler uses the information in the debuggable executables to correlate lines of code in your executable and the source code. To maximize the information you get while profiling, use executables with debug information for both running and debugging.

6 Click Apply.

7 Click OK.

8 Rebuild your project.
Profiling your programs

To build a Standard Make C/C++ project for profiling, compile and link using the `-p` option. For example, your `Makefile` might have a line like this:

```
CFLAGS=-p  CXXFLAGS=-p  LDFLAGS=-p
```

Running and profiling a process

To run and profile a process, with `qconn` on the target:

1. Create a QNX Application launch configuration for an executable with debug information as you normally would, but don’t click OK. You may choose either a Run or a Debug session.

2. On the launcher, click the Tools tab.

3. Click Add/Delete Tool. The Select tools to support dialog appears.

4. Enable the QNX Application Profiler tool.

5. Click OK.

6. On the launcher, click the QNX Application Profiler tab:
Profiling your programs

7 Fill in these fields:

Profiler update interval (ms)

Use this option to control how often the Profiler polls for data. A low setting causes continuous (but low) network traffic and fast refresh rates. A high setting causes larger network data bursts and may cause higher memory usage on the target because the target must buffer the data. The default setting of 1000 should suffice.

Shared library paths

The IDE doesn’t know the location of your shared library paths, so you must specify the directory containing any libraries that you wish to profile. For a list of the library paths that are automatically included in the search path, see the appendix Where Files Are Stored.

Switch to this tool’s perspective on launch.

Check this option to automatically switch to the QNX Application Profiler perspective when this launch configuration is used.
Profiling your programs

8 If you want the IDE to automatically change to the QNX Application Profiler perspective when you run or debug, check the Switch to this tool’s perspective on launch. box.

9 Click Apply.

10 Click Run or Debug. The IDE starts your program and profiles it.

To produce full profiling information with function timing data, you need to run the application as root. This is the case when running through qconn.

If you run the application as a normal user, the profiler can generate only call-chain information.

Profiling a running process

To profile a process that’s already running on your target:

1 While the application is running, open the Launch Configurations dialog by choosing Run→Debug… from the menu.

2 Select C/C++ QNX Attach to Process w/QConn (IP) in the list on the left.

3 Click the New button to create a new attach-to-process configuration.

4 Configure things as you normally would for launching the application with debugging.

5 On the Tools tab, click Add/Delete Tool… The Tools Selection dialog appears.

6 Enable the QNX Application Profiler tool, then click OK.

7 On the launcher, click the Application Profiler tab:
Fill in these fields:

Profiler update interval (ms)
You use this option to control how often the Profiler polls for data. A low setting causes continuous (but low) network traffic and fast refresh rates. A high setting causes larger network data bursts and may cause higher memory usage on the target because the target must buffer the data. The default setting of 1000 should suffice.

Shared library paths
The IDE doesn’t know the location of your shared library paths, so you must specify the directory containing any libraries that you wish to profile. For a list of the library paths that are automatically included in the search path, see the appendix Where Files Are Stored.

Switch to this tool’s perspective on launch.
Check this to automatically switch to the QNX Application Profiler perspective when using this launcher.
9 Click **Apply**, then click **Debug**. The Select Process dialog is displayed, showing all of the currently running processes:

![Select Process Dialog](image)

10 Select the process you want to profile, then click **OK**.
Your running application won’t generate call-pair information unless you ran it with the `QCONN_PROFILER` environment variable set to `/dev/profiler`.

If you’re launching the application from the IDE, add `QCONN_PROFILER` to the `Environment` tab of the launch configuration’s Properties dialog.

If you’re running the application from the command line, you can simply add `QCONN_PROFILER` to your shell environment, or the application’s command-line:

```bash
QCONN_PROFILER=/dev/profile ./appname
```

### Postmortem profiling

The IDE lets you profile your program after it terminates, using the traditional `gmon.out` file. Postmortem profiling doesn’t provide as much information as profiling a running process:

- Multithreaded processes aren’t supported. Thus, the Thread Processor Usage view always shows the totals of all your program’s threads combined as one thread.

- Call-pair information from shared libraries and DLLs isn’t shown.

Profiling a `gmon.out` file involves three basic steps:

- gathering profiling information into a file
- importing the file into your workspace
- starting the postmortem profiling session

### Gathering profiling information

The IDE lets you store your profiling information in the directory of your choice using the `PROFDIR` environment variable.

To gather profiling information:
Profiling your programs

1. Create a launch configuration for a *debuggable* executable as you normally would, but don’t click **Run** or **Debug**.

   You must have the QNX Application Profiler tool *disabled* in your launch configuration.

2. Select the **Environment** tab.

3. Click **New**.

4. In the **Name** field, type **PROFDIR**.

5. In the **Value** field, enter a valid path to a directory on your target machine.

6. Click **OK**.

7. Run your program. When your program exits successfully, it creates a new file in the directory you specified. The filename format is `pid.fileName` (e.g. `3047466.helloworld.g`). This is the **gmon.out** profiler data file.

**Importing a gmon.out file**

You can bring existing **gmon.out** files that you created outside the IDE into your workspace from your target system.

To import a **gmon.out** file into your workspace:

1. Open the Target File System Navigator view (Window→Show View→Other…→QNX Targets→Target File System Navigator).

2. In the Target File System Navigator view, right-click your file and select **Copy to…→Workspace**. The Select target folder dialog appears.

3. Select the project related to your program.

4. Click **OK**.

5. In the C/C++ Projects view, right-click the file you imported into your workspace and select **Rename**.
Controlling your profiling sessions

6 Enter gmon.out (or gmon.out.n, where n is any numeric character). The IDE renames your file.

Starting a postmortem profiling session

To start the postmortem profiling session:

1 In the C/C++ Projects view, right-click your gmon.out file and select Open in QNX Application Profiler. The Program Selection dialog appears.

2 Select the program that generated the gmon.out file.

3 Click OK. You can now profile your program in the QNX Application Profiler perspective.

Controlling your profiling sessions

The Application Profiler view (Window → Show View → Other… → QNX Application Profiler → Application Profiler) lets you control multiple profiling sessions simultaneously. You can:

• terminate applications
• choose the executable or library to show profiling information for in the Sampling Information, Call Information, and Thread Processor Usage views
Controlling your profiling sessions

The Application Profiler view displays the following as a hierarchical tree for each profiling session:

<table>
<thead>
<tr>
<th>Session item</th>
<th>Description</th>
<th>Possible icons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Launch instance</td>
<td>Launch configuration name and launch type (e.g. prof201 [C/C++ QNX QConn (IP)])</td>
<td>![Icon]</td>
</tr>
<tr>
<td>Profiled program</td>
<td>Project name and start time (e.g. prof201 on localhost pid 4468773 (3/4/03 12:41 PM))</td>
<td>![Icon]</td>
</tr>
<tr>
<td>Application Profiler instance</td>
<td>Program name and target computer (e.g. Application Profiler Attached to: prof201 &lt;4468773&gt; on 10.12.3.200)</td>
<td>![Icon]</td>
</tr>
<tr>
<td>Executable</td>
<td></td>
<td>![Icon]</td>
</tr>
<tr>
<td>Shared libraries</td>
<td></td>
<td>![Icon]</td>
</tr>
<tr>
<td>DLLs</td>
<td></td>
<td>![Icon]</td>
</tr>
<tr>
<td>Unknown</td>
<td></td>
<td>![Icon]</td>
</tr>
</tbody>
</table>

To choose which executable or library to display information for in the QNX Application Profiler perspective:

- In the Application Profiler view, click one of the following:
  - the QNX Application Profiler instance
  - an executable
  - a shared library
  - a DLL

To terminate an application running on a target:
Understanding your profiling data

1. In the Application Profiler view, select a launch configuration.
2. Click the **Terminate** button ( ) in the view’s title bar.

To clear old launch listings from this view, click the **Remove All Terminated Launches** button ( ).

To disconnect from an application running on a target:
1. In the Application Profiler view, select a running profiler.
2. Click the **Disconnect** button ( ) in the view’s title bar.

To clear old launch listings from this view, click the **Remove All Terminated Launches** button ( ).

**Understanding your profiling data**

For each item you select in the Application Profiler view, other views within the QNX Application Profiler perspective display the profiling information for that item:

<table>
<thead>
<tr>
<th>This view:</th>
<th>Shows:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Profiler</td>
<td>Usage by line</td>
</tr>
<tr>
<td>Sampling Information</td>
<td>Usage by function</td>
</tr>
<tr>
<td>Thread Processor Usage</td>
<td>Usage by thread</td>
</tr>
<tr>
<td>Call Information</td>
<td>Call counts</td>
</tr>
</tbody>
</table>

**Usage by line**

The Application Profiler editor lets you see the amount of time your program spends on each line of code and in each function.

To open the editor:
Understanding your profiling data

1. Launch a profile session for a debuggable (i.e. `_g`) executable.

2. In the Application Profiler view, select your program by selecting an Application Profiler instance (⌘I) or an executable (⌘). 

3. In the Sampling Information or Call Information view, double-click a function that you have the source for. The IDE opens the corresponding source file in the Application Profiler editor:
You may get incorrect profiling information if you change your source after compiling, because the Application Profiler editor relies on the line information provided by the debuggable version of your code.

The Application Profiler editor displays a bar graph on the left side. The bars are color coded:

- **Green**: CPU time spent within the function as a percentage of the program’s total CPU time. The green bar appears on the first line of executable code in the function.

- **Orange**: CPU time spent on a line of code as a percentage of the program’s total CPU time. Within a function, the lengths of the orange bars add up to the length of the green bar.

- **Blue**: CPU time spent on a line of code as a percentage of the function’s total CPU time. Within a function, the sum of all the blue bars spans the width of the editor’s margin.

To view quantitative profiling values:

➤ In the Application Profiler editor, let the pointer hover over a colored bar. The CPU usage appears, displayed as a percentage and a time:
Understanding your profiling data

Usage by function

The Sampling Information view shows a flat profile of the item that’s currently selected in the Application Profiler view. You can examine profiling information for programs, shared libraries, and DLLs:

- the total CPU time spent in the function
- the CPU time spent in the function since you last reset the counters

If you select a program compiled for profiling, the view also displays:
- the number of times the function has been called
- the average CPU time per call

To see your function usage:

1. Launch a profile session for a profiling-enabled (i.e. \_g) executable.
2. In the Application Profiler view, select your program by selecting an Application Profiler instance (\_f) or any subordinate line. The Sampling Information view displays profiling information for your selection.

To reset the counters in the Time since last reset(s) column:

- Click the Reset Sample counts button (\_r) in the Sampling Information view’s title bar.
Understanding your profiling data

Usage by thread

The Thread Processor Usage view displays the CPU usage (in seconds and as a percentage of your program’s total time) for each thread of the item that’s currently selected in the Application Profiler view:

You can use this information to:

- identify which threads are the most and least active
- determine the appropriate size of your application’s thread pool. (If there are idle threads, you might want to reduce the size of the pool.)

To see your thread usage:

1. Launch a profile session for a profiling-enabled (i.e. `-g`) executable.

2. In the Application Profiler view, select your program by selecting an Application Profiler instance (Ctrl) or any subordinate line. The Thread Processor Usage view displays profiling information for your selection.

Call counts

For the item that’s currently selected in the Application Profiler view, the Call Information view shows your call counts in three panes:

- Call Pairs
- Call Graph
- Call Pair Details.

To display your call counts:
Understanding your profiling data

1. Launch a profile session for a profiling-enabled (i.e. `-g`) executable.

2. In the Application Profiler view, select your program by selecting an Application Profiler instance or any subordinate line. The Call Information view displays profiling information for your selection:

   ![Call Information View]

   **Call Pairs pane**

   The Call Pairs pane shows you where every function was called from as well as the call-pair count, i.e. the number of times each function called every other function.
Understanding your profiling data

Call Graph pane

The **Call Graph** pane shows you a graph of the function calls. Your selected function appears in the middle, in blue. On the left, in yellow, are all the functions that called your function. On the right, also in yellow, are all the functions that your function called.

To see the calls to and from a function:

➤ Click a function in:

- the **Function** column in the **Call Pairs** pane
- or:
- the **Call Graph** pane

You can display the call graph only for functions that were compiled with profiling enabled.

Call Pair Details pane

The **Call Pair Details** pane shows the information about the function you’ve selected in the Call Graph pane. The **Caller** and **Call Count** columns show the number of times each function called the function you’ve selected.

The **Called** and **Called Count** columns show the number of times your selected function called other functions. This pane shows only the functions that were compiled with profiling. For example, it doesn’t show calls to functions, such as `printf()`, in the C library.
Chapter 9
Using Code Coverage

In this chapter...

Code coverage in the IDE 275
Enabling code coverage 278
Controlling your session 282
Examining data line-by-line 284
Examining your coverage report 285
Seeing your coverage at a glance 287
Use the Code Coverage tool to help test your code.

**Code coverage in the IDE**

Code coverage is a way to measure how much code a particular process has executed during a test or benchmark. Using code-coverage analysis, you can then create additional test cases to increase coverage and determine a quantitative measure of code coverage, which is an indirect measure of the quality of your software (or better, a direct measure of the quality of your tests).

**Types of code coverage**

Several types of metrics are commonly used in commercial code-coverage tools, ranging from simple line or block coverage (i.e. “this statement was executed”) to condition-decision coverage (i.e. “all terms in this Boolean expression are exercised”). A given tool usually provides a combination of types.

The coverage tool in the IDE is a visual front end to the `gcov` metrics produced by the `gcc` compiler. These coverage metrics are essentially basic block coverage and branch coverage.
The IDE presents these metrics as line coverage, showing which lines are fully covered, partially covered, and not covered at all. The IDE also presents percentages of coverage in terms of the actual code covered (i.e. not just lines).

### Block coverage

Block coverage, sometimes known as line coverage, describes whether a block of code, defined as not having any branch point within (i.e. the path of execution enters from the beginning and exits at the end.) is executed or not.

By tracking the number of times the block of code has been executed, the IDE can determine the total coverage of a particular file or function. The tool also uses this information to show line coverage by analyzing the blocks on each line and determining the level of coverage of each.

### Branch coverage

Branch coverage can track the path of execution taken between blocks of code. Although this metric is produced by the gcc compiler, currently the IDE doesn’t provide this information.

### How the coverage tool works

The IDE’s code coverage tool works in conjunction with the compiler (gcc), the QNX C library (libc), and optionally the remote target agent (qconn). When code coverage is enabled for an application, the compiler instruments the code so that at run time, each branch execution to a basic block is counted. During the build, the IDE produces data files in order to recreate the program’s flow graph and to provide line locations of each block.
CAUTION: Since the IDE creates secondary data files at compilation time, you must be careful when building your programs in a multitargeted build environment such as QNX Neutrino.

You must either:

- ensure that the last compiled binary is the one you’re collecting coverage data on,
  or:
- simply enable only one architecture and debug/release variant.

Note also that the compiler’s optimizations could produce unexpected results, so you should perform coverage tests on a unoptimized, debug-enabled build.

When you build a program with the Build with Code Coverage build option enabled and then launch it using a C/C++ QNX Qconn (IP) launch configuration, the instrumented code linked into the process connects to qconn, allowing the coverage data to be read from the process’s data space.

But if you launch a coverage-built process with coverage disabled in the launch configuration, this causes the process to write the coverage information to a data file (.da) at run time, rather than read it from the process’s data space.

You should use data files only if you’re running the local launch configuration on a QNX Neutrino self-hosted development system. Note that the data can later be imported into the IDE code coverage tool.

Once a coverage session has begun, you can immediately view the data. The QNX Code Coverage perspective contains a Code Coverage Sessions view that lists previous as well as currently active sessions. You can explore each session and browse the corresponding source files that have received coverage data.
Enabling code coverage

To build executables with code coverage enabled:

1. In the C/C++ Projects view, right-click your project and select Properties. The properties dialog for your project appears.
2. In the left pane, select QNX C/C++ Project.
3. In the Build Options pane, check Build with Code Coverage.
4. In the Build Variants tab, check only one build variant.

If the IDE is set to build more than one variant, an error is displayed and the OK button is disabled.

5. Click OK.
6. In the C/C++ Projects view, right-click your project and select Rebuild Project.

Enabling code coverage for Standard Make projects

If you’re using your own custom build environment, rather than QNX Makefiles, you’ll have to manually pass the coverage option to the compiler.

To enable code coverage for non-QNX projects

1. Compile using these options to gcc:
   
   `-fprofile-arcs -ftest-coverage
   
   If you’re using qcc, compile with:
   
   `-Wc,-fprofile-arcs -Wc,-ftest-coverage

2. Link using the `-p` option.

For example, your Makefile might look something like this:

```makefile
objects:=Profile.o main.o
CC:=qcc -Vgcc .ntox86
```
Enabling code coverage

CFLAGS:=-g -Wc,-ftest-coverage -Wc,-fprofile-arcs -I. -I../proflibCPP-std
LDFLAGS:=-p -g -L../proflibCPP-std -lProfLib -lcpp
all: profileCPP-std
clean:
   -rm $(objects) profileCPP-std *.bb *.bbg
profileCPP-std: $(objects)
   $(CC) $^ $(LDFLAGS)

Starting a coverage-enabled program

To start a program and measure the code coverage:

1. Create a C/C++ QNX QConn (IP) launch configuration as you normally would, but don’t click OK yet.

2. On the launcher, click the Tools tab.

3. Click Add/Delete Tool. The Tools selection dialog appears.

4. Check the Code Coverage tool:
Enabling code coverage

5 Click **OK**.

6 Click the Code Coverage tab, and fill in these fields:
Enable GCC 3 Coverage metrics collection

Check this if your application was compiled with `gcc` 3.3.1 or later. The default is to collect code coverage information from applications compiled with `gcc` 2.95.3.

Code Coverage data scan interval (sec)

This option sets how often the Code Coverage tool polls for data. A low setting can cause continuous network traffic. The default setting of 5 seconds should suffice.

Referenced projects to include coverage data from

Check any project in this list you wish to gather code-coverage data for. Projects must be built with coverage enabled.

Comments for this coverage session

Your notes about the session, for your own personal use. The comments appear at the top of the generated reports.
Controlling your session

7 Check **Switch to this tool’s perspective on launch** if you want to automatically go to the **QNX Code Coverage** perspective when you run or debug.

8 Click **Apply**.

9 Click **Run** or **Debug**.

**Controlling your session**

The Code Coverage Sessions view lets you control and display multiple code-coverage sessions:

![Code Coverage Sessions](image)

The view displays the following as a hierarchical tree for each session:
Controlling your session

<table>
<thead>
<tr>
<th>Session item</th>
<th>Description</th>
<th>Possible icons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code coverage session</td>
<td>Launch configuration name, coverage tool, and start time (e.g. ccov102_factor [GCC Code Coverage] (7/2/03 2:48 PM))</td>
<td>![icon]</td>
</tr>
<tr>
<td>Project</td>
<td>Project name and amount of coverage (e.g. ccov102_factor [ 86.67% ])</td>
<td>![icon]</td>
</tr>
<tr>
<td>File</td>
<td>Filename and amount of coverage (e.g. ccov102_factor.c [ 86.67% ])</td>
<td>![icon]</td>
</tr>
<tr>
<td>Function</td>
<td>Function name and amount of coverage (e.g. main [ 100% ])</td>
<td>![icon]</td>
</tr>
</tbody>
</table>

The IDE uses several icons in this view:

<table>
<thead>
<tr>
<th>Icon</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>![icon]</td>
<td>No coverage</td>
</tr>
<tr>
<td>![icon]</td>
<td>Partial coverage</td>
</tr>
<tr>
<td>![icon]</td>
<td>Full (100%) coverage</td>
</tr>
<tr>
<td>![icon]</td>
<td>Missing or out-of-date source file</td>
</tr>
</tbody>
</table>

The IDE also adds a coverage markup icon (⊗) to indicate source markup in the editor. (See the “Examining data line-by-line” section, below.)

To reduce the size of the hierarchical tree, click the Collapse All (.removeChild) button.
To combine several sessions:

1. In the Code Coverage Sessions view, select the sessions you want to combine.

2. Right-click your selections and select **Combine/Copy Sessions**. The IDE prompts you for a session name and creates a combined session.

### Examining data line-by-line

The IDE can display the line-by-line coverage information for your source code. In the left margin, the editor displays a “covered” icon (✓) beside each line of source. In the right margin, the editor displays a summary of the coverage by showing green sections for fully-covered code, yellow for partial coverage, and red for no coverage:

To open a file in the QNX Code Coverage perspective:

- In the Code Coverage Sessions view, expand a session and double-click a file or function.
Examining your coverage report

To display coverage information from a particular session:

➤ In the Code Coverage Sessions view, right-click a session and select **Coverage Markup**, then select one of the following:

- Mark lines not covered
- Mark lines partially covered
- Mark lines fully covered

The selected icon appears beside the corresponding source in the C/C++ editor. In the Code Coverage Sessions view, a coverage marker (●) overlays the source file icon.

To automatically show coverage information when opening a file:

1. Open the Preferences dialog (Window→Preferences).
2. In the left pane, select **QNX→Code Coverage**.
3. In the right pane, check the desired markers in the **Coverage markup when file is opened** field.
4. Click **OK**. The next time you open a file, the markers appear automatically. To add markers from another session, add them manually, as described above.

To remove all coverage markers:

➤ In the Code Coverage Sessions view’s title bar, click the **Remove All Coverage Markers** button (●).

Examining your coverage report

The Code Coverage Report view provides a summary (in XML) of your session. The view lets you drill down into your project and see the coverage for individual files and functions:
Examining your coverage report

To generate a report, simply right-click a coverage session and select Generate Report.

By default, the IDE displays reports in the Code Coverage Report view, but you can also have the IDE display reports in an external
Seeing your coverage at a glance

Using an external browser lets you compare several reports simultaneously.

To toggle between viewing reports in the Code Coverage Report view and in an external browser:

1. Open the Preferences dialog (Window→Preferences).
2. In the left pane, select QNX→Code Coverage.
3. In the right pane, check/uncheck the View reports in external browser item.
4. Click OK.

To print a report:

➤ In the Code Coverage Report view’s title bar, click the Print button ( ).

To save a report:

1. Right-click in the Code Coverage Report view to display the context menu.
2. Click Save As... to save the report.

You can also refresh the report:

➤ In the Code Coverage Report view’s title bar, click the Refresh button ( ).

Seeing your coverage at a glance

The Properties view displays a summary of the code coverage for a project, file, or function you’ve selected in the Code Coverage Sessions view.

The Properties view tells you how many lines were covered, not covered, and so on:
Seeing your coverage at a glance

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coverage Info</td>
<td></td>
</tr>
<tr>
<td>Lines Fully Covered</td>
<td>15.83% (115 lines)</td>
</tr>
<tr>
<td>Lines Not Covered</td>
<td>76.22% (454 lines)</td>
</tr>
<tr>
<td>Lines Partially Covered</td>
<td>1.9% (11 lines)</td>
</tr>
<tr>
<td>Total Coverage</td>
<td>23.47%</td>
</tr>
<tr>
<td>Info</td>
<td></td>
</tr>
<tr>
<td>derived</td>
<td>false</td>
</tr>
<tr>
<td>editable</td>
<td>true</td>
</tr>
<tr>
<td>last modified</td>
<td>3/22/05 3:08 PM</td>
</tr>
<tr>
<td>linked</td>
<td>false</td>
</tr>
<tr>
<td>location</td>
<td>C:\Program Files\eclipse\workspace\unzip\fileio.c</td>
</tr>
<tr>
<td>name</td>
<td>fileio.c</td>
</tr>
<tr>
<td>path</td>
<td>unzip/fileio.c</td>
</tr>
<tr>
<td>size</td>
<td>84828</td>
</tr>
</tbody>
</table>
Chapter 10
Finding Memory Errors

In this chapter...

Introduction 291
Analyzing your program 301
Tracing memory events 305
Introduction

Have you ever had a customer say, “The program was working fine for days, then it just crashed”? If so, chances are good that your program had a memory error — somewhere.

Debugging memory errors can be frustrating; by the time a problem appears, often by crashing your program, the corruption may already be widespread, making the source of the problem difficult to trace.

The QNX Memory Analysis perspective shows you how your program uses memory and can help ensure that your program won’t cause problems. The perspective helps you quickly pinpoint memory errors in your development and testing environments before your customers get your product.

Use the QNX Memory Analysis perspective to solve memory problems.
The QNX Memory Analysis perspective may produce incorrect results when more than one IDE is communicating with the same target system. To use this perspective, make sure only one IDE is connected to the target system.

Memory management in QNX Neutrino

By design, Neutrino’s architecture helps ensure that faults, including memory errors, are confined to the program that caused them. Programs are less likely to cause a cascade of faults because processes are isolated from each other and from the microkernel. Even device drivers behave like regular debuggable processes:

This robust architecture ensures that crashing one program has little or no effect on other programs throughout the system. When a program faults, you can be sure that the error is restricted to that process’s operation.

Neutrino’s full memory protection means that almost all the memory addresses your program encounters are virtual addresses. The process manager maps your program’s virtual memory addresses to the actual physical memory; memory that is contiguous in your program may be transparently split up in your system’s physical memory.
The process manager allocates memory in small pages (typically 4KB each). To determine the size for your system, use the `sysconf(SC_PAGESIZE)` function.

As you’ll see when you use the memory-analysis tools, the IDE categorizes your program’s virtual address space as follows:

- program
- stack
- shared library
- objects
- heap
Introduction

The Memory Information and Malloc Information views of the QNX System Information perspective provide detailed, live views of a process’s memory. See the Getting System Information chapter for more information.

**Program memory**

Program memory holds the executable contents of your program. The code section contains the read-only execution instructions (i.e. your actual compiled code); the data section contains all the values of the global and static variables used during your program’s lifetime:
Stack memory

Stack memory holds the local variables and parameters your program’s functions use. Each process in Neutrino contains at least the main thread; each of the process’s threads has an associated stack. When the program creates a new thread, the program can either allocate the stack and pass it into the thread-creation call, or let the system allocate a default stack size and address:

When your program runs, the process manager reserves the full stack in virtual memory, but not in physical memory. Instead, the process manager requests additional blocks of physical memory only when your program actually needs more stack memory. As one function calls another, the state of the calling function is pushed onto the stack.

```
int min=10;
int max = 50;
int main () {
    // Code...
}
```
When the function returns, the local variables and parameters are popped off the stack.

The used portion of the stack holds your thread’s state information and takes up physical memory. The unused portion of the stack is initially allocated in virtual address space, but not physical memory:

At the end of each virtual stack is a *guard page* that the microkernel uses to detect stack overflows. If your program writes to an address within the guard page, the microkernel detects the error and sends the process a SIGSEGV signal.

As with other types of memory, the stack memory appears to be contiguous in virtual process memory, but not necessarily so in physical memory.

**Shared-library memory**

Shared-library memory stores the libraries you require for your process. Like program memory, library memory consists of both code and data sections. In the case of shared libraries, all the processes map to the same physical location for the code section and to unique locations for the data section:
Object memory

Object memory represents the areas that map into a program’s virtual memory space, but this memory may be associated with a physical device. For example, the graphics driver may map the video card’s memory to an area of the program’s address space:

Heap memory

Heap memory represents the dynamic memory used by programs at runtime. Typically, processes allocate this memory using the malloc(), realloc(), and free() functions. These calls ultimately rely on the mmap() function to reserve memory that the malloc library distributes.

The process manager usually allocates memory in 4KB blocks, but allocations are typically much smaller. Since it would be wasteful to use 4KB of physical memory when your program wants only 17 bytes, the malloc library manages the heap. The library dispenses...
the paged memory in smaller chunks and keeps track of the allocated and unused portions of the page:

Each allocation uses a small amount of fixed overhead to store internal data structures. Since there’s a fixed overhead with respect to block size, the ratio of allocator overhead to data payload is larger for smaller allocation requests.

When your program uses the `malloc()` function to request a block of memory, the `malloc` library returns the address of an appropriately sized block. To maintain constant-time allocations, the `malloc` library may break some memory into fixed blocks. For example, the library may return a 20-byte block to fulfill a request for 17 bytes, a 1088-byte block for a 1088-byte request, and so on.

When the `malloc` library receives an allocation request that it can’t meet with its existing heap, the library requests additional physical
memory from the process manager. As your program frees memory, the library merges adjacent free blocks to form larger free blocks wherever possible. If an entire memory page becomes free as a result, the library returns that page to the system. The heap thus grows and shrinks in 4KB increments:

What the Memory Analysis perspective can reveal

The main system allocator has been instrumented to keep track of statistics associated with allocating and freeing memory. This lets the memory statistics module nonintrusively inspect any process’s memory usage.

When you launch your program with the Memory Analysis tool, your program uses the debug version of the malloc library (libmalloc.so). Besides the normal statistics, this library also tracks the history of every allocation and deallocation, and provides cover functions for the string and memory functions (e.g. strcmp(), memcpy(), memmove()). Each cover function validates the corresponding function’s arguments before using them. For example, if you allocate 16 bytes, then forget the terminating null character and attempt to copy a 16-byte string into the block using the strcpy() function, the library detects the error.
The debug version of the `malloc` library uses more memory than the nondebug version. When tracing all calls to `malloc()` and `free()`, the library requires additional CPU overhead to process and store the memory-trace events.

The QNX Memory Analysis perspective can help you pinpoint and solve various kinds of problems, including:

- memory leaks
- memory errors

### Memory leaks

Memory leaks can occur if your program allocates memory and then forgets to free it later. Over time, your program consumes more memory than it actually needs.

In its mildest form, a memory leak means that your program uses more memory than it should. QNX Neutrino keeps track of the exact memory your program uses, so once your program terminates, the system recovers all the memory, including the lost memory.

If your program has a severe leak, or leaks slowly but never terminates, it could consume all memory, perhaps even causing certain system services to fail.

These tools in the QNX Memory Analysis perspective can help you find and fix memory leaks:

- Memory Trace view — shows you all the instances where you program allocates, reallocates, and frees memory. The view lets you hide allocations that have a matching call to `free()`; the remaining allocations are either still in use or forgotten.

- Memory Problems view — shows you all memory errors, including leaks (unreachable blocks).
Memory errors

Memory errors can occur if your program tries to free the same memory twice or uses a stale or invalid pointer. These “silent” errors can cause surprising, random application crashes. The source of the error can be extremely difficult to find, because the incorrect operation could have happened in a different section of code long before an innocent operation triggered a crash.

In the event of a such an error, the IDE can stop your program’s execution and let you see all the allocations that led up to the error. The Memory Problems view displays memory errors and the exact line of source code that generated each error. The Memory Trace view lets you find the prior call that accessed the same memory address, even if your program made the call days earlier.

To learn more about the common causes of memory problems, see Heap Analysis: Making Memory Errors a Thing of the Past in the QNX Neutrino Programmer’s Guide.

Analyzing your program

To extract the most information from your program, you should launch it with the Memory Analysis tool enabled:

1 Create a Run or Debug type of QNX Application launch configuration as you normally would, but don’t click Run or Debug.

2 In the Create, manage, and run configurations dialog, click the Tools tab.

3 Click Add/Delete Tool.

4 In the Tools Selection dialog, check the Memory Analysis tool.

5 Click OK.

6 Click the Memory Analysis tab.
Configure the Memory Analysis settings for your program:

Memory Errors

This group of configuration options controls the Memory Analysis module’s behavior when memory errors are detected.

Enable error detection

Check this to detect memory allocation, deallocation, and access errors:

Verify parameters in string and memory functions

When enabled, check the parameters in calls to `str*()` and `mem*()` functions for sanity.

Perform full heap integrity check on every allocation/deallocation

When enabled, check the heap’s memory chains for consistency before every allocation or deallocation. Note that this
Analyzing your program

checking comes with a performance penalty.

**Enable bounds checking (where possible)**

When enabled, check for buffer overruns and underruns. Note that this is possible only for dynamically allocated buffers.

**When an error is detected**

Memory Analysis takes the selected action when a memory error is detected. By default, it reports the error and attempt to continue, but you can also choose to launch the debugger or terminate the process.

**Limit trace-back depth to**

Specify the number of stack frames to record when logging a memory error.

**Memory Tracing**

This group of configuration options controls the Memory Analysis module’s memory tracing features.

**Enable memory allocation/deallocation tracing**

When checked, trace all memory allocations and deallocations.

**Enable leak detection**

When checked, detect memory leaks:

- **Perform leak check automatically every n seconds**
  
  When checked, look for memory leaks every n seconds (default: 60).

- **Perform leak check when process exits**
  
  When checked, look for memory leaks when the process exits, before
Analyzing your program

the operating system cleans up the process’s resources.

**Limit back-trace depth to**

Specify the number of stack frames to record when tracing memory events.

**Advanced**

Click the **Advanced** button to modify the advanced configuration properties such as the path to the debugging `malloc()` library, the destination for logged data, etc.

---

8 If you want the IDE to automatically change to the QNX Memory Analysis perspective when you run or debug, check **Switch to this tool’s perspective on launch.**

9 Click **Apply** to save your changes.

10 Click **Run** or **Debug**. The IDE starts your program and lets you analyze your program’s memory.
Tracing memory events

When you launch a Memory Analysis session, a new entry is made in the Memory Analysis Sessions view:

When you select an entry in the Memory Analysis Sessions view, the Memory Trace view shows the associated memory events:
Tracing memory events

Each event includes the address of the memory block, the block size, the workbench resource and location that caused the memory event, as well as the project folder name and memory analysis session name.

Select a row in the Memory Trace view to see a backtrace of one or more function calls in the Event Backtrace view.

Double-click a row in the Memory Trace view to open that event’s source file and go to the line that caused the memory event.

Select a column in the Memory Trace view to sort the data by the entries in that column.

The Memory Trace view uses the following icons to indicate different types of events:

- An allocated block that has become a leak (no references to this block exist; it cannot be deallocated by the application until it exits).
- An allocation that has a matching deallocation.
- An allocation event.
- A deallocation that has a matching allocation.
- A deallocation event.

Filtering memory trace events

You can filter the events displayed in the Memory Trace view by choosing Filters... from the view’s menu (▼):
To filter memory trace events:

1. In the Filters dialog, check the **Enabled** box.

2. To limit the number of displayed items, check **Limit visible items to**: and enter the maximum number of items to display.
Tracing memory events

3 Select one or more memory events in the Show events of type: list by checking their boxes.

You can quickly select or deselect all memory events using the Select All and Deselect All buttons below the list.

4 Use the radio buttons to limit the displayed events to this Memory Analysis session, the currently selected source file, or the current working set, if any.

You can select a working set by clicking the Select… button.

5 To hide matching allocations and deallocations, check Hide matching allocation/deallocation pairs.

6 Click OK to close the Filters dialog and apply your filter settings.

Memory Problems view

The Memory Problems view displays memory leaks, errors (such as buffer overruns), and warnings.

Each entry in the Memory Problems view displays a description, the memory address, size, and the source code location (resource and line number) of the memory problem.

Select a column in the Memory Problems view to sort the data by the entries in that column.

Double-click an entry to open an editor with the source code line highlighted.
Filtering memory problems

You can filter the problems displayed in the Memory Problems view by choosing Filters… from the view’s menu (▼):
To filter memory problems:

1. In the Filters dialog, check the **Enabled** box.

2. To limit the number of displayed items, check **Limit visible items to:** and enter the maximum number of items to display.

3. Select one or more memory events in the **Show events of type:** list by checking their boxes.
   
   You can quickly select or deselect all memory events using the **Select All** and **Deselect All** buttons below the list.

4. Use the radio buttons to limit the displayed events to this Memory Analysis session, the currently selected source file, or the current working set, if any.
   
   You can select a working set by clicking the **Select...** button.

5. To search the descriptions for a specific string, enter it in the **Where description** field and select **contains** from the drop-down menu.
   
   You can search for descriptions that don’t contain the specified string by choosing **does not contain** from the drop-down menu.

6. Click **OK** to close the Filters dialog and apply your filter settings.

**Event Backtrace view**

The Event Backtrace view displays a call stack trace leading up to your selected memory event or error:
To display error information in the Event Backtrace view:

➤ Click on a memory event in the Memory Trace view.
Chapter 11
Getting System Information

In this chapter...

Introduction 315
What the System Information perspective reveals 316
Controlling your system information session 320
Examining your target system’s attributes 324
Watching your processes 326
Examining your target’s memory 328
Examining process signals 335
Getting channel information 336
Tracking file descriptors 338
Tracking resource usage 339
Introduction

The IDE provides a rich environment not only for developing and maintaining your software, but also for examining the details of your running target systems.

Within the IDE, you’ll find several views whose goal is to provide answers to such questions as:

- Are my processes running?
- What state are they in?
- What resources are being used, and by which processes?
- Which processes/threads are communicating with which other processes/threads?

Such questions play an important role in your overall system design. The answers to these questions often lie beyond examining a single process or thread, as well as beyond the scope of a single tool, which is why a structured suite of integrated tools can prove so valuable.
What the System Information perspective reveals

The tools discussed in this chapter are designed to be mixed and matched with the rest of the IDE’s development components to help you gain insight into your system and thereby develop better products.

What the System Information perspective reveals

The System Information perspective provides a complete and detailed report on your system’s resource allocation and use, along with key metrics such as CPU usage, program layout, the interaction of different programs, and more:

The perspective’s metrics may prove useful throughout your development cycle, from writing and debugging your code through your quality-control strategy.
What the System Information perspective reveals

Key terms

Before we describe how to work with the System Information perspective, let’s first briefly discuss the terms used in the perspective itself. The main items are:

- **thread**: The minimum “unit of execution” that can be scheduled to run.
- **process**: A “container” for threads, defining the virtual address space within which threads execute. A process always contains at least one thread. Each process has its own set of virtual addresses, typically ranging from 0 to 4GB.

  Threads within a process share the same virtual memory space, but have their own stack. This common address space lets threads within the process easily access shared code and data, and lets you optimize or group common functionality, while still providing process-level protection from the rest of the system.

- **scheduling priority**: Neutrino uses priorities to establish the order in which threads get to execute when multiple threads are competing for CPU time.

  Each thread can have a scheduling priority ranging from 1 to 255 (the highest priority), independent of the scheduling policy. The special idle thread (in the process manager) has priority 0 and is always ready to run. A thread inherits the priority of its parent thread by default.

  You can set a thread’s priority using the pthread_setschedparam() function.

- **scheduling policy**: When two or more threads share the same priority (i.e. the threads are directly competing with each other for the CPU), the OS relies on the threads’ scheduling policy to determine which thread should run next. Three policies are available:
What the System Information perspective reveals

- round-robin
- FIFO
- sporadic

You can set a thread’s scheduling policy using the `pthread_setschedparam()` function or you can start a process with a specific priority and policy by using the `on -p` command (see the Utilities Reference for details).

**state**

Only one thread can actually run at any one time. If a thread isn’t in this RUNNING state, it must either be READY or BLOCKED (or in one of the many “blocked” variants).

**message passing**

The most fundamental form of communication in Neutrino. The OS relays messages from thread to thread via a send-receive-reply protocol. For example, if a thread calls `MsgSend()`, but the server hasn’t yet received the message, the thread would be SEND-blocked; a thread waiting for an answer is REPLY-blocked, and so on.

**channel**

Message passing is directed towards channels and connections, rather than targeted directly from thread to thread. A thread that wishes to receive messages first creates a channel; another thread that wishes to send a message to that thread must first make a connection by “attaching” to that channel.

**signal**

Asynchronous event notifications that can be sent to your process. Signals may include:

- simple alarms based on a previously set timer
- a notification of unauthorized access of memory or hardware
- a request for termination
- user-definable alerts
What the System Information perspective reveals

The OS supports the standard POSIX signals (as in UNIX) as well as the POSIX realtime signals. The POSIX signals interface specifies how signals target a particular process, not a specific thread. To ensure that signals go to a thread that can handle specific signals, many applications mask most signals from all but one thread.

You can specify the action associated with a signal by using the `sigaction()` function, and block signals by using `sigprocmask()`. You can send signals by using the `raise()` function, or send them manually using the Target Navigator view (see “Sending a signal” below).

For more information on all these terms and concepts, see the QNX Neutrino Microkernel chapter in the System Architecture guide.

The views in this perspective

You use the views in the System Information perspective for these main tasks:

<table>
<thead>
<tr>
<th>To:</th>
<th>Use this view:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control your system information session</td>
<td>Target Navigator</td>
</tr>
<tr>
<td>Examine your target system’s attributes</td>
<td>System Summary</td>
</tr>
<tr>
<td>Watch your processes and view thread activity</td>
<td>Process Information</td>
</tr>
<tr>
<td>Inspect virtual address space</td>
<td>Memory Information</td>
</tr>
<tr>
<td>Track heap usage</td>
<td>Malloc Information</td>
</tr>
</tbody>
</table>

continued…
### Controlling your system information session

<table>
<thead>
<tr>
<th><strong>To:</strong></th>
<th><strong>Use this view:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Examine process signals</td>
<td>Signal Information</td>
</tr>
<tr>
<td>Get channel information</td>
<td>System Blocking Graph</td>
</tr>
<tr>
<td>Track file descriptors</td>
<td>Connection Information</td>
</tr>
<tr>
<td>Track resource usage</td>
<td>System Resources</td>
</tr>
</tbody>
</table>

**Controlling your system information session**

The selections you make in the Target Navigator view control the information you see in the System Information perspective:
Controlling your system information session

You can customize the Target Navigator view to:

- sort processes by PID (process ID) or by name
- group processes by PID family
- control the refresh rate

To access the Target Navigator view’s customization menu, click the menu button ( ) in the Target Navigator view’s title bar.

You can reverse a selected sort order by clicking the **Reverse sort** button ( ) in the view’s title bar.

You can enable or disable the automatic refresh by clicking the **Automatic Refresh** button ( ) in the view’s title bar. Entries in the Target Navigator are grey when their data is stale and needs refreshing.
Controlling your system information session

If you’ve disabled automatic refresh, you can refresh the Target Navigator view by right-clicking and choosing Refresh from the context menu.

The Target Navigator view also let you control the information displayed by the following views:

- Malloc Information
- Memory Information

To control the display in the Malloc Information or Memory Information view:

➤ In the Target Navigator view, expand a target and select a process:

![Target Navigator screenshot]

Sending a signal

The Target Navigator view lets you send signals to the processes on your target. For example, you can terminate a process by sending it a SIGTERM signal.

To send a signal to a process:

1. In the Target Navigator view, right-click a process and select Deliver Signal.
Controlling your system information session

2 Select a signal from the dropdown menu.
3 Click **OK**. The IDE delivers the signal to your selected process.

---

**CAUTION:** Delivering a signal to a process usually causes that process to terminate.

### Updating the views

To update the views in the System Information perspective:

- In the Target Navigator view, expand a target and select a process. (You can also select groups of processes by using the Ctrl or Shift keys.) The views reflect your selection.

The data displayed in the System Information perspective is updated automatically whenever new data is available.

### Adding views to the System Information perspective

By default, some views don’t appear in the System Information perspective. To add a view to the perspective:

1 From the main menu, select **Window→Show View** and select a view.
2 The view appears in your perspective.
3 If you want to save a customized set of views as a new perspective, select **Window→Save Perspective As** from the main menu.
Examining your target system’s attributes

Some of the views associated with the System Information perspective can add a noticeable processing load to your host CPU. You can improve its performance by:

- Closing the System Information perspective when you’re not using it.
- Closing unneeded views within the perspective. You can instantly reopen all the closed views by selecting Window→Reset Perspective from the main menu.
- Reducing the refresh rate (as described above).
- Minimizing or hiding unneeded views.

Examining your target system’s attributes

The System Summary view displays a listing of your target’s system attributes, including your target’s processor(s), memory, active servers, and processes:
Examining your target system’s attributes

The System Summary view includes the following panes:

- System Specifications
- System Memory
- Processes

**System Specifications pane**

The System Specifications pane displays your system’s hostname, board type, OS version, boot date, and CPU information. If your target is an SMP system, the pane lists CPU information for each processor.

**System Memory pane**

The System Memory pane displays your system’s total memory and free memory in numerical and graphical form.
Watching your processes

Processes panes

The Processes panes display the process name, heap usage, CPU usage time, and start time for the processes running on your selected target. The panes lets you see application processes, server processes, or both. Server processes have a session ID of 1; application processes have a session ID greater than 1.

Watching your processes

The Process Information view displays information about the processes you select in the Target Navigator view. The view shows the name of the process, its arguments, environment variables, and so on. The view also shows the threads in the process and the states of each thread:

The Process Information view includes the following panes:

- Thread Details
- Environment Variables
- Process Properties
Watching your processes

Thread Details pane

The Thread Details pane shows information about your selected process’s threads, including the thread’s ID, priority, scheduling policy, state, and stack usage.

The Thread Details pane lets you display a substantial amount of information about your threads, but some of the column entries aren’t shown by default.

To configure the information displayed in the Thread Details pane:

1. In the Process Information view, click the menu dropdown button (▼).

2. Select Configure. The Configure dialog appears:

   ![Configure dialog]

3. You can:
   - Add entries to the view by selecting items from the Available Items list and clicking Add.
Examining your target’s memory

- Remove entries from the view by selecting items in the New Items list and clicking **Remove**.
- Adjust the order of the entries by selecting items in the New Items list and clicking **Shift Up** or **Shift Down**.

4 Click **OK**. The view displays the entries that you specified in the New Items list.

**Environment Variables pane**

The Environment Variables pane provides the values of the environment variables that are set for your selected process. (For more information, see the Commonly Used Environment Variables appendix in the Utilities Reference.

**Process Properties pane**

The Process Properties pane shows the process’s startup arguments, and the values of the process’s IDs: real user, effective user, real group, and effective group.

The process arguments are the arguments that were used to start your selected process as they were passed to your process, but not necessarily as you typed them. For example, if you type `ws *.c`, the pane might show `ws cursor.c io.c my.c phditto.c swaprelay.c`, since the shell expands the `*.c` before launching the program.

The process ID values determine which permissions are used for your program. For example, if you start a process as **root**, but use the `seteuid()` and `setegid()` functions to run the program as the user **jsmith**, the program runs with **jsmith**’s permissions. By default, all programs launched from the IDE run as **root**.

**Examining your target’s memory**

Two views in the QNX System Information perspective are especially useful for examining the memory of your target system:

- Malloc Information view (for heap usage and other details)
Examining your target's memory

- Memory Information view (for examining virtual address space)

**Malloc Information view**

The Malloc Information view displays statistical information from the general-purpose, process-level memory allocator:

- **Total Heap**
- **Calls Made**
- **Core Requests**
- **Distribution**

When you select a process in the Target Navigator view, the IDE queries the target system and retrieves the allocator's statistics. The IDE gathers statistics for the number of bytes that are allocated, in use, and overhead.

The view includes the following panes:

- Total Heap
- Calls Made
- Core Requests
- Distribution
Examining your target's memory

- History

**Total Heap**

The Total Heap pane shows your total heap memory, which is the sum of the following states of memory:

- used (dark blue)
- overhead (turquoise)
- free (lavender)

The bar chart shows the relative size of each.

**Calls Made**

The Calls Made pane shows the number of times a process has allocated, freed, or reallocated memory by calling `malloc()`, `free()`, and `realloc()` functions. (See the *Neutrino Library Reference*.)

**Core Requests**

The Core Requests pane displays the number of allocations that the system allocator automatically made to accommodate the needs of the program you selected in the Target Navigator view. The system allocator typically dispenses memory in increments of 4KB (one page).

The number of allocations never equals the number of deallocations, because when the program starts, it allocates memory that isn’t released until it terminates.

**Distribution**

The Distribution pane shows a distribution of the memory allocation sizes. The pane includes the following columns:

- **Byte Range** The size range of the memory blocks.
- **Total mallocs and frees** The total number of calls that effectively allocate or free memory. For example, if your program

---

Chapter 11 • Getting System Information
Examining your target's memory

reallocated memory from 10 bytes to 20 bytes, both the free count for the 0-16 byte range and the malloc count for the 17-32 range would increment.

Allocated  The remaining number of allocated blocks. The value is equal to the number of allocations minus the number of deallocations.

% Returned  The ratio of freed blocks to allocated blocks, expressed as a percentage. The value is calculated as the number of deallocations divided by the number of allocations.

Usage (min/max)

The calculated minimum and maximum memory usage for a byte range. The values are calculated by multiplying the number of allocated blocks by the minimum and maximum sizes of the range. For example, if the 65-128 byte range had two blocks allocated, the usage would be \(\frac{130}{160}\). You should use these values for estimated memory usage only; the actual memory usage usually lies somewhere in between.

History

The History pane shows a chronology of the heap usage shown in the Total Heap pane. The pane automatically rescales as the selected process increases its total heap.

The History pane updates the data every second, with a granularity of 1KB. Thus, two 512-byte allocations made over several seconds trigger one update.
Examining your target’s memory

You can choose to hide or display the Distribution and History panes:

1. In the Malloc Information view’s title bar, click the dropdown menu button (▲), followed by Show.
2. Click the pane you want displayed.

Virtual address space

The Memory Information view displays the memory used by the process you select in the Target Navigator view:

The view shows the following major categories of memory usage:

- Stack (red)
  - guard (light)
Examining your target’s memory

- unallocated (medium)
- allocated (dark)

- Program (royal blue)
  - data (light)
  - code (dark)

- Heap (blue violet)

- Objects (powder blue)

- Shared Library (green)
  - data (light)
  - code (dark)

- Unused (white)

The Process Memory pane shows the overall memory usage. To keep large sections of memory from visually overwhelming smaller sections, the view scales the display semilogarithmically and indicates compressed sections with a split.

Below the Process Memory pane, the Process Memory subpane shows your selected memory category (e.g. Stack, Library) linearly. The subpane colors the memory by subcategory (e.g. a stack’s guard page), and shows unused memory.

The Memory Information view’s table lists all the memory segments and the associated virtual address, size, permissions, and offset. The major categories list the total sizes for the subcategories (e.g. Library lists the sizes for code/data in the Size column). The Process Memory pane and subpane update their displays as you make selections in the table.

The Memory Information view’s table includes the following columns:

Name The name of the category.
Examining your target's memory

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>V. Addr.</td>
<td>The virtual address of the memory.</td>
</tr>
<tr>
<td>Size</td>
<td>The size of the section of memory. For the major categories, the column lists the totals for the minor categories.</td>
</tr>
<tr>
<td>Map Flags</td>
<td>The flags and protection bits for the memory block. See the <code>mmap()</code> function’s flags and prot arguments in the Neutrino Library Reference.</td>
</tr>
<tr>
<td>Offset</td>
<td>The memory block’s offset into shared memory, which is equal to the <code>mmap()</code> function’s off argument.</td>
</tr>
</tbody>
</table>

To toggle the Memory Information view’s table arrangement between a flat list and a categorized list:

- Select the dropdown menu (▼) in the Memory Information view’s title bar and select Categorize.

Stack errors

Stack errors can occur if your program contains functions that are deeply recursive or use a significant amount of local data. Errors of this sort can be difficult to find using conventional testing; although your program seems to work properly during testing, the system could fail in the field, likely when your system is busiest and is needed the most.

The Memory Information view lets you see how much stack memory your program and its threads use. The view can warn you of potential stack errors.

Inefficient heap usage

Your program can experience problems if it uses the heap inefficiently. Memory-allocation operations are expensive, so your program may run slowly if it repeatedly allocates and frees memory, or continuously reallocates memory in small chunks.
Examining process signals

The Malloc Information view displays a count of your program’s memory allocations; if your program has an unusually high turnover rate, this might mean that the program is allocating and freeing more memory than it should.

You may also find that your program uses a surprising amount of memory, even though you were careful not to allocate more memory than you required. Programs that make many small allocations can incur substantial overhead.

The Malloc Information view lets you see the amount of overhead memory the `malloc` library uses to manage your program’s heap. If the overhead is substantial, you can review the data structures and algorithms used by your program, and then make adjustments so that your program uses its memory resources more efficiently. The Malloc Information view lets you track your program’s reduction in overall memory usage.

To learn more about the common causes of memory problems, see Heap Analysis: Making Memory Errors a Thing of the Past in the QNX Neutrino Programmer’s Guide.

Examine process signals

The Signal Information view shows the signals for the processes selected in the Target Navigator view.
Getting channel information

The view shows signals that are:

- blocked — applies to individual threads
- ignored — applies to the entire process
- pending

You can send a signal to any process by using the Target Navigator view (see the section “Sending a signal” in this chapter).

Getting channel information

The System Blocking Graph view presents a color-coded display of all the active channels in the system and illustrates the interaction of threads with those channels.

Interaction with resource objects are such that a thread can be blocked waiting for access to the resource or waiting for servicing (i.e. the thread is SEND-blocked on a channel).

The thread could also be blocked waiting for a resource to be released back to the thread or waiting for servicing to terminate (i.e. the thread is REPLY-blocked).
Clients in such conditions are shown on the left side of the graph, and the resource under examination is in the middle. Threads that are waiting to service a request or are active owners of a resource, or are actively servicing a request, are displayed on the right side of the graph:

In terms of “classical” QNX terminology, you can think of the items in the legend at the top of the graph like this:

<table>
<thead>
<tr>
<th>Legend item</th>
<th>Thread state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Servicing request</td>
<td>Not RECEIVE-blocked (e.g. RUNNING, blocked on a mutex, etc.)</td>
</tr>
<tr>
<td>Waiting for request</td>
<td>RECEIVE-blocked</td>
</tr>
<tr>
<td>Waiting for reply</td>
<td>REPLY-blocked</td>
</tr>
<tr>
<td>Waiting for service</td>
<td>SEND-blocked</td>
</tr>
</tbody>
</table>
Tracking file descriptors

The Connection Information view displays the file descriptors, server, and connection flags related to your selected process’s connections. The view also shows (where applicable) the pathname of the resource that the process accesses through the connection:

<table>
<thead>
<tr>
<th>File Descriptors</th>
<th>Server Name</th>
<th>IOFlags</th>
<th>Seek Offset</th>
<th>Resource Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>proc抗击-ist (1)</td>
<td>--</td>
<td>0</td>
<td>/dev/con1</td>
</tr>
<tr>
<td>1</td>
<td>proc抗击-ist (1)</td>
<td>--</td>
<td>0</td>
<td>/dev/con2</td>
</tr>
<tr>
<td>2</td>
<td>proc抗击-ist (1)</td>
<td>--</td>
<td>0</td>
<td>/dev/con3</td>
</tr>
<tr>
<td>6s</td>
<td>dev-con (7)</td>
<td>--</td>
<td>0</td>
<td>/dev/con4</td>
</tr>
<tr>
<td>9s</td>
<td>proc抗击-ist (1)</td>
<td>--</td>
<td>0</td>
<td>/dev/ibd</td>
</tr>
</tbody>
</table>

The information in this view comes from the individual resource manager servers that are providing the connection. Certain resource managers may not have the ability to return all the requested information, so some fields are left blank.

The IOFlags column describes the read (r) and write (w) status of the file. A double dash (--) indicates no read or write permission; a blank indicates that the information isn’t available.

The Seek Offset column indicates the connector’s offset from the start of the file.

Note that for some FDs, an “s” appears beside the number. This means that the FD in question was created via a side channel — the connection ID is returned from a different space than file descriptors, so the ID is actually greater than any valid file descriptor.
Tracking resource usage

For more information on side channels, see `ConnectAttach()` in the *Neutrino Library Reference*.

To see the full side channel number:

1. In the Connection Information view, click the menu dropdown button (▼).

2. Select **Full Side Channels**.

**Tracking resource usage**

The System Resources view shows various pieces of information about your system’s processes. You can choose one of the following displays:

- System Uptime
- General Resources
- Memory Resources

To select which display you want to see, click the menu dropdown button (▼) in the System Resources view.

**System Uptime display**

The System Uptime display provides information about the start time, CPU usage time, and the usage as a percent of the total uptime, for all the processes running on your selected target:
Tracking resource usage

The General Resources display provides information about CPU usage, heap size, and the number of open file descriptors, for all the processes running on your selected target.
Tracking resource usage

Memory Resources display

The Memory Resources display provides information about the heap, program, library, and stack usage for each process running on your selected target:
Tracking resource usage

To learn more about the meaning of the values shown in the Memory Resources display, see the Finding Memory Errors chapter in this guide.
Chapter 12

Analyzing Your System with Kernel Tracing

In this chapter...

Introducing the QNX System Profiler 345
Configuring a target for system profiling 349
Capturing instrumentation data in event log files 353
Viewing and interpreting the captured data 355
Introducing the QNX System Profiler

The System Profiler is a tool that works in concert with the Neutrino instrumented kernel (procnto-instr) to provide insight into the operating system’s events and activities. Think of the System Profiler as a system-level software logic analyzer. Like the Application Profiler, the System Profiler can help pinpoint areas that need improvement, but at a system-wide level.

The instrumented kernel can gather a variety of events, including:

- kernel calls
- process manager activities
- interrupts
- scheduler changes
- context switches
- user-defined trace data

You might use the System Profiler to solve such problems as:

Use the System Profiler to analyze your system via instrumentation.
Introducing the QNX System Profiler

- IPC bottlenecks (by observing the flow of messages among threads)
- resource contention (by watching threads as they change states)
- cache coherency in an SMP machine (by watching threads as they migrate from one CPU to another)

Details on kernel instrumentation (such as types and classes of events) are more fully covered in the System Analysis Toolkit (SAT) User’s Guide.

The QNX System Profiler perspective includes several components that are relevant to system profiling:

Navigator view

Events are stored in log files (with the extension .kev) within projects in your workspace. These log files are associated with the System Profiler editor.

Target Navigator view

When you right-click a target machine in the Target Navigator view, you can select Kernel Events Tracing..., which initiates the Trace Logging wizard. You use this wizard to specify which events to capture, the duration of the capture period, as well as specific details about where the generated event log file (.kev file) is stored.

System Profiler editor

This editor provides the graphical representation of the instrumentation events in the captured log file. Like all other Eclipse editors, the System Profiler editor shows up in the editor area and can be brought into any perspective. This editor is automatically associated with .kev files, but if you have other file types that contain instrumentation data, you could associate the editor with those files as well.
Introducing the QNX System Profiler

Trace Event Log view
This view lists instrumentation events, as well as their details (time, owner, etc.), surrounding the selected position in the currently active System Profiler editor.

General Statistics view
A tabular statistical representation of events.

Statistics can be gathered for the entire log file or for a selected range.

Condition Statistics view
A tabular or graphical statistical representation of the conditions used in the search panel.

Event Owner Statistics view
A tabular statistical representation of events broken down per owner.

Bookmarks view
Just as you can bookmark lines in a text file, here you can bookmark particular locations and event ranges displayed in the System Profiler editor, then see your bookmarked events in the Bookmarks view.

The QNX System Profiler perspective may produce incorrect results when more than one IDE is communicating with the same target system. To use this perspective, make sure only one IDE is connected to the target system.

Before you begin
As mentioned earlier, in order to capture instrumentation data for analysis, the instrumented kernel (\texttt{procnto-instr}) must be running. This kernel is a drop-in replacement for the standard kernel (though the instrumented kernel is slightly larger). When you’re not gathering instrumentation data, the instrumented kernel is almost exactly as fast as the regular kernel.
Introducing the QNX System Profiler

To determine if the instrumented kernel is running, enter this command:

```bash
ls /proc/boot
```

If `procnto-instr` appears in the output, then the OS image is running the instrumented kernel.

To substitute the `procnto-instr` module in the OS image on your board, you can either manually edit your buildfile, then run `mkifs` to generate a new image, or use the System Builder perspective to configure the image’s properties.

**Replacing the kernel using the System Builder**

1. In the System Builder Projects view, double-click the `project.bld` file for the image you want to change.

2. In the Images pane of the System Builder editor, select the image.

3. In the Properties view, click the **Procnto** field (under **System**). A dropdown-menu button appears in the field:

   ![Properties View](image)

4. Select `procnto-instr`, press Enter, then save your change.
5 Rebuild your project, then transfer your new OS image to your board.

Assuming you’re running the instrumented kernel on your board, you’re ready to use the System Profiler. A profiling session usually involves these three steps:

- configuring a target for system profiling
- capturing instrumentation data in event log files
- viewing and interpreting the captured data

### Configuring a target for system profiling

You can gather trace events from the instrumented kernel in two different ways. You run a command-line utility (e.g. `tracelogger`) on your target to generate a log file, and then transfer that log file back to your development environment for analysis. Or, you can capture events directly from the IDE using the Trace Events Configuration wizard.

In order to get timing information from the kernel, you need to run `tracelogger` as the root user.

If you gather system profiling data through `qconn` in the IDE, you’re already accessing the instrumented kernel as root.

Using the command-line server currently offers more flexibility as to when the data is captured, but requires that you set up and configure filters yourself using the `TraceEvent()` API. The Trace Events Configuration wizard lets you set a variety of different static filters and configure the duration of time that the events are logged for.

For more information on the `tracelogger` utility, see its entry in the `Utilities Reference`. For `TraceEvent()`, see the `Neutrino Library Reference`.
Configuring a target for system profiling

Launching the System Profiler Configuration wizard

➤ In the Target Navigator view, right-click a target, then select Kernel Events Tracing... from the menu.

If you don’t have the Target Navigator view open, choose Window→Show View→Other..., then QNX Targets→Target Navigator.

If you don’t already have a target project, you’ll have to create one:

➤ In the Target Navigator view, right-click and select Add New Target.

You can use this target project for a number of different tasks (debugging, memory analysis, profiling), so once you create it, you won’t have to worry about connecting to your target again. Note also that the qconn target agent must be running on your target machine.

Selecting options in the wizard

The wizard takes you through the process of selecting:

- the location of the captured log file (both on the target temporarily and on the host in your workspace)
- the duration of the event capture
- the size of the kernel buffers
- the event-capture filters (to control which events are captured)
Configuring a target for system profiling

Here are the main fields in this wizard:

**Save Project as**

The name you want to use for the kernel events log file (`.kev`) in your workspace.

---

**Table:**

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logging Configuration</td>
<td></td>
</tr>
<tr>
<td>Enter or select the parent folder</td>
<td>arrow55683</td>
</tr>
<tr>
<td>Save to project as</td>
<td>base64</td>
</tr>
<tr>
<td>Tracing method</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Period of time</td>
</tr>
<tr>
<td>Period length</td>
<td>3.0</td>
</tr>
<tr>
<td>Trace file</td>
<td></td>
</tr>
<tr>
<td>Mode</td>
<td>Save on target then upload</td>
</tr>
<tr>
<td>Filename on target</td>
<td><code>dev/shmem/logfile.kev</code></td>
</tr>
<tr>
<td>Trace statistics file</td>
<td></td>
</tr>
<tr>
<td>Mode</td>
<td>Generate only on the target</td>
</tr>
<tr>
<td>Filename on target</td>
<td><code>dev/shmem/infofile.txt</code></td>
</tr>
<tr>
<td>Buffers</td>
<td></td>
</tr>
<tr>
<td>Number of kernel buffers</td>
<td>32</td>
</tr>
<tr>
<td>Number of output buffers</td>
<td>128</td>
</tr>
</tbody>
</table>
Configuring a target for system profiling

Tracing method, Type (Period of time)
The duration of the capture of events as defined by a time. This is the default.

Tracing method, Period length
Floating-point value in seconds representing the length of time to capture kernel events on the target.

Tracing method, Type (Iterations)
The duration of the capture of events as defined by the number of kernel event buffers.

Tracing method, Number of Iterations
Total number of full kernel event buffers to log on the target.

Trace file, Mode (Save on target then upload)
In this mode, kernel event buffers are first saved in a file on the target, then uploaded to your workspace. This is the default.

Trace file, Filename on target
Name of the file used to save the kernel event buffers on the target.

Trace file, Mode (Stream)
In this mode, no file is saved on the target. Kernel event buffers are directly sent from qconn to the IDE.

Trace statistics File, Mode (Generate only on the target)
The information file is generated only on the target. This is the default.

Trace statistics file, Mode (Do not generate)
No file is generated.
Capturing instrumentation data in event log files

If your target is running QNX 6.2.1, you must use this option instead of “Generate only on the target” because the trace statistics file is not supported under QNX 6.2.1.

Trace statistics File, Mode (Save on target then upload)

The statistical information is first saved in a file on the target, then uploaded to your workspace.

Trace statistics File, Filename on target

Name of the file used to save the statistical information on the target.

Buffers, Number of kernel buffers

Size of the static ring of buffers allocated in the kernel.

Buffers, Number of qconn buffers

Maximum size of the dynamic ring of buffers allocated in the qconn target agent.

Capturing instrumentation data in event log files

Regardless of how your log file is captured, you have a number of different options for how to regulate the amount of information actually captured:

- On/Off toggling of tracing
- Static per-class Off/Fast/Wide mode filters
- Static per-event Off/Fast/Wide mode filters
- User event-handler filters

(For more information, see the SAT User’s Guide.)

The IDE lets you access the first three of the above filters. You can enable tracing (currently done by activating the tracing wizard), and
then select what kind of data is logged for various events in the system.

The events in the system are organized into different classes (kernel calls, communication, thread states, interrupts, etc). You can toggle each of these classes in order to indicate whether or not you want to generate such events for logging.

The data logged with events comes in the following modes:
Viewing and interpreting the captured data

Fast mode A small-payload data packet that conveys only the most important aspects of the particular event. Better for performance.

Wide mode A larger-payload data packet that contains a more complete event definition, with more context. Better for understanding the data.

Class Specific This mode lets you select Disable (no data is collected), Fast, Wide, or Event Specific for each of the following event classes:

- Control Events
- Interrupts
- Process and Thread
- Container
- Communication

Choosing Event Specific lets you select Disable, Fast, or Wide for each event in that class.

Depending on the purpose of the trace, you'll want to selectively enable different tracing modes for different types of events so as to minimize the impact on the overall system. For its part in the analysis of these events, the IDE does its best to work with whatever data is present. (But note that some functionality may not be available for post-capture analysis if it isn’t present in the raw event log. ;-))

Viewing and interpreting the captured data

Once an event file is generated and transferred back to the development host for analysis (whether it was done automatically by the IDE or generated by using `tracelogger` and manually extracted back to the IDE), you can then invoke the System Profiler editor.
Viewing and interpreting the captured data

If you receive a “Could not find target: Read timed out” error while capturing data, it’s possible that a CPU-intensive program running at a priority the same as or higher than `qconn` is preventing `qconn` from transferring data back to the host system.

If this happens, restart `qconn` with the `qconn.prio=` option to specify a higher priority. You can use `hogs` or `pidin` to see which process is keeping the target busy, and discover its priority.

The IDE includes a custom perspective for working with the System Profiler. This perspective sets up some of the more relevant views for easy access.

**The System Profiler editor**

In order to start examining an event file, the easiest way is to name it with a `.kev` (kernel event) extension. Files with this extension are automatically bound to the System Profiler editor.

The System Profiler editor is the center of all of the analysis activity. It provides different visualization options for the event data in the log files:
Viewing and interpreting the captured data

CPU Activity presentation
Displays the CPU activity associated with a particular thread of process. For a thread, CPU activity is defined as the amount of runtime for that thread. For a process, CPU activity is the amount of runtime for all the process’s threads combined.

CPU Usage presentation
Displays the percent of CPU usage associated with all event owners. CPU usage is the amount of runtime that event owners get. CPU usage can also be displayed as a time instead of a percentage.

Element Activity presentation
Displays CPU usage for an individual selected process or thread.

Timeline presentation (the default)
Displays events associated with their particular owners (i.e. processes, threads, and interrupts) along with the state of those particular owners (where it makes sense to do so).

The **Timeline** presentation is the default. To choose one of the other types, right-click in the editor, then select **Display→Toggle**. Then choose one of:

- **CPU Activity**
- **CPU Usage**
- **Element Activity**

For displays other than the **Timeline**, you can display the data using your choice of graph by right-clicking the graph and choosing **Graph Type**. Select one of the graph types from the list:

- **Line Graph**
- **Bar Graph**
- **Histogram**
Viewing and interpreting the captured data

- **Pie Chart**

  3D versions of the graphs are also available, with the exception of the Pie Chart.

  Each of these visualizations is available as a “pane” in a stack of “panes.” Additionally, the visualization panes can be split — you can look at the different sections of the same log file and do comparative analysis.

  All panes of the same stack share the same display information. A new pane inherits the display information of the previous pane, but becomes independent after it’s created.

  To split the display, right-click in the editor, then select **Display→Split**. Note that you can lock two panes to each other.

  From the **Split** submenu, choose the graph you want to display in the new pane:

  - **CPU Activity**
  - **CPU Usage**
  - **Element Activity**
  - **Timeline**

  You can have a maximum of four panes.

  A number of different features are available from within the editor:

  **Event owner selection**

  Clicking on event owners selects them in the IDE. These selected event owners can then be used by other components of the IDE (such as Filters and Find).

  If an owner has children (e.g. a parent process with threads), you’ll see an plus sign beside the parent’s name. To see a parent’s children, click the plus sign.
Viewing and interpreting the captured data

(or press Shift – E to expand all owners, and Shift – C to collapse).

Filters
Event owners and specific events can be filtered out using the Event Owner Filters and Event Filters items in the right-click (context) menu. You can use this filtering feature to significantly cut down on the unwanted event “noise” in the display. Once filtered, the log file can be saved as a new log file (using Save As) to produce a smaller, more succinct log file for further examination.

For example, to view only processes that are sending pulses, right-click in the timeline, then select Event Owner Filters → Show Only → MsgSend Family.

Find
Pressing Ctrl – F (or selecting Edit → Find/Replace) opens a dialog that lets you quickly move from event to event. This is particularly useful when following the flow of activity for a particular event owner or when looking for particular events.

Bookmarks
You can place bookmarks in the timeline editor just as you would to annotate text files. Press the B key to add a bookmark, or right-click in the editor and choose Bookmark from the menu.

These bookmarks show up in the Bookmarks view and can represent a range of time or a single particular event instance.

Cursor tracking
The information from the System Profiler editor is also made available to other components in the IDE such as the Trace Event Log and the Trace Event Statistics views. These views can synchronize with the cursor, event owner selections, and time ranges, and can adjust their content accordingly.
Viewing and interpreting the captured data

IPC representation

The flow of interprocess communication (e.g., messages, pulses) is represented by a vertical arrow between the two elements.

You can toggle IPC tracing on/off by pressing I or clicking this button in the toolbar:

![Turn on the IPC tracing](image)

Display Event Labels

The Display Event Labels button in the toolbar toggles kernel-event labels. I/O events, memory events, etc. are all given labels when this is enabled:

![Display Event Labels](image)

Types of selection

Within the editor, you can select either of the following:

- an element (e.g., a process or thread)
- a point in time

Elements

To select a single element, simply click the element’s name. To unselect an element, press and hold the Ctrl key, then click each selected element’s name.

To select multiple elements, press and hold the Ctrl key, then click each element’s name.
Viewing and interpreting the captured data

**Time**

To select a point in time, click an element on the timeline.

To select a range, click the start point on the timeline, then drag and release at the end point.

Or, select the start point, then hold down the Shift key and select the end point.

**Zooming**

When zooming in, the display centers the selection. If a time-range selection is smaller than the current display, the display adjusts to the range selection (or by a factor of two).

When zooming out, the display centers the selection and adjust by a factor of two.

When using a preset zoom factor (100% to 0.01%), the display centers the current selection and adjust to the new factor.

There are various ways to zoom:

- right-click menu (**Display**→**Zoom**)
- toolbar icons
- hotkeys (+ to zoom in; - to zoom out)

**Scrolling**

You use these keys to scroll through time:

<table>
<thead>
<tr>
<th>To move:</th>
<th>Use this key:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The selection to the left by one event</td>
<td>←</td>
</tr>
<tr>
<td>The selection to the right by one event</td>
<td>→</td>
</tr>
<tr>
<td>The display to the right by one page (horizontal scrollbar thumb size)</td>
<td>Shift – →</td>
</tr>
</tbody>
</table>

*continued...*
Viewing and interpreting the captured data

To move: 

<table>
<thead>
<tr>
<th>To move</th>
<th>Use this key</th>
</tr>
</thead>
<tbody>
<tr>
<td>The display to the left by one page (horizontal scrollbar thumb size)</td>
<td>Shift – ←</td>
</tr>
<tr>
<td>The display to the beginning of the timeline</td>
<td>Shift – Home</td>
</tr>
<tr>
<td>The display to the end of the timeline</td>
<td>Shift – End</td>
</tr>
</tbody>
</table>

You use these keys to scroll through elements:

<table>
<thead>
<tr>
<th>To move the display:</th>
<th>Use this key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up by one element</td>
<td>↑</td>
</tr>
<tr>
<td>Down by one element</td>
<td>↓</td>
</tr>
<tr>
<td>Up by one page (horizontal scrollbar thumb size)</td>
<td>Page Up</td>
</tr>
<tr>
<td>Down by one page (horizontal scrollbar thumb size)</td>
<td>Page Down</td>
</tr>
<tr>
<td>To the top of the element list</td>
<td>Home</td>
</tr>
<tr>
<td>To the bottom of the element list</td>
<td>End</td>
</tr>
</tbody>
</table>

Hovering

When you pause your mouse pointer over an element or an event, you’ll see relevant information (e.g. PID, timestamps, etc.).

Other views in the System Profiler

There are a number of additional components outside of the editor that you can use to examine the event data in more detail:

Trace Event Log view

This view can display additional details for the events surrounding the cursor in the editor. The additional detail includes the event number, time, class, and type, as well as decoding the data associated with a particular event.
Viewing and interpreting the captured data

Trace Search panel

Invoked by Ctrl – H (or via Search → Search...), this panel lets you execute more complex event queries than are possible with the Find dialog.

You can define conditions, which may include regular expressions for matching particular event data content (e.g. all(MsgSend events whose calling function corresponds to mmap()). You can then evaluate these conditions and place annotations directly into the System Profiler editor. The results are shown in the Search view.

Unlike the other search panels in the IDE, the Trace Search panel can search for events only in the currently active System Profiler editor. You use this search panel to build conditions and then combine them into an expression. A search iterates through the events from the active log file and be applied against the expression; “hits” appear in the Search Results view and are highlighted in the System Profiler editor.

By default, the Trace Search panel returns up to 1000 hits. You can change this maximum on the QNX → System Profiler of the Preferences dialog (Window → Preferences).

Condition Statistics, Event Owner Statistics, General Statistics views These views provide a tabular statistical representation of particular events. The statistics can be gathered for the entire log file or for a selected range.
You’ll need to click the **Refresh** button to populate these views with data.

Here’s an example of the General Statistics view:

![General Statistics view](image)

**Properties view**

Shows information about the log file that was captured, such as the date and time as well as the machine the log file was captured on.
Chapter 13
Common Wizards Reference

In this chapter...

Introduction 367
Creating a C/C++ project 369
Creating a target 387
Converting projects 389
Importing projects 402
Introduction

Wizards guide you through a sequence of tasks, such as creating a new project or converting an existing non-IDE project to a QNX C/C++ application or library project.

Wizards aren’t directly connected to any perspective. You can access all the project creation wizards from the main menu by selecting File→New→Other…

In the New Project dialog, the wizards are categorized according to the nature of the project. If you expand C, you’ll see all projects that have a C nature; expand QNX, and you’ll see all the projects with a QNX nature:
Introduction

Notice the overlap: the QNX C Project wizard appears in both C and QNX.

Besides the nature-specific wizards, the IDE also has “simple” wizards that deal with the very basic elements of projects: Project, Folder, and File. These elements have no natures associated with them. You can access these wizards by selecting File→New→Other...→Simple.
Although a project may seem to be nothing other than a directory in your workspace, the IDE attaches special meaning to a project — it won’t automatically recognize as a project any directory you happen to create in your workspace.

But once you’ve created a project in the IDE, you can bring new folders and files into your project folder, even if they were created outside the IDE (e.g. using Windows Explorer). To have the IDE recognize such folders and files:

➤ In the Navigator view, right-click the navigator pane and select **Refresh**.

### Creating a C/C++ project

You use the New Project wizard to create a C or C++ project, which can be one of these varieties:

- QNX C Project (application)
- QNX C++ Project (application)

  A C or C++ application for multiple target platforms. It supports the QNX-specific project structure using **common.mk** files to perform a QNX recursive **make**.
Creating a C/C++ project

If you open a `common.mk` file in the editor, you can toggle the display to reveal hidden internal code by clicking the expand icon in the editor:

QNX C Project (library)
QNX C++ Project (library)

A library that other projects can reference. In most other respects, library projects resemble QNX C/C++ application Projects.

Standard Make C Project
Standard Make C++ Project

A basic C or C++ project that uses a standard `Makefile` and GNU `make` to build the source files. You don’t get the added functionality of the QNX build organization and the `common.mk` file, but these standard projects adapt well to your existing code that you wish to bring into the IDE. (For more about `Makefiles` and the `make` utility, see the Conventions for Makefiles and Directories appendix in the Neutrino Programmer’s Guide.)

As a rule, the IDE provides UI elements to control most of the build properties of QNX projects, but not of Standard Make projects (unless you consider a `Makefile` a “UI element”.)
Creating a C/C++ project

How to create a C/C++ project

To create a C/C++ project:

1. From the menu, select **File → New → Project...**

2. In the left pane, select the project’s nature according to this table:

<table>
<thead>
<tr>
<th>If you want to build a:</th>
<th>Select:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Make C project</td>
<td>C</td>
</tr>
<tr>
<td>QNX C application project</td>
<td>C or QNX</td>
</tr>
<tr>
<td>QNX C library project</td>
<td>C or QNX</td>
</tr>
<tr>
<td>Standard Make C++ application project</td>
<td>C++</td>
</tr>
<tr>
<td>QNX C++ application project</td>
<td>C++ or QNX</td>
</tr>
<tr>
<td>QNX C++ library project</td>
<td>C++ or QNX</td>
</tr>
</tbody>
</table>

3. In the right pane, select the type of project that you want (e.g. **QNX C Project**).

4. Click **Next**.

5. Give your project a name.

Even though the wizard allows it, don’t use any of the following characters in your project name (they’ll cause problems later): `| ! $ ( " ) & ' : ; \ ' * ? [ ] # ~ = % < > { }`

6. Ensure that **Use Default Location** is checked.

7. Select the type (application or library):
Creating a C/C++ project

8  Click Next. The wizard displays the appropriate tabs.

9  Select each tab and fill in the required information. The fields for each tab are described in the “Tabs in the New C/C++ Project wizard” section, below.

10 Click Finish. The IDE creates your new project in your workspace.

If you’re building a library, see below.
Creating a C/C++ project

In the C/C++ Development perspective, you can also access the QNX C/C++ Projects wizards via the New C/C++ Project button:

- QNX C Project
- Standard Make C Project
- Convert to a C/C++ Make Project
- QNX C++ Project
- Standard Make C++ Project

If you're building a library project

You’ll need to choose the type of library you wish to build:
Creating a C/C++ project

Static library (libxx.a)
Combine binary object files (i.e. *.o) into an archive that is directly linked into an executable.

Shared library (libxx.so)
Combine binary objects together and join them so they’re relocatable and can be shared by many processes. Shared libraries are named using the format libxx.so.version, where version is a number with a default of 1. The libxx.so file is a symbolic link to the latest version.
Creating a C/C++ project

Static library for shared objects (**libxxS.a**)

Same as static library, but using position-independent code (PIC). Use this if you want a library that is linked into a shared object. The System Builder uses these types of libraries to create new shared libraries that contain only the symbols that are absolutely required by a specific set of programs.

Shared library without export (**xx.dll**)

A shared library without versioning. Generally, you manually open the library with the `dlopen()` function and look up specific functions with the `dlsym()` function.

**If you’re building a Standard Make C/C++ project**

Since this type of project doesn’t use the QNX recursive multivariant `Makefile` structure, you’ll have to set up your own `Makefile`.

Here’s how to create a simple “Hello World” non-QNX project:

1. Open the New Project wizard.
2. Select Standard Make C (or C++) Project, then click Next.
3. Name your project, then click Finish. The IDE has now created a project structure.

Even though the wizard allows it, don’t use any of the following characters in your project name (they’ll cause problems later): `! $ ( " ) & ' : ; \ ' * ? [ ] # ~ = % < > { }`

4. Now you’ll create a makefile for your project. In the Navigator view, highlight your project, then click the **Create a File** button on the toolbar:

5. Name your file “**Makefile**” and click Finish. The editor should now open, ready for you to create your `Makefile`.

Even though the wizard allows it, don’t use any of the following characters in your project name (they’ll cause problems later): `| ! $ ( " ) & ' : ; \ ' * ? [ ] # ~ = % < > { }`
Creating a C/C++ project

Here’s a sample **Makefile** you can use:

```bash
CC:=qcc

hello: hello.c

all: hello

clean:
    rm -f hello.o hello
```

Use Tab characters to indent commands inside of **make** rules, not spaces.

6. When you’re finished editing, save your file (right-click, then select Save, or click the Save button in the tool bar).

7. Finally, you’ll create your “hello world” C (or C++) source file. Again, open a new file, which might look something like this when you’re done:

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return EXIT_SUCCESS;
}
```

### Tabs in the New C/C++ Project wizard

Depending on the type of project you choose, the New Project wizard displays different tabs:

QNX C or C++ application or library project

Tabs:
- Build Variants
- Projects
- Make Builder
- Error Parsers
Creating a C/C++ project

- Options

Standard Make C or C++ project

Tabs:
- Projects
- Make Builder
- Error Parsers
- Binary Parser
- Paths and Symbols

Build Variants tab

The Build Variants tab lets you choose the platforms to compile executables for:
Creating a C/C++ project

By default, all platforms are enabled. You might want to set your preferences for QNX projects to build only for the specific target platforms you want. To do this, open Window→Preferences→QNX→New Project→Build Variants.

Click the Select All button to enable all of the listed variants, or the Deselect All button to disable all of the listed variants.

You can click the Add button to add a new variant under the currently selected target architecture, or the Delete button to remove the currently selected variant.

You must choose at least one platform to be the default build variant:

1. Select the build variant you want as the default.

2. Click the Default button.

The variant’s name changes to include “- default” and a blue exclamation mark is displayed beside it to indicate that it is now the default build variant.

Projects tab

The Projects tab lets you specify your preferred order of building:
Creating a C/C++ project

For example, if you associate *myProject* with *mySubProject*, the IDE builds *mySubProject* first when you rebuild all your projects. If you change *mySubProject*, the IDE doesn’t automatically rebuild *myProject*.

**Make Builder tab**

The Make Builder tab lets you configure how the IDE handles *make* errors, what command to use to build your project, and when to do a build:
Creating a C/C++ project

Build Setting
If you want the IDE to stop building when it encounters a `make` or compile error, check **Stop on Error**. Otherwise, check **Keep Going On Error**.

Build Command
If you want the IDE to use the default `make` command, check **Use Default**. If you want to use a different utility, uncheck **Use Default** and enter your own command in the Build Command field (e.g. `C:/myCustomMakeProgram`).
Workbench Build Behavior

You can specify how you want the IDE to build your project:

- whenever you save any file in your project
- incremental build (**make all**)
- full rebuild (**make clean all**)

Error Parsers

The Error Parsers tab lets you specify which build output parsers (e.g. Intel C/C++ Compiler Error Parser, CDT GNU Assembler Error Parser, etc.) apply to this project and in which order. To change the order, simply select an item, then use the **Up** or **Down** buttons to position the item where you want in the list.
Creating a C/C++ project

Options tab

The Options tab lets you specify several attributes for the project you’re building:
Creating a C/C++ project

General options

By default, some project properties (e.g. active targets) are local — they’re stored in the .metadata folder in your own workspace. If you want other developers to share all of your project’s properties, then set Share all project properties on. The IDE then stores the properties in a .cdtproject file, which you can save in your version control system so that others may share the project file.
Creating a C/C++ project

Build Options

If you want to profile your application and take full advantage of the QNX Application Profiler, then check **Build with Profiling** (see the Profiling an Application chapter in this guide).

If you want use the QNX Code Coverage tool, then check **Build with Code Coverage** (see the Using Code Coverage chapter in this guide).

If you want the IDE to do more dependency checking than it normally would, then set the **Enhanced dependency checking** option on. Note that this means slower builds, so you may want to turn this off in order to improve build times.

Binary Parser tab

If you’re building a Standard Make C/C++ project, then this tab lets you define which binary parser (e.g. ELF Parser) should be used to deal with the project’s binary objects.
Creating a C/C++ project

Discovery Options tab

If you’re building a Standard Make C/C++ project, then this tab lets you control how include paths and C/C++ macro definitions for this particular project are automatically discovered. Certain features of the IDE (e.g. syntax highlighting, code assistance, etc.) rely on this information, as do source-code parsers.
Creating a C/C++ project

At a later time, you can supply this data using the **Search Paths** item in the project properties.

### C/C++ Indexer tab

If you're building a Standard Make C/C++ project, then this tab lets you control the C/C++ source code indexer. Certain features of the IDE rely on this information.
Creating a target

You must create a Target System Project for every target you want to use with the IDE.

To create a new target:

1  From the menu, select File→New→Project...

2  In the left pane, select QNX.
3 In the right pane, select **QNX Target System Project**.

4 Click **Next**. The New QNX Target System Project wizard appears:

5 Complete the fields described below:

- **Target Name**
  - Type a descriptive name for your QNX Target System Project.

- **Project contents**
  - Check **Use default** to store it in your workspace, or turn this option off and select another location in the **Directory** field.
QNX Connector Selection

Type the target connection in the Hostname or IP and Port fields. If you’re running the IDE on a QNX Neutrino machine running qconn, then check Use local QNX Connector; the IDE automatically fills in the connection information. (If you wish to connect to a different target, you may turn Use local QNX Connector off, and then fill in the fields manually.)

Target Configuration

This section is for a future feature.

Click Finish. Your new QNX Target System Project appears in the Navigator view. When you create a launch configuration, the target is listed under the Main tab in the Target Options pane. Note that you can use the Add New Target button in the Target Options pane to open the New Target System Project wizard.

You can also reach the New Target System Project wizard from within the Target Navigator view (right-click, then select Add New Target).

Converting projects

At various times, you may need to convert non-QNX projects to QNX projects (i.e. give them a QNX nature). For example, suppose another developer committed a project to CVS without the .project and .cdtproject files. The IDE won’t recognize that project as a QNX project when you check it out from CVS, so you’d have to convert it. Or, you may wish to turn a Standard Make C/C++ project into a QNX C/C++ project in order to take advantage of the QNX recursive Makefile hierarchy (a project with a QNX nature causes the IDE to use the QNX make tools and structure when building that project).

The IDE lets you convert many projects at once, provided you’re converting all those projects into projects of the same type.
Converting projects

If you wish to convert a QNX project back into a Standard Make C/C++ project, you can use the Convert C/C++ Projects wizard. From the main menu, select File→New→Other… Expand C, then select Convert to a C or C++ Project.

Converting to a QNX project

To convert a non-QNX project to a QNX project:

1. From the menu, select File→New→Other…
2. Expand QNX.
3. Select Convert to a QNX Project.
Converting projects

5 Select the project(s) you want to convert in the Candidates for conversion field.

6 Specify the language (C or C++).

7 Specify the type of project (application or library).

8 Click Finish. Your converted project appears in the C/C++ Projects view and the Navigator view.

You now have a project with a QNX nature, but you’ll need to make further adjustments (e.g. specify a target platform) via the Properties dialog if you want it to be a working QNX project.

Completing the conversion

The conversion wizard gave your Standard Make project a QNX nature; you now need to use the Properties dialog to fully convert your project to a working QNX project.

To bring up the Properties dialog of a project:

1 In the C/C++ Projects or Navigator view, right-click your project.

2 Select Properties from the context menu. The Properties dialog appears:
In the left pane, select **QNX C/C++ Project**.

Specify the properties you want using the available tabs:

**Options**

See the section “Tabs in the New C/C++ Project wizard” above.

**Build Variants**

See the section “Tabs in the New C/C++ Project wizard” above.

**General**

In the **Installation directory** field, you can specify the destination directory (e.g. **bin**) for the output binary you’re building. (For more information, see the Conventions for Makefiles and Directories appendix in the *Neutrino Programmer’s Guide*.)

In the **Target base name** field, you can specify your binary’s base name, i.e. the name without any prefixes or suffixes. By default, the IDE uses your project name as the executable’s base name. For example, if your project is called “Test_1,” then a debug
version of your executable would be called “Test_1_g” by default.
In the **Use file name**, enter the name of the file containing the usage message for your executable. (For more on usage messages, see the entry for `usemsg` in the *Utilities Reference*.

**Compiler**
See the section “Compiler tab” below.

**Linker**
See the section “Linker tab” below.

**Make Builder**
See the section “Tabs in the New C/C++ Project wizard” above.

**Error Parsers**
See the section “Tabs in the New C/C++ Project wizard” above.

5 When you’ve finished specifying the options you want, click **Apply**, then **OK**. The conversion process is complete.

**Compiler tab**

The Compiler tab changes depending on which of the three categories you select:

- General options
- Extra source paths
- Extra include paths
Converting projects

If you’ve selected General options, the first item you specify is the type of compiler. Currently, the choices are:

- GCC 2.95.3
- GCC 3.3.5
- Intel (icc), if you’ve installed the Intel ICC for QNX Neutrino product

Here you can specify the warning level (0 to 9), i.e. the threshold level of warning messages that the compiler outputs. You can also choose to have the preprocessor output intermediate code to a file; the IDE names the output file your_source_file.i (C) or your_source_file.ii (C++), using the name of your source file as the base name.

For the Optimization level, you can specify four levels: from 0 (no optimization) to 3 (most
Converting projects

In the Stack size field, you can specify the stack size, in bytes or kilobytes.

Definitions field

Here you can specify the list of compiler defines to be passed to the compiler on the command line in the form `-D name [=value]`, but you don’t have to bother with the `-D` part; the IDE adds it automatically.

Other options field

Here you can specify any other command-line options that aren’t already covered in the Compiler tab. For more information on the compiler’s command-line options, see qcc in the Utilities Reference.

Extra source paths

If you want to specify source locations other than your project’s root directory, select this category. Then click the appropriate button to specify the location:

- **Project.** — You can add source from another project in your current workspace. Note that the IDE uses relocatable notation, so even if other team members have different workspace locations, they can all work successfully without having to make any additional project adjustments.
- **QNX target.** — You can add source from anywhere in or below your `${QNX_TARGET}` directory on your host.
- **Disk.** — You can choose to add source from anywhere in your host’s filesystem.

Extra include paths

You can specify a list of directories where the compiler should look for include files. The options
Converting projects

here are the same as for Extra source paths, except that here you can change the order of directories in the list, which can be important if you happen to have more than one header file with the same name.

Linker tab

The Linker tab changes depending on which of the four categories you select:

- General options
- Extra library paths
- Extra libs
- Post-build actions
Converting projects

Export symbol options
This field lets you define the level of final stripping of your binary, ranging from exporting all symbols to removing just the debugger symbols to removing them all.

Generate map file
If you set this option on, the IDE prints a link map to the build console.

Build goal name
Specify the output filename for an application or library project. Note that the name you enter in this field forces the library’s shared-object name to match.

By default, a generated application has the same name as the project it’s built from. A library has prefix of “lib” and a suffix of “.a” or “.so” after the project name. In addition, debug variants of applications and libraries have a suffix of “_g”.

Link against CPP library (valid for C++ projects only)
Select the particular C++ library you want to use. QNX Momentics currently ships with these C++ libraries:

- Default — The standard QNX C++ library, with support for all standard C++ features (exceptions, STL, etc.).
- Dinkum with exceptions and Dinkum without exceptions — The Dinkum C++ library, with support for exceptions or without.
- Dinkum Abridged with exceptions and Dinkum Abridged without exceptions — The Dinkum Abridged C++ library, with support for exceptions or without.
- Dinkum Embedded with exceptions and Dinkum Embedded without exceptions —
Converting projects

The Dinkum Embedded C++ library, with support for exceptions or without.

- **GNU with exceptions** — The GNU G++ Standard Library, with support for exceptions.

Compiling C++ code without support for exceptions usually results in a faster executable.

**Library shared object name**

You can use this field to override the shared-object name used in C/C++ library projects. Note that this doesn’t affect the actual filename.

If you specify a filename in the **Build goal name** field, don’t use the Library shared object name field.

**Library version**

This dropdown list lets you select a version number for both the library’s shared-object name and filename. If this is a library that doesn’t have a version number (e.g. “platform.so”), then select “No.”

Note that you can still set the library version even if **Build goal name** is specified.

**Other options field**

Here you can specify any other command-line options that aren’t already covered in the Linker tab. For more information on the linker’s options, see the entry for **ld** in the *Utilities Reference*.

**Extra library paths**

Select this category if you want to specify locations where the linker should look for import libraries (.so or .a files). Then click the appropriate button to specify the location. (These buttons work the same as those in the Compiler tab when you select **Extra source paths**.)
Extra libraries

Here you can add a list of libraries (.so or .a files) to search for unsatisfied references. For each item in this list, you can define:

- Stripped name, the base name without the **lib** prefix (which **ld** adds automatically) and without the suffix (.so or .a).
- Library type (static or dynamic)
- Debug/Release mode. A “No” or “Yes” in this field indicates whether or not the builder matches the debug or release version of the library with the final binary’s type. For example, if you select “Yes” and you want to link against a debug version of the library, the IDE appends “-_g” to the library’s base name. If you select “No,” then the builder passes (to **ld**) this name exactly as you entered it. So, if you wanted to use a release version of your binary and link against a debug version of the library, you’d specify **MyLibraryName_g** as the name.

Adding a new element to the extra library list automatically adds the directory where this library resides to the **Extra library paths** list (see above), if it’s not already there. But if you remove an item from the list, its parent directory is not automatically removed.

You can add a library in two ways:

- **Add** button — lets you create an empty element and define it manually
- **Add from project** — lets you browse your workspace for the library. Note that when you add a library from your workspace, the IDE uses relocatable notation so other members with different workspace locations can all work successfully without having to make any project adjustments.
Converting projects

Extra object files  This lets you link a project against any object file or library, regardless of the filename.

The file-selection dialog may seem slow when adding new files. This is because the system can’t make assumptions about naming conventions and instead must use a binary parser to determine if a file is an object file or a library.

Note also that the Extra object files option is available for an individual platform only. If a project has more than one active platform, you can’t use this feature. In that case, you can still specify extra object files using the Advanced mode for each platform separately.

Post-build actions

When you select this category and click the Add button, you’ll see a dialog that lets you select one of four predefined post-build actions for your project:

- Copy result to other location
- Move result to other location
- Rename result
- Run other shell command

In the What field, you specify the item (e.g. application) you want to copy or move; in the Where field, you specify the destination. You can use the To Workspace or To Filesystem buttons to locate the place.

If you select Rename result, a New Name field appears for you to enter the name. If you select Other command, enter the shell command in the field.

Note that you can set up more than one post-build action; they’re processed sequentially.
Advanced/regular modes

The Properties dialog can appear in two different modes: regular (default) and advanced.

To activate the advanced mode, press the Advanced button at the bottom of the dialog.

To return to regular mode, press the Regular button.

In advanced mode, you can override various options that were set at the project level for the particular build variant you’ve selected:

- platform (the one specified or all supported platforms)
- build mode (e.g. debug, release, user-defined)
- compiler options
- linker options

For example, you can change the optimization level for a particular C file, specify which set of import libraries to use for a specific architecture, and so on.
Importing projects

During the final build, the IDE merges the options you’ve set for the project’s general configuration with the advanced options, giving priority to the advanced settings.

Importing projects

Use the Import wizard to bring resources into your workspace from a filesystem, ZIP archive, or CVS repository.

To bring up the Import wizard:

➤ Choose File→Import…
Or
Right-click in the Navigator or C/C++ Projects view, then choose Import…
Importing projects

The Import wizard. The Import wizard can import resources from several different sources:

- Existing Container Project into Workspace
- Existing Project into Workspace
- External Features

Chapter 13 • Common Wizards Reference 403
Importing projects

- External Plug-ins and Fragments
- File System
- GCC Coverage Data from Project
- QNX Board Support Package
- QNX mkifs Buildfile
- QNX Source Package
- Team Project Set
- Zip file

**Existing Container Project into Workspace**

To import a container project and its associated C/C++ projects from another workspace:

1. In the Import wizard, choose **Existing Container Project into Workspace** and click the **Next** button.

   The IDE displays the **Import Container Project From File System** panel.
Importing a container project.

2 Enter the full path to an existing container project directory in the **Project contents** field, or click the **Browse...** button to select a container project directory using the file selector.

Click **Next** to continue. The IDE displays the **Select components to install** panel.
Importing projects

Select components to install
Select components of container project. If some of components have error marker, it is possible that wizard just cannot find their location. Click "Location..." button to help wizard to find them.

3 By default, every project reference by the container project is also be imported. To exclude certain projects, expand the project tree and deselect projects you don’t want to import.

Click Finish to import the container project and its subprojects.
Existing Project into Workspace

To copy an existing project from another workspace:

1. In the Import wizard, choose **Existing Project into Workspace** and click the **Next** button.

   The IDE displays the **Import Project From Filesystem** panel.

   ![Import Project From Filesystem Panel](image)

   *Importing an existing project.*
Importing projects

2 Enter the full path to an existing project directory in the **Project contents** field, or click the **Browse...** button to select a project directory using the file selector.

3 Click the **Finish** button to import the selected project into your workspace.

**External Features**

Eclipse developers use this for developing IDE plugins and features.

**External Plugins and Fragments**

Eclipse developers use this for developing IDE plugins and features.

**File System**

To copy files and folders from your filesystem into an existing project in your workspace:

1 In the Import wizard, choose **File System** and click the **Next** button.

The IDE displays the **File system** panel.
2 Enter the full path to the code in the From directory field, or click the Browse... button to select a source directory.

3 Use the Filter Types..., Select All, and Deselect All buttons to control which files are imported.

Click a directory on the left panel to see a list of files in the right panel.
Importing projects

The Select Types dialog lets you filter imported files by selecting one or more extensions.

4 Enter the name of a project or folder in the Into folder field, or click the Browse... button to select one.
Importing projects

This project or folder must already exist before you bring up the Import wizard.

![Import Into Folder dialog box]

*Browsing for a project or folder.*

5 To overwrite existing files, check the **Overwrite existing resources without warning** box.
6 To import only the selected folders, check **Create selected folders only**.

To import the selected folder and all sub folders, check **Create complete folder structure**.

7 Click **Finish** to import the selected resources.

**GCC Coverage Data from Project**

The **GCC Coverage Data from Project** option in the Import wizard lets you import code coverage data from applications that have been run outside of the IDE.

For example, in a self-hosted build environment, if you run a code-coverage-enabled program from the command-line, it writes code-coverage data into a `programname.da` file in the same directory as the program’s code.

To import code-coverage data:

1 In the Import wizard, choose **GCC Coverage Data from Project** and click the **Next** button.

   The IDE displays the **GCC Coverage Import** panel.
Importing GCC coverage data.

2 Enter a code-coverage session name in the Session name field.

3 Enter a project name in the Project field, or click the Browse… button to select a project.

4 Click Next to continue.
   The IDE displays the next panel.
Importing projects

5. To include code-coverage data from referenced projects, select them in the **Referenced projects to include coverage data from** list.

6. To include any comments with the new code-coverage session (such as details about the data you’re importing), enter them in the **Comments for this coverage session** field.

7. Click **Finish** to import the code coverage data as a new session in the **Code Coverage Sessions** view.
QNX Board Support Package

To copy a Board Support Package (BSP) into your workspace:

1. In the Import wizard, choose QNX Board Support Package and click the Next button.

The IDE displays the Import QNX BSP panel.

2. Select an installed BSP from the Known Packages list.
Importing projects

You can also enter the full path to a BSP archive (\.zip file) in the Filename field, or click the Select Package... button to browse to a BSP archive.

Click Next to continue.

The IDE displays the Select Source Projects panel.

Selecting source projects from a BSP archive.

3 All of the projects in the BSP archive are imported by default. Uncheck any projects you don’t need to import. Click Next to continue.
The IDE displays the **Select Working Set** panel.

![Select Working Set panel](image)

Selecting a working set from a BSP archive.

4 To change the working-set name for the imported projects, enter a new working-set name in the **Working Set Name** field, or select one from the drop-down list.

To change the project name’s prefix, enter a new prefix in the **Project Name Prefix** field. This is prepended to the name of each project imported from the BSP archive.
Importing projects

To change the destination directory for the projects, enter a new path in the Directory for Projects field, or click the Browse… button to select one. The default is your IDE workspace.

Click Finish to import the BSP projects.

The IDE imports the selected projects from the BSP archive and displays the Build Projects dialog.

![Build Projects?](image)

Building BSP projects.

5 Click Yes to build all of the BSP projects that were just imported. Click No to return to the IDE.

QNX mkifs Buildfile

The IDE can import the .build files used by mkifs into an existing System Builder project.

To import a mkifs .build file:

1 In the Import wizard, choose QNX mkifs Buildfile and click the Next button.

The IDE displays the Import mkifs Buildfile panel.
Importing projects

2. Enter the full path to a `mkifs .build` file in the **Select the file to import** field, or click the **Browse...** button to select one.

3. Click the **Browse...** button beside **Select the destination project** to select a destination for this import.

The IDE displays the **Select System Builder Project** dialog.
Importing projects

Selecting a destination System Builder project.

4 Select one or more project, then click OK.

The IDE imports the selected .build file’s System Builder configuration.
QNX Source Package

To copy a QNX Source Package into your workspace:

1. In the Import wizard, choose **QNX Source Package** and click the **Next** button.

   The IDE displays the **Import QNX Source Package** panel.

   ![Import QNX Sources](image)

2. Select an installed source package from the **Known Packages** list.

   ![Importing a QNX Source Package](image)
You can also enter the full path to a source package (\.zip file) in the Filename field, or click the Select Package… button to browse to a source package.

Click Next to continue.

The IDE displays the Select Source Projects panel.

3 All of the projects in the source package are imported by default. Uncheck any projects you don’t need to import. Click Next to continue.

The IDE displays the Select Working Set panel.

4 To change the working-set name for the imported projects, enter a new working-set name in the Working Set Name field, or select one from the drop-down list.

To change the project name prefix, enter a new prefix in the Project Name Prefix field. This is prepended to the name of each project imported from the source package.

To change the destination directory for the projects, enter a new path in the Directory for Projects field, or click the Browse… button to select one. The default is your IDE workspace.

Click Finish to import the projects.

The IDE imports the selected projects from the source package and displays the Build Projects dialog.

5 Click Yes to build all of the projects that were just imported. Click No to return to the IDE.
Team Project Set

Team project sets are a convenient way of distributing a collection of projects stored in a CVS server among members of your development team. Create them with the Export wizard.

To import a team project set and the projects it references:

1. In the Import wizard, choose Team Project Set and click the Next button.

The IDE displays the Import a Team Project Set panel.
Importing projects

To create a working-set for the imported projects, check the Create a working set containing the imported projects box, and enter a name for the working-set in the Working Set Name field.

Click Finish to import the projects from the CVS repository.
Zip file

To copy files and folders from a ZIP archive into an existing project in your workspace:

1. In the Import wizard, choose **Zip File** and click the **Next** button.

   The IDE displays the **Zip File** panel.

2. Enter the full path to the ZIP archive in the **From zip file** field, or click the **Browse...** button to select a ZIP archive.

3. Use the **Filter Types**... **Select All**, and **Deselect All** buttons to control which files are imported.
Click a directory on the left panel to see a list of files in the right panel.

The Select Types dialog lets you filter imported files by selecting one or more extensions.

4 Enter the name of a project or folder in the Into folder field, or click the Browse… button to select one.
This project or folder must already exist before you bring up the Import wizard.

To overwrite existing files, check the **Overwrite existing resources without warning** box.
Importing projects

6  To import only the selected folders, check **Create selected folders only**.
To import the selected folder and all sub folders, check **Create complete folder structure**.

7  Click **Finish** to import the selected resources.
Chapter 14
Launch Configurations Reference

In this chapter...
What is a launch configuration? 431
Types of launch configurations 431
Running and debugging the first time 433
Running and debugging subsequent times 437
Setting execution options 439
You must set up a Launch Configuration before you can run or debug a program.

What is a launch configuration?

To run or debug programs with the IDE, you must set up a *launch configuration* to define which programs to launch, the command-line options to use, and what values to use for environment variables. The configurations also define which special tools to run with your program (e.g. Code Coverage tool).

The IDE saves your launch configurations so you can quickly reproduce the particular execution conditions of a setup you’ve done before, no matter how complicated.

Each launch configuration specifies a single program running on a single target. If you want to run your program on a different target, you can copy and modify an existing launch configuration. And you can use the same configuration for both running and debugging your program, provided that your options are the same.

Types of launch configurations

The IDE supports these types of launch configurations:
Types of launch configurations

C/C++ QNX QConn (IP)

If you’re connecting to your target machine by IP, select this configuration (even if your host machine is also your target). You’ll have full debugger control and can use the Application Profiler, Memory Trace, and Code Coverage tools. Your target must be running qconn.

C/C++ QNX PDebug (Serial)

If you can access your target only via a serial connection, select this configuration. Rather than use qconn, the IDE uses the serial capabilities of gdb and pdebug directly. This option is available only when you select Debug.

C/C++ Local

If you’re developing on a self-hosted system, you may create a C/C++ Local launch configuration. You don’t need to use qconn; the IDE launches your program through gdb.

C/C++ Postmortem debugger

If your program produced a dump file (via the dumper utility) when it faulted, you can examine the state of your program by loading it into the postmortem debugger. This option is available only when you select Debug. When you debug, you’re prompted to select a dump file.

PhAB Application

If you wish to run a PhAB application, follow the steps for creating a C/C++ QNX QConn (IP) launch configuration.

The main difference between the C/C++ QNX QConn (IP) launch configurations and the other types is that the C/C++ QNX QConn (IP) type supports the runtime analysis tools (QNX System Profiler and QNX Memory Trace).
Running and debugging the first time

You can use the same launch configuration to run or debug a program. Your choices in the Launch Configurations dialog may cause subtle changes in the dialog but greatly affect such things as:

- options in the dialog
- how the IDE connects to the target
- what tools are available for the IDE to use
Running and debugging the first time

The Run and Debug menu items appear in the C/C++ Development perspective by default, but they may not appear in all perspectives. You’ll need the Run→Run... menu item in order to set up a launch configuration. To bring the menu item into your current perspective:

1. From the main menu, select Window→Customize Perspective.
2. Select the Commands tab.
3. Check the Launch box in the Available command groups list.
4. Click OK.

Debugging a program the first time

To create a launch configuration in order to debug a program for the first time:

1. In the C/C++ Projects or Navigator view, select your project.
2 From the main menu, select Run→Debug... (or, click the Debug icon and select Debug... from the dropdown menu).

3 Select a launch configuration type:

If you’re connecting to your target via IP, select C/C++ QNX QConn (IP). If not, see the “Types of launch configurations” section in this chapter before deciding.

4 Click New. The dialog displays the appropriate tabs.

5 Give this configuration a name.

6 Fill in the details in the various tabs. See the “Setting execution options” section in this chapter for details about each tab.

7 Click Debug. You can now launch and debug your program.
Running and debugging the first time

You can also use the **Debug As** menu item to conveniently select a particular launch configuration:

<table>
<thead>
<tr>
<th>Run</th>
<th>Ctrl+Shift+F11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Last Launched</td>
<td></td>
</tr>
<tr>
<td>Debug Last Launched</td>
<td>F11</td>
</tr>
<tr>
<td>Run History</td>
<td></td>
</tr>
<tr>
<td>Run As</td>
<td></td>
</tr>
<tr>
<td>Run...</td>
<td></td>
</tr>
<tr>
<td>Debug History</td>
<td></td>
</tr>
<tr>
<td><strong>Debug As</strong></td>
<td>1 C/C++ QM: QConn</td>
</tr>
<tr>
<td>Debug...</td>
<td>2 Local C/C++ Application</td>
</tr>
</tbody>
</table>

Running a program the first time

When you configure a program to run, you should also configure it to debug as well.

There are fewer options for running programs than for debugging. Some configurations aren’t available.

To run a program the first time:

- Repeat the procedure for debugging a program (see “Debugging a program the first time”), with the following changes:
  - Instead of selecting **Run→Debug** from the main menu, select **Run→Run...** (or, click the **Run** icon and select **Run...** from the dropdown menu).
  - Instead of clicking **Debug** when you’re done, click **Run**.
  - Instead of running under the control of a debugger, your program simply runs.
You can also use the **Run As** menu item to conveniently select a particular launch configuration:

<table>
<thead>
<tr>
<th>Run</th>
<th>Ctrl+F11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Last Launched</td>
<td></td>
</tr>
<tr>
<td>Debug Last Launched</td>
<td>F11</td>
</tr>
<tr>
<td>Run History</td>
<td></td>
</tr>
<tr>
<td><strong>Run As</strong></td>
<td>1 C/C++ QNX QConn</td>
</tr>
<tr>
<td>Run...</td>
<td></td>
</tr>
<tr>
<td>Debug History</td>
<td></td>
</tr>
<tr>
<td>Debug As</td>
<td></td>
</tr>
<tr>
<td>Debug...</td>
<td></td>
</tr>
<tr>
<td>External Tools</td>
<td></td>
</tr>
</tbody>
</table>

The IDE also lets you run a program without creating a launch configuration, but the program’s output doesn’t appear in the Console view.

To run a program without using the launcher:

1. After building the program, drag the executable from the C/C++ Projects view to a target listed in the Target File System Navigator view. (To learn more about the view, see the “Moving files between the host and target” in the Building OS and Flash Images chapter.)

2. In the Target File System Navigator view, right-click your file and select **Run**. When the dialog appears, click **OK**. Your program runs.

**Running and debugging subsequent times**

Once you’ve created a launch configuration, running or debugging a program is as easy as selecting that configuration. You can do this in several ways:
Running and debugging subsequent times

- fast way: see “Launching a selected program”
- faster way: see “Launching from a list of favorites”
- fastest way: see “Launching the last-launched program”

Launching a selected program

To debug or run a program that you’ve created a launch configuration for:

1. From the main menu, select Run → Debug... or Run → Run...
2. In the left pane, select the launch configuration you created when you first ran or debugged your program.
3. Click Debug or Run.

Launching from a list of favorites

If you have a program that you launch frequently, you can add it to the Debug or Run dropdown menu so you can launch it quickly.

To debug or run a program using your favorites list:

1. Do one of the following:
   - Run: From the main menu, select Run → Run History.
   - Run: Click the dropdown menu (▼) part of the run menu button set ( ).
   - Debug: From the main menu, select Run → Debug History.
   - Debug: Click the dropdown menu (▼) part of the debug menu button set ( ).
You’ll see a list of all the launch configurations you specified in the Display in favorites menu:

2 Select your launch configuration.

Launching the last-launched program

To relaunch the last program you ran or debugged:

➤ Press F11 or click the Debug or Run dropdown button ( ▼ ), then select your Launch configuration.

Setting execution options

The Launch Configurations dialog has several tabs:

- Main
- Arguments
- Environment
- Download
- Debugger
Setting execution options

- Source
- Common
- Tools

All of these tabs appear when you select the C/C++ QNX QConn (IP) type of launch configuration; only some tabs appear when you select the other types.

Main tab

This tab lets you specify the project and the executable that you want to run or debug. The IDE might fill in some of the fields for you:

Different fields appear in the Main tab, depending on the type of configuration you’re creating. Here are descriptions of all the fields:
Setting execution options

Project
Enter the name of the project that contains the executable you want to launch. You may also locate a project by clicking Browse.... You can create or edit launch configurations only for open projects.

C/C++ Application
Enter the relative path of the executable’s project directory (e.g. x86/o/Test1_x86). For QNX projects, an executable with a suffix of _g indicates it was compiled for debugging. You may also locate an available executable by clicking Search....

Target Options
- If you don’t want the IDE to create a “pseudo terminal” on the target that sends terminal output to the Console view on a line-by-line basis, then check the Don’t use terminal emulation on target option. To use terminal emulation, your target must be running the devc-pty utility.
- If you want to filter-out platforms that don’t match your selected executable, then set the Filter targets based on C/C++ Application selection on. For example, if you’ve chosen a program compiled for PowerPC, you’ll see only PowerPC targets and offline targets.
- Select a target from the available list. If you haven’t created a target, click the Add New Target button. For more information about creating a target, see the Common Wizards Reference chapter.

General Options
If you’re creating a C/C++ QNX PDebug (Serial) launch configuration, then you’ll see the Stop in main option, which is set on by default. This means that after you start the debugger, it stops in main() and waits for your input.
Setting execution options

For serial debugging, make sure that the pseudo-terminal communications manager (\texttt{devc-pty}) is running.

Serial Port Options

Here you can specify the serial port (e.g. \texttt{COM1} for Windows hosts; \texttt{/dev/ser1} for Neutrino) and the baud rate, which you select from the dropdown list.

Arguments tab

This tab lets you specify the arguments your program uses and the directory where it runs.

C/C++ Program Arguments

Enter the arguments that you want to pass on the command line. For example, if you want to send the equivalent of \texttt{myProgram -v -L 7}, type \texttt{-v -L 7} in this field. You can put \texttt{-v} and \texttt{-L 7}
Setting execution options

...on separate lines because the IDE automatically strings the entire contents together.

Working directory on target

The option **Use default working directory** is set on by default. This means the executable runs in the `/tmp` directory on your target. If you turn off this option, you can click **Browse...** to locate a different directory.

Environment tab

The Environment tab lets you set the environment variables and values to use when the program launches. For example, if you want to set the environment variable named **PHOTON** to the value `/dev/photon` when you run your program, use this tab. Click **New** to add an environment variable.
**Download tab**

The Download tab lets you tell the IDE whether to transfer an executable from the host machine to the target, or to select one that already resides on the target.

**Executable**

If you select **Download executable to target**, the IDE sends a fresh copy of the executable every time you run or debug.

The **Download directory on target** field shows the default directory of `/tmp` on your target. If you select the **Use executable on target** option, you’ll need to specify a directory here. You can also use the **Browse...** button to locate a directory.

The **Strip debug information before downloading** option is set on by default. Turn it off if you don’t want the IDE to strip the executable you’re downloading to your target.
Setting execution options

The **Use unique name** option is set on by default. This means the IDE makes your executable’s filename unique (e.g. append a number) during each download session.

**Extra libraries**

The **Extra libraries** pane lets you select the shared libraries your program needs. If you click the **Auto** button, the IDE tries to automatically find the libraries needed. If you click **From project**, the IDE looks in your workspace for libraries.

You also have the option of not downloading any shared libraries to your target.

By default, the IDE removes the files it has downloaded after each session. If you don’t want the IDE to “clean up” after itself, then turn off the **Remove downloaded components after session** option.

**Debugger tab**

The Debugger tab lets you configure how your debugger works. The content in the Debugger Options pane changes, depending on the type of debugger you select:
Setting execution options

The settings in the Debugger tab affect your executable only when you debug it, not when you run it.

**Generic debugger settings**

- **Debugger**
  - The debugger dropdown list includes the available debuggers for the selected launch-configuration type. The list also varies depending on whether you’re debugging a remote or a local target.

- **Stop at main() on startup**
  - This option is set on by default. If you turn it off, the program runs until you interrupt it manually, or until it hits a breakpoint.

- **Advanced button**
  - Click the Advanced button to display the Advanced Options dialog.
Setting execution options

Enable these options if you want the system to track every variable and register as you step through your program. Disable the option if you want to manually select individual variables to work with in the Variables view in the debugger (see the Debugging Your Programs chapter). Disabling the Registers option works the same way for the Registers view.

Debugger Options

GDB command file
This field lets you specify a file for running `gdb` using the `-command` option (see the Utilities Reference).

Load shared library symbols automatically
This option (on by default) lets you watch line-by-line stepping of library functions in the C/C++ editor. You may wish to turn this option off if your target doesn’t have much memory; the library symbols take up RAM on the target.

You can use the pane to select specific libraries or use the Auto button to have the IDE attempt to select your libraries.

Stop on shared library events
Choose this option if you want the debugger to break automatically when a shared library or DLL is loaded or unloaded.
Setting execution options

Source tab

The Source tab lets you specify where the debugger should look for source files. By default, the debugger uses the source from your project in your workspace, but you can specify source from other locations (e.g., from a central repository).

To specify a new source location:

1. On the Source tab, click Add... The Add Source Location dialog appears. You may choose to add the source either from your workspace or elsewhere:

   1a. If you wish to add source from your workspace, select Existing Project Into Workspace, click Next, select your project, and then click Finish.

   1b. If you wish to add source from outside your workspace, select File System Directory, and then click Next.
Setting execution options

2 Type the path to your source in the **Select location directory** field or use the **Browse** button to locate your source.

If you want to specify a mapping between directories, choose the **Associate with** option and enter the directory in the available field. For example, if your program was built in the `C:/source1` directory and the source is available in the `C:/source2` directory, enter `C:/source2` in the first field and associate it with `C:/source1` using the second field.

If you want the IDE to recurse down the directories you pointed it at to find the source, then choose the **Search subfolders** option.

3 Click **Finish**. The IDE adds the new source location.

**Common tab**

The Common tab lets you define where the launch configuration is stored, how you access it, and what perspective you change to when you launch.
Setting execution options

Type of launch configuration

When you create a launch configuration, the IDE saves it as a .launch file. If you select Local, the IDE stores the configuration in one of its own plugin directories. If you select Shared, you can save it in a location you specify (such as in your project). Saving as Shared lets you commit the .launch file to CVS, which allows others to run the program using the same configuration.

Display in favorites

You can have your launch configuration displayed when you click the Run or Debug dropdown menus in the toolbar. To do so, check the Run or Debug options under the Display in favorites menu heading.

Launch in background

This is enabled by default, letting the IDE launch applications in the background. This lets you continue to use the IDE while
Setting execution options

waiting for a large application to be transferred to the target and start.

Tools tab

The Tools tab lets you add runtime analysis tools to the launch. To do this, click the Add/Delete Tool button at the bottom of the tab:

You can add the following tools (some launch options affect which tools are available):

Application Profiler

 lets you count how many times functions are called, who called which functions, and so on. For more on this tool, see the Profiling Your Application chapter.
Setting execution options

Memory Analysis

Lets you track memory errors. For more on this tool, see the Finding Memory Errors chapter.
Setting execution options

Code Coverage

Lets you measure what parts of your program have run, and what parts still need to be tested. For more on this tool, see the Code Coverage chapter.
If you want the IDE to open the appropriate perspective for the tool during the launch, then check **Switch to this tool’s perspective on launch.**
Appendix A
Tutorials

In this appendix...

Before you start... 457
Tutorial 1: Creating a Standard Make C/C++ project 457
Tutorial 2: Creating a QNX C/C++ project 460
Tutorial 3: Importing an existing project into the IDE 461
Tutorial 4: Importing a QNX BSP into the IDE 463
Before you start...

Before you begin the tutorials, we recommend that you first familiarize yourself with the IDE’s components and interface by reading the IDE Concepts chapter.

You might also want to look at the core Eclipse basic tutorial on using the workbench in the Workbench User Guide (Help→Help Contents→Workbench User Guide, then Getting started→Basic tutorial).

Tutorial 1: Creating a Standard Make C/C++ project

In this tutorial, you’ll create a simple, Standard Make C/C++ project (i.e. a project that doesn’t involve the QNX recursive Makefile structure).

You use the New Project wizard whenever you create a new project in the IDE. Follow these steps to create a simple “hello world” project:
1. To open the New Project wizard, select File → New → Project... from the main menu of the workbench.

2. In the wizard’s left pane, select C (or C++). In the right pane, select Standard Make C (or C++) Project, then click Next:

3. Name your project (e.g. “MyFirstProject”), then click Finish. The IDE has now created a project structure.

4. Now you’ll create a Makefile for your project. In the Navigator view (or the C/C++ Projects view — it doesn’t matter which), highlight your project.

5. Click the Create a File button on the toolbar:
Name your file "Makefile" and click Finish. The editor should now open, ready for you to create your Makefile. Here’s a sample Makefile you can use:

```bash
CC:=qcc

all: hello

hello: hello.c

clean:
  rm -f hello.o hello
```

Use Tab characters to indent commands inside of Makefile rules, not spaces.

When you’re finished editing, save your file (right-click, then select Save, or click the Save button in the tool bar).

Finally, you’ll create your “hello world” C (or C++) source file. Again, open a new file called hello.c, which might look something like this when you’re done:

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("Hello, world!\n");
  return EXIT_SUCCESS;
}
```

Congratulations! You’ve just created your first Standard Make C/C++ project in the IDE.

For instructions on building your program, see the section “Building projects” in the Developing C/C++ Programs chapter.
In order to run your program, you must first set up a *Neutrino target system*. For details, see:

- the Preparing Your Target chapter
- the section “Running projects” in the Developing C/C++ Programs chapter.

### Tutorial 2: Creating a QNX C/C++ project

Unlike Standard Make C/C++ projects, a QNX C/C++ project relies on the QNX recursive `Makefile` system to support multiple CPU targets. (For more on the QNX recursive `Makefile` system, see the Conventions for Makefiles and Directories appendix in the *Neutrino Programmer’s Guide*.)

Follow these steps to create a simple QNX C (or C++) “hello world” project:

1. In the C/C++ Development perspective, click the **New QNX C Project** (or **New QNX C++ Project**) button in the toolbar:

   ![New Project Wizard](image)

   The New Project wizard appears.

2. Name your project, then select the *type* (e.g. **Application**).

3. Click **Next** – but don’t press Enter! (Pressing Enter at this point amounts to clicking the Finish button, which causes the IDE to create the project for *all* CPU variants, which you may not want.)

4. In the Build Variants tab, check the build variant that matches your target type, such as X86 (Little Endian), PPC (Big Endian), etc. and the appropriate build version (Release or Debug).
In this tutorial, you’ll use the IDE’s Import wizard, which lets you import existing projects, files, even files from ZIP archives into your workspace.

You can use various methods to import source into the IDE. For details, see the chapter Managing Source Code.

Follow these steps to bring one of your existing C or C++ projects into the IDE:

1. Select File→Import... to bring up the Import wizard.

Tutorial 3: Importing an existing project into the IDE

In order to run your program, you must first set up a Neutrino target system. For details, see:

- the Preparing Your Target chapter
- the section “Running projects” in the Developing C/C++ Programs chapter.

Tutorial 3: Importing an existing project into the IDE

Click Finish. The IDE creates your QNX project and displays the source file in the editor.

Congratulations! You’ve just created your first QNX project.

For instructions on building your program, see the section “Building projects” in the Developing C/C++ Programs chapter.

Follow these steps to bring one of your existing C or C++ projects into the IDE:

1. Select File→Import... to bring up the Import wizard.
Tutorial 3: Importing an existing project into the IDE

2 In the Import wizard, choose **Existing Project into Workspace** and click the **Next** button.

The IDE displays the **Import Project From Filesystem** panel.

3 Enter the full path to an existing project directory in the **Project contents** field, or click the **Browse...** button to select a project directory using the file selector.

4 Click the **Finish** button to import the selected project into your workspace.

Congratulations! You’ve just imported one of your existing projects into the IDE.
Tutorial 4: Importing a QNX BSP into the IDE

QNX BSPs and other source packages are distributed as .zip archives. The IDE lets you import both kinds of packages into the IDE:

<table>
<thead>
<tr>
<th>When you import:</th>
<th>The IDE creates a:</th>
</tr>
</thead>
<tbody>
<tr>
<td>QNX BSP source package</td>
<td>System Builder project</td>
</tr>
<tr>
<td>QNX C/C++ source package</td>
<td>C or C++ application or library project</td>
</tr>
</tbody>
</table>

For more information on System Builder projects, see the Building OS and Flash Images chapter.

Step 1: Use File → Import...

You import a QNX source archive using the standard Eclipse import dialog:
Tutorial 4: Importing a QNX BSP into the IDE

As you can see, you can choose to import either a QNX BSP or a “source package.” Although a BSP is, in fact, a package that contains source code, the two types are structured differently and generates different types of projects. If you try to import a BSP archive as a QNX Source Package, the IDE won’t create a System Builder project.

**Step 2: Select the package**

After you choose the type of package you’re importing, the wizard then presents you with a list of the packages found in $QNX_TARGET/usr/src/archives on your host:
Notice that as you highlight a package in the list, a description for that package is displayed.

To add more packages to the list:

1. Click the **Select Package...** button.
2. Select the **.zip** source archive you want to add.
Step 3: Select the source projects

Each source package contains several components (or projects, to use the IDE term). For the package you selected, the wizard then gives you a list of each source project contained in the archive:

![Image of the Import QNX BSP dialog box showing selected source projects]

You can decide to import only certain parts of the source package — simply uncheck the entries you don’t want (they’re all selected by default). Again, as you highlight a component, you’ll see its description in the bottom pane.
Step 4: Select a working set

The last page of the import wizard lets you name your source projects. You can specify:

- Working Set Name — to group all related imported projects together as a set

- Project Name Prefix — for BSPs, this becomes the name of the System Builder project; for other source projects, this prefix allows the same source to be imported several times without any conflicts.
If you plan to import a source BSP and a binary BSP into the IDE, remember to give each project a different name.

**Step 5: Build**

When you finish with the wizard, it creates all the projects and brings in the sources from the archive. It then asks if you want to build all the projects you’ve just imported.

If you answer Yes, the IDE begins the build process, which may take several minutes (depending on how much source you’ve imported).

If you decide not to build now, you can always do a Rebuild All from the main toolbar’s **Project** menu at a later time.

If you didn’t import all the components from a BSP package, you can bring in the rest of them by selecting the System Builder project and opening the import wizard (right-click the project, then select **Import...**). The IDE detects your selection and then extends the existing BSP (rather than making a new one).

**QNX BSP perspective**

When you import a QNX Board Support Package, the IDE opens the QNX BSP perspective. This perspective combines the minimum elements from both the C/C++ Development perspective and the QNX System Builder perspective:
Congratulations! You’ve just imported a QNX BSP into the IDE.
Appendix B

Where Files Are Stored
This appendix shows you where to find key files used by the IDE.

Here are some of the more important files used by the IDE:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Default location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workspace folder</td>
<td>$HOME/workspace</td>
</tr>
<tr>
<td>.metadata folder</td>
<td>$HOME/workspace/.metadata</td>
</tr>
<tr>
<td>Error log</td>
<td>$HOME/workspace/.metadata/.log</td>
</tr>
</tbody>
</table>

On Windows, C:/QNX630 is used instead of the HOME environment variable or the C:/Documents and Settings/username directory (so the spaces in the path name don’t confuse any of the tools).

You can specify where you want your workspace folder to reside. For details, see the section “Running Eclipse” in the Tasks chapter of the Workbench User Guide. (To access the guide, open Help→Help Contents, then select Workbench User Guide from the list.)
Appendix C

Utilities Used by the IDE
This appendix lists the utilities used by the IDE.

Here are the utilities used by the IDE:

- **addr2line** — Convert addresses into line number/file name pairs (GNU)

- **dumper** — Dump the postmortem state of a program (QNX)

- **flashcmp** — Compress files for a Flash filesystem

- **gcc** — Compile and link a program (GNU)

- **gcov** — Gather code coverage data (GNU)

- **gdb** — Debugger (GNU)

- **gprof** — Code profiler (GNU)

- **icc** — Intel C and C++ compiler (x86 only, purchased separately)

- **ld** — Linker command (POSIX)

- **make** — Maintain, update, and regenerate groups of programs (POSIX)

- **mkefs** — Build an embedded filesystem (QNX Neutrino)
- **mkifs** — Build an OS image filesystem (QNX Neutrino)
- **mkimage** — Build a socket image from individual files (QNX Neutrino)
- **mkrec** — Convert a binary image into ROM format (QNX Neutrino)
- **mksbp** — Build a QNX System Builder project
- **objcopy** — Copy the contents of one object file to another (GNU)
- **pdebug** — Process-level debugger
- **pidin** — Display system statistics (QNX Neutrino)
- **procnto** — QNX Neutrino microkernel and process manager (QNX Neutrino)
- **qcc** — Compile command (QNX Neutrino, QNX 4)
- **qconfig** — Query and display QNX installations and configurations
- **qconn** — Provide service support to remote IDE components
- **QWinCfg** — Query and display QNX installations and configurations
- **sendnto** — Send an OS image to a target over a serial or parallel port (QNX 4 and Neutrino)
- **strip** — Remove unnecessary information from executable files (POSIX)
- **tracelogger** — Log tracing information
- **usemsg** — Change the usage message for a command (QNX Neutrino)

For more information, see the *Utilities Reference*. 
Appendix D

Migrating from Earlier Releases

In this appendix...

Introduction  481
From 6.3.0 to 6.3.0 Service Pack 2  482
From 6.2.1 to 6.3.0  489
From 6.2.0 to 6.3.0  493
Introduction

Upgrading from a previous version of the IDE involves two basic steps:

**Step 1** — converting your development workspace to be compliant with the latest version of the IDE framework. The IDE performs this process automatically at startup when it detects an older workspace version.

You can redirect the IDE to point at different workspaces by launching it with this command:

```
qde -data path_to_workspace
```

**Step 2** — converting your individual projects. Depending on which version of the IDE framework you’re migrating from (6.2.0 or 6.2.1), you’ll have to take different steps to convert your projects.

You can easily migrate your old workspace and projects to this release.
From 6.3.0 to 6.3.0 Service Pack 2

In addition to the many fixes and enhancements to the QNX plug-ins, Service Pack 2 introduces a completely new version of the IDE, based on Eclipse 3 and CDT 2.

For a list of new workbench features, see What’s New in 3.0 in the Workbench User Guide (Help→Help Contents→Workbench User Guide→What’s new).

For a list of new CDT features, see What’s new in the CDT? in the C/C++ Development User Guide (Help→Help Contents→C/C++ Development User Guide→What’s new).

In addition to information about migrating your workspace and your projects, this section includes some issues you might run into.

Migrating your workspace

Your workspace is automatically upgraded the first time you launch the new IDE. This process is entirely automated and cannot be prevented. If you might need to revert to an older version of the IDE, be sure to read the Reverting to an older IDE section.

You get an error message on-screen during this process:

![Problems dialog box]

This is caused by internal changes to many of the perspectives commonly used for C/C++ development. You can safely ignore this error.

482 Appendix: D • Migrating from Earlier Releases
To prevent this error from coming up every time you load the IDE (and to prevent a similar error when you exit the IDE):

1 Switch to the IDE workbench, if necessary.
2 Choose Window→Reset Perspective from the menu.
3 Switch to each of your open perspectives, and repeat step 2.

This error reappears later if you open a perspective that’s currently closed, but that had been used at some point in the older IDE. Use this same process to get rid of the error message.

Resetting the existing perspectives also gives you full access to all of the new features available in views that were open in those perspectives.

**Migrating your projects**

Like your existing workspace, your projects are automatically upgraded to take advantage of the new IDE.

To complete the migration of your projects to the new IDE:

1 Right-click your project in the C/C++ Projects view or the Navigation view.
2 Select Properties from the pop-up menu.
3 If your project is a Standard Make C/C++ project, select C/C++ Make Project in the list on the left to display the Make Builder settings:
If your project is a QNX C/C++ project, select **QNX C/C++ Project** in the list on the left, then the **Make Builder** tab to display the **Make Builder** settings:
5 Check the **Clean** box in the **Workbench Build Behavior** group, and enter **clean** in the text field.

6 Click **Apply** to save your settings, or **OK** to save your settings and close the dialog.

7 Repeat this process for each of your projects.

**Migration issues**

- Intel ICC error parser
- File search error
- Reverting to an older IDE
- Missing features in context menus
- System Builder Console doesn’t come to front
- Old launch configurations don’t switch perspective automatically
From 6.3.0 to 6.3.0 Service Pack 2

Intel ICC error parser

If you have the Intel C/C++ Compiler installed, you need to update the Error Parser tab in the Properties dialog for each of your projects using ICC.

1. Right-click your project in the C/C++ Projects view or the Navigator view.

2. Choose Properties from the context menu. The project Properties dialog is displayed.

3. Choose the QNX C/C++ Project entry in the list on the left, then the Error Parsers tab.

In the list of error parsers, you’ll notice a selected blank entry, and an unselected entry for the ICC error parser:

<table>
<thead>
<tr>
<th>Error Parsers</th>
</tr>
</thead>
<tbody>
<tr>
<td>☑ CDT GNU C/C++ Error Parser</td>
</tr>
<tr>
<td>☑ CDT GNU Assembler Error Parser</td>
</tr>
<tr>
<td>☑ CDT GNU Linker Error Parser</td>
</tr>
<tr>
<td>☑ CDT GNU Make Error Parser</td>
</tr>
<tr>
<td>☑ CDT Visual C Error Parser</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>☑ Intel C/C++ Compiler Error Parser</td>
</tr>
</tbody>
</table>

The selected blank entry is for the 6.3.0 ICC error parser, and the new ICC error parser is the 6.3.0 Service Pack 2 error parser.

4. Uncheck the blank entry.

5. Check the Intel C/C++ Compiler Error Parser entry.

6. Click Apply to save your changes, or OK to save your changes and close the dialog.
File search error

If you’re using a 6.3.0 workspace instead of a new 6.3.0 Service Pack 2 workspace, you may get errors when doing a file search (Search→File…):

![Error Message]

You can ignore this error; it doesn’t affect the search results.

To get rid of this error when doing file searches, create a new workspace and import your previous projects into the new workspace.

Reverting to an older IDE

When you load an existing project created with an older version of the IDE, the IDE updates the project to take advantage of new features.
This can cause problems if you try to load the project into an older version of the IDE.

If you plan on reverting to an older version of the IDE, you need to make a backup copy of your workspace before using the new version of the IDE.

Your workspace is located in `C:/QNX630/workspace` under Windows, or `~/workspace` under QNX, Linux, and Solaris.

Don’t use `cp` to back up your workspace under Windows; use `xcopy` or an archiving/backup utility.

**Importing into an older IDE**

You can also import an existing project to an older version of the IDE:

1. Make a backup copy of your workspace.
2. Remove the `.cdtproject` and `.project` files from your project’s directory.
3. Import your project into the older version of the IDE.

**Missing features in context menus**

If you’re missing new features in context menus, such as the ones available in the C/C++ Projects perspective, or if you’re missing standard views, such as the Problems view in the C/C++ Development perspective, you need to reset your perspective.

To reset your perspective, follow the instructions in the “Migrating your workspace” section.

**System Builder Console doesn’t come to front**

By default, the QNX System Builder perspective’s Console view doesn’t automatically switch to the front when building. In the new IDE, changed views change the style of the view title.

If you prefer the old behavior and want the Console view to automatically come to the front during a build:
1 Choose **Window→Preferences** to open the Preferences dialog.

2 Expand the **C/C++** entry in the list, then choose **Build Console** to display the console preferences.

3 Check **Bring console to top when building (if present)**, then click the **OK** button to save your changes and close the Preferences dialog.

**Old launch configurations don’t switch perspectives automatically**

Because of the internal data structure changes, launch configurations created with an older version of the IDE won’t automatically switch to the Debug perspective when used as a debug configuration.

To fix this problem:

1 Choose **Run→Debug...** to open the **Debug** configuration dialog.

2 Change any of the settings for this launch configuration, then click **Apply** to save the change.

3 Change the setting back to the way you had it before, then click **OK** to revert your change and close the dialog.

**From 6.2.1 to 6.3.0**

**Migrating your workspace**

This conversion is a one-way process. Although your data files remain intact, you won’t be able to use this workspace with earlier versions of the IDE.

1 Start the IDE pointing at your 6.2.1 workspace. You’ll see a splash page (“Please wait — Completing the install”), followed by the Different Workspace Version dialog:

---

Appendix: D • Migrating from Earlier Releases  489
From 6.2.1 to 6.3.0

2 Click **OK** to convert your workspace.

3 If a System Builder project exists in your workspace, the Migrate Project dialog is displayed:

   ![Migrate Project dialog](image)

   Click **Yes** to update your System Builder project, or **No** to leave it in the 6.2.1 format. You won’t be able to use the project with the 6.3 IDE unless you update it.

4 Next the Workbench Layout dialog tells you that the layout of some of the views and editors can’t be restored:

   ![Workbench Layout dialog](image)

   This is to be expected, because we upgraded the minor version of installed components, so there may be some UI adjustments. Click **OK**.

   Now you’re ready to migrate your existing projects to 6.3.0.
Migrating your projects

If the 6.3.0 IDE detects any 6.2.1 Standard Make C/C++ projects at startup, you’ll be prompted to convert these projects to the new format:

You must run this conversion process over each 6.2.1 project so it can take full advantage of the new features of the C/C++ Development Tools.

QNX C/C++ projects are automatically converted to the new project format.

Running the conversion wizard

At startup, the conversion wizard automatically checks for projects to convert. Note that you can convert older projects that were never in the workspace (e.g. projects you’ve brought in via a revision control system).

You can access the Make Project Migration wizard at any time:

- Open Window→Customize Perspective…→Other→Update Make Projects.

The IDE then adds an icon (Update Old Make Project) to the toolbar so you can launch the conversion wizard. The icon is activated whenever you select projects that are candidates for conversion.
From 6.2.1 to 6.3.0

The conversion wizard looks like this:
Start the IDE pointing at your 6.2.0 workspace. You’ll see a splash page (“Please wait — Completing the install”), followed by the Different Workspace Version dialog:

From 6.2.0 to 6.3.0
From 6.2.0 to 6.3.0

2 Click OK to convert your workspace.

3 Next the Cannot Preserve Layout dialog tells you that the saved interface layout can’t be preserved:

This is to be expected, because we upgraded the major, incompatible versions of installed components and of the workspace itself. Click OK.

Now you’re ready to migrate your existing projects to 6.3.0.

**Migrating your projects**

The format of 6.2.0 C/C++ projects (including QNX projects) is incompatible with the 6.3.0 format — you must follow these steps to convert your old projects:

1 The initial projects appear in the workspace as non-C/C++ projects. First you must convert each project based on the type of project it was originally stored as:

   - Standard C/C++ projects (which are based on an external build configuration such as a **Makefile**)

494 Appendix: D • Migrating from Earlier Releases
- QNX C/C++ projects (which are based specifically on the QNX multiplatform `Makefile` macros).

  Use the appropriate conversion wizard:

<table>
<thead>
<tr>
<th>For this type of project:</th>
<th>Open this wizard:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard C/C++</td>
<td>File→New→Other…→C→Convert to a C/C++ Make Project</td>
</tr>
<tr>
<td>QNX C/C++</td>
<td>File→New→Other…→QNX→Migrate QNX 6.2.0 Projects</td>
</tr>
</tbody>
</table>

2 Go through this conversion process for each 6.2.0 project so it can take full advantage of the new features of the C/C++ Development Tools. You must also do this for any projects that are stored outside your workspace (e.g. in a revision control system).

☞ Many project options have changed from QNX 6.2.0 to QNX 6.3.0. Although the conversion process attempts to maintain configuration options, you should verify your individual project files to make sure any new settings have been initialized to the values you want.
console
Name for a general view that displays output from a running program. Some perspectives have their own consoles (e.g. C-Build Console, Builder Console).

drop cursors
When you move a “floating” view over the workspace, the normal pointer changes into a different image to indicate where you can dock the view.

Eclipse
Name of a tools project and platform developed by an open consortium of vendors (Eclipse.org), including QNX Software Systems.
The QNX Developer Tools Suite consists of a set of special plugins integrated into the standard Eclipse framework.

editors
Visual components within the workbench that let you edit or browse a resource such as a file.

navigator
One of the main views in the workbench, the Navigator shows you a hierarchical view of your available resources.

outline
A view that shows a hierarchy of items, as the functions and header files used in a C-language source file.

perspectives
Visual “containers” that define which views and editors appear in the workspace.
plugins

In the context of the Eclipse Project, plugins are individual tools that seamlessly integrate into the Eclipse framework. QNX Software Systems and other vendors provide such plugins as part of their IDE offerings.

profiler

A QNX perspective that lets you gather sample “snapshots” of a running process in order to examine areas where its performance can be improved. This perspective includes a Profiler view to see the processes selected for profiling.

project

A collection of related resources (i.e. folders and files) for managing your work.

resources

In the context of the workbench, resources are the various projects, folders, and files that you work with.

In the context of the QNX System Information Perspective, resources are the memory, CPU, and other system components available for a running process to use.

script

A special section within a QNX buildfile containing the command lines to be executed by the OS image being generated.

stream

Eclipse term for the head branch in a CVS repository.

target

Has two meanings:

As a software term, refers to the file that the make command examines and updates during a build process. Sometimes called a “make target.”
As a *hardware* term, refers to the Neutrino-based PC or embedded system that’s connected to the host PC during cross-development.

**tasks**

A view showing the resources or the specific lines within a file that you’ve marked for later attention.

**UI**

User interface.

**views**

Alternate ways of presenting the information in your workbench. For example, in the QNX System Information *perspective*, you have several views available: Memory Information, Malloc Information, etc.

**workbench**

The Eclipse UI consisting of *perspectives*, *views*, and *editors* for working with your *resources*. 
Index

A

Add New Target 234
addr2line command 477
advanced mode (Properties dialog) 401

B

binaries
seeing usage messages for 186
Binary Inspector view 184, 186
Binary Parser tab (New Project wizard) 384

daste file (QNX System Builder) 187
dt project file 18, 75, 112, 122, 383, 389
efs file (QNX System Builder) 195
ifs file (QNX System Builder) 193
kev files (System Profiler) 346
launch file 450
metadata folder 383, 473
project 18, 75, 112, 122, 197, 389
sysbldr meta file 197
zip archives 134

using to override regular options 401
analysis tools
specifying for launch 451
Application Profiler 251–271
specifying for launch 451
Application Profiler editor 267
colors used in 267
Arguments (Process Information view) 328
Arguments tab (launch configurations dialog)
440, 442
assumptions in this guide xxv
autobuild
turning off 79
Index

block coverage 275
  defined 276
bookmarks
  for Help documents 13
Bookmarks view 347
boot order 189
boot script 191
boot scripts 187
branch coverage 275
  currently not provided in the
    IDE 276
  defined 276
breakpoints 164–170
  removing 170
BSPs
  buildfiles ship with 187
  filename conventions in 192
  importing source into the
    IDE 134
BSPs perspective 140
build
  all projects 80
  automatic feature 78
  configurations 128
  CPULIST 83
  executables with code coverage
    enabled 278
  options 83
  order of projects 82
  saving resources before 81
  single projects 81
  terminology 78
  turning off the autobuild
    feature 79
  variants 78
Build Command field (New Project
  wizard) 380
Build Setting field (New Project
  wizard) 380
Build Variants tab (New Project
  wizard) 377
buildfile
  defined 187
  importing a 200

C

C project
  Standard Make, as distinct from
    a QNX multivariant
    project 17
C++ library 397
C/C++ Development
  perspective 71–105
C/C++ editor 71, 89–98
  adding comments to code in 95
  color 96
  font 96
  line numbers 96
  preferences 96
C/C++ Indexer tab (New Projects
  wizard) 386
C/C++ Local launch
  configuration 432
C/C++ Postmortem debugger launch
  configuration 432
C/C++ Projects view 71, 72
  adding standard includes and
    defines 106
  compared with Navigator
    view 73
Index

C/C++ QNX PDebug (Serial) launch configuration 432
C/C++ QNX QConn (IP) launch configuration 432
C/C++ Editor debugging from 157
call counts See also call pairs
call pairs 270
Calls Made pane (Malloc Information view) 330
case
don’t use to distinguish a project or file 17
channels
shown in the System Blocking Graph view 336
clean (build term) 78
code coverage 275–287
  block coverage 275
  branch coverage 275
  combining sessions 284
  defined 275
  enabling 278
  for non-QNX projects 278
icons 283
IDE tool is a visual font end to gcov 275
line-by-line information 284
markers 285
measures quality of tests 275
scan interval 281
should be set for only one build variant at a time 278
summary report 287
viewing reports in a browser 287
Code Coverage Report view 285
Code Coverage Sessions view 282
code coverage tool
  specifying for launch 453
coexistence of OS versions 21
  PhAB 22
  Photon 22
colors
  for signals 336
  in the C/C++ editor 96
Combine images (QNX System Builder) 204
combined image 192, 195
comments
  adding to code in C/C++ editor 95
Common tab (launch configurations dialog) 440, 449
common.mk file
toggle to reveal hidden internal code 370
communications
  IP 61
  serial 61
compiler
  optimization levels 395
  selecting type of 394
  specifying command-line options 395
  specifying defines to be passed to 395
  warning levels 394
Compiler tab (Properties dialog) 393
Condition Statistics view 347
Connection Information view 319, 338
Console view 78
Index

in the Debugging perspective 177
containers
build configurations
  creating 131
  editing 131
build configurations for 131
building 133
  creating 129
  defined 128
  editing configurations for 131
  seeing tree-view of 132
Content Assist 91
conventions
  for filenames in QNX BSPs 192
  for filenames in the QNX System Builder 193
Core Requests pane (Malloc Information view) 330
CPULIST 83
Create image (QNX System Builder) 202
CVS
default source-management system in the IDE 111
CVS Repositories view 114
CVS Repository Exploring perspective 112

D

DDK
  importing source into the IDE 135
debug
  agent 63
Debug perspective
  views in 159
Debug view 152
  selections in drive other views 152
Debug/Release mode
  for adding extra libraries (Linker tab in Properties dialog) 399
debugger 147–179
  options in launch configuration 447
  specifying source locations for 448
Debugger tab (launch configurations dialog) 440, 445
debugging
  assembly-language functions 159
  building an executable for 148
  controls for 154
  session
    controlling a 158
    several programs simultaneously 147
    via C/C++ editor 157
    via hotkeys 156
    without the Debug view 156
devc-pty 441
Dietician (QNX System Builder) 227
Dinkum C++ library 397
disassembly mode (Debugger) 159
Discovery Options tab (New Project wizard) 385
Index

Distribution pane (Malloc
Information view)  330
Download tab (launch configurations
dialog)  440, 444
drag-and-drop  114, 140
dumper command  477

E

Eclipse
consortium  3
documentation on using CVS in
the IDE  112
Platform  3
editors
alternate
aren’t integrated with rest of
IDE  15
inside the IDE  15, 97
outside the IDE  15, 97
C/C++  89, 105
Content Assist feature  91
defined  15
enhanced source
navigation  105
jumping to function source
(F3)  105
opening headers (Ctrl – Shift –
o)  105
QNX System Builder  184
System Profiler  346
Element Statistics view  347
Environment tab (launch
configurations dialog)
440, 443

environment variables
CPULIST  83
MAKEFLAGS  24
QNX_CONFIGURATION  21,
24
QNX_HOST  24
QNX_TARGET  24
setting in launch configurations
dialog  443
TMPDIR  24

Environment Variables pane (Process
Information view)  328
error log file  473
Error Parsers tab (New Project
wizard)  381
errors
markers used to indicate  99
Event Backtrace view  310
executables
building for debugging  148
sending fresh copy of whenever
you run or debug  444
stripping debug information
before downloading  444
unique name of for each
download session  445
Export wizard  141
expressions
evaluating/examining in the
debugger  170
Expressions view  170
extra libraries
adding to launch
configuration  445
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
</table>

**F**

F3 (jump to function source) 105
FDs
  side channels shown for 338
file name conventions
  in QNX BSPs 192
  in the QNX System Builder 193
File System Navigator view 234
files
  associations with editors 97
  bringing into a project folder 369
  created outside the IDE 369
  creating from scratch 98
  exporting 140
  filtering 74
  host-specific 24
  IDE removes from target after downloading 445
  importing 114
  locations in workspace 473
  moving between host and target machines 234
  opening 73
  opening headers 105
  target-specific 24
Filesystem pane (QNX System Builder) 184
Flash filesystem
  appending to an OS image 206
flashcmp command 477
font
  in the C/C++ editor 96
functions
  finishing names of in the editor 91
  jumping to source for 105

<table>
<thead>
<tr>
<th>G</th>
</tr>
</thead>
</table>

gcc command 477
gcov command 477
GDB
  using directly from the IDE 179
gdb command 477
General Resources display (System Resources view) 340
General Statistics view 347
gmon.out file 254
  importing a 262
gprof command 477

<table>
<thead>
<tr>
<th>H</th>
</tr>
</thead>
</table>
heap usage 334
hello world 77
  running 87
help
  creating bookmarks in 13
  hover for function synopsis 94
HTML server within the IDE 11
navigating 11
tips and tricks 13
History pane (Malloc Information view) 331
hosts
host-specific files, location of 24
hovering (in System Profiler editor) 362

**I**

**icc** command 477
icons in Code Coverage Sessions view 283
IDE convenience of 3
help system 11
migrating to the current version 481
starting for the first time 5
starting via icon or menu item 4
starting via the **qde** command 9
**IDE User’s Guide** how to use xxiii
Identification Details pane (Process Information view) 328
image adding files to 214
booting 189
combined 192
combines 195
downloading a 207
final format of 206
Flash filesystem 192, 195
OS 192, 193
properties of 217
types of in QNX System Builder 191
**Images** directory 197
Images pane (QNX System Builder) 184
Import wizard 114
include paths adding standard includes and defines 106
specifying 396
instrumented kernel 347
IOFlags column (Connection Information view) 338
IP communications 62
IPL 189, 190
adding to an image 204

**K**

kernel instrumented 347
performance 347
keyboard shortcuts 19

**L**

launch configurations 86, 431–454
creating for debugging 434
creating for running 436
dialog 439
tabs in 439
for a debuggable executable 149
launch in background 451
list of favorites 438, 450
not automatically deleted when you delete associated projects 89
old, removing 265
types of 431
launcher 19
running a program without 437
ld command 477
libraries
  buttons for adding (Linker tab in Properties dialog) 399
  how to build 373
  optimizing in QNX System Builder 227
  shared 374
  shared without export 375
  specifying locations of for the linker 398
  static 374
  static for shared objects 375
line numbers
  how to show them in the C/C++ editor 96
link map
  generating a 397
linked resources 114
linker
  command-line options 398
  Linker tab (Properties dialog) 396
  log files (System Profiler) 346

M
Main tab (launch configurations dialog) 440
make
  command 477
  targets
    adding 83
    removing 84
Make Builder tab (New Project wizard) 379
Make Project Migration wizard 491
Makefile
  recursive hierarchy 76
MAKEFLAGS 24
Malloc Information view 319, 322, 329, 335
markers
  for errors 99
memory
  changing a process’s errors 291–334
  leaks 300
  management 292
Memory Analysis 299
Memory Analysis Sessions view 305
Memory Analysis tool
  launching a program to use 301
  specifying for launch 452
Memory Information view 319, 322, 332, 334
Memory Problems view 300, 301, 308
  filtering problems 310
Memory Resources display (System Resources view) 341
Memory Trace view 300, 301, 305
filtering events 307
Memory view
customizing 174
Memory view (Debugger) 172
migrating
involves two steps 481
to the current version of the IDE 481–495
mkefs 184
command 477
mkifs 184
command 477
mkimage command 477
mkrec command 477
mksbp 184
command 477
multivariant project
as distinct from a Standard Make C project 17

N
natures 17
Navigator view 71
compared with C/C++ Projects view 73
New Project wizard 77, 369
tabs in 376
New QNX Target System Project wizard 388

O
objcopy command 477
opening headers 105
Options tab (New Project wizard) 382
OS image
components of 189
OS versions
coexistence of 21
PhAB 22
Photon 22
specifying which to build for 23
Outline view 75
Overrides directory 197
must be first in the list of search paths 226

P
padding (QNX System Builder) 196
pathname delimiter in QNX Momentics documentation xxii
pdebug 63, 432
command 477
perspectives
C/C++ Development 71–105
defined 13
govern which views appear 14
specifying which to switch to during launch 454
PhAB 239
editing code in 245
Index

how to close 244
reopening 245
Phindows 67
pidin command 477
platforms
all are enabled by default 378
how to specify which to build for 378
position-independent code (PIC) 375
post-build actions (Linker tab in Properties dialog) 400
preferences 20
C/C++ editor 96
Problems view 78, 98
Process Information view 319, 326
Processes pane (System Summary view) 326
procno 189, 191, 220
command 477
variants 191
procno-instr 347
PROFDIR 261
profiling
a running process 258
building a program for 254
instrumented 253
non-QNX projects 256
per-function 268
per-line 265
per-thread 269
postmortem 253, 261
running and profiling a process 256
running as root 258
sessions
how to control 263
statistical types of 252
programs
creating 72
debugging for the first time 434
relaunching a 439
running 72, 85
running for the first time 436
running/debugging once you’ve created a launch configuration 438
Project Name Prefix (BSP import wizard) 138
project.bld file (QNX System Builder) 187, 197
projects
.cdtproject file 18, 75, 112, 122, 383, 389
.project file 18, 75, 112, 122, 197, 389
building 201
container 128
converting to QNX type 389
creating 76
creating in the QNX System Builder 199
defined 16
deleting 89
does not also delete launch configuration 89
different types of 370
exporting 140
Flash filesystem 200
how to create 371
importing 114
migrating from 6.2.0 to 6.3.0 494
migrating from 6.2.1 to 6.3.0 491

names
don’t use case alone to distinguish 17
don’t use spaces in 17
nesting
not yet supported in Eclipse 116

non-QNX
how to create 375
opening 73
properties of
setting 401
QNX System Builder 197
configuring 213, 223
target system 387

Projects tab (New Project wizard) 378

properties
image (QNX System Builder) 218
item (QNX System Builder) 222

Properties dialog
advanced mode 401
used when converting a project 391

Properties view
in QNX System Builder 214

Q
qcc command 477
qconfig 21
command 477
qconn 432
command 477
qde command 9
location of executable file 9
QNX
recursive Makefile hierarchy 76
QNX Application Profiler
perspective
configuring the IDE to automatically change to 258
QNX BSP Perspective 140
QNX C or QNX C++ Project relies on QNX recursive
Makefile hierarchy 76
QNX C/C++ project as distinct from a Standard Make C/C++ project 17
QNX Code Coverage perspective 284
QNX GDB Console view enabling 178
QNX Memory Analysis perspective 291
switching to automatically 304
QNX Momentics
version of on host must be the same or newer than version on target 62
QNX Neutrino
memory management in 292
robust architecture of 292
QNX System Builder
  editor 184
  Filesystem pane 184
  Images pane 184
  toolbar 185
QNX System Builder
  perspective 183
QNX System Builder project
  creating a 199
QNX Target System Project
  creating a 86
QNX tools
  overview 4
QNX_CONFIGURATION 21, 24
QNX_HOST 24
QNX_TARGET 24
qwincfg command 477

R
rebuild 78
Reductions directory 197, 233
  must be second in the list of
  search paths 226
Registers view 172
regular mode (Properties
dialog) 401
resources
  defined 19
  linked 114
ROM monitor 183, 207, 213
root
  all programs launched from the
  IDE run as 328
Run→Run… menu item 434

S
scan interval (code coverage) 281
scrolling (in System Profiler
editor) 361
search paths (QNX System
Builder) 224
Seek Offset column (Connection
Information view) 338
selection
  types of in the System Profiler
editor 360
sendto 183, 210
  command 477
serial communications 62
serial terminal 207
  choosing a device 208
  communicating with your
target 208
  communication
    parameters 207
    supported control
      sequences 207
    transferring files 209
Set QNX Build Environment
  wizard 106
shared libraries
  deleting reduced versions
  of 233
Shared Libraries view
  (Debugger) 175
shortcuts, keyboard 19
side channels 338

514 Index
Signal Information view 319, 335
signals
  color-coding for 336
  sending to a running process 322
  sending to a suspended program 177
Signals view (Debugger) 175
source code
  exporting
to a .zip file 140
to the filesystem 140
importing into the IDE 114
importing QNX source code 134
  specifying locations of 395
Source tab (launch configurations dialog) 440, 448
spaces
don’t use when naming a project or file 17
stack errors 334
startup code 189, 190
strip command 477
stripped name (Linker tab in Properties dialog) 399
symbols
  loading 175
  stripping from a binary 397
System Blocking Graph view 319, 336
System Builder Console view 202
System Builder Projects view 187
System Information
  perspective 315
  CPU processing load and key terms used in 324
  updating views in 323
views in 319
System Memory pane (System Summary view) 325
System Resources view 319, 339
  selecting which display to see 339
System Specifications pane (System Summary view) 325
System Summary view 319, 324
System Uptime display (System Resources view) 339

T
target (machine) 61
CPULIST 83
Target File System Navigator view
  Add New Target option 234
Target Navigator view 319
  customizing 321
  sending a signal using 322
  using to control Malloc Information and Memory Information views 322
target system project
  creating a 387
target-specific files, location of 24
targets (make) 83
  adding 83
  removing 84
terminal emulation 441
TFTP server 207, 210
  TFTP Server view 183
Thread Details pane
configuring 327
Thread Details pane (Process Information view) 327
Thread Information view 319
tips and tricks (item in Help menu) 13
**TMPDIR** 24
toolbar
  QNX System Builder 185
Tools tab (launch configurations dialog) 440, 451
Total Heap pane (Malloc Information view) 330
Trace Event Log view 347
Trace Search 363
**tracelogger** command 477

**U**
update interval (QNX Application Profiler) 257, 259
Update Old Make Project icon 491
usage message
  displayed for each item in QNX System Builder 184
  for binaries in QNX System Builder 186
**usemsg** command 477
user’s guide
  how to use this xxiii
utilities
  used by the QNX System Builder 202

**V**
variables
  preventing the debugger from reading 163
Variables view 163
views
  Application Profiler 264
  Binary Inspector 184, 186
  Bookmarks 347
  C/C++ editor 71
  C/C++ Projects 71, 72
  Code Coverage Report 285
  Code Coverage Sessions 282
  Condition Statistics 347
  Console 78
  Debug 152
defined 14
  Element Statistics 347
  General Statistics 347
  Navigator 71
  Outline 75
  Problems 78
  System Builder Console 202
  System Builder Projects 187
  Target File System
    Navigator 234
  TFTP Server 183
  Trace Event Log 347

**W**
watchpoints 164–170
  removing 170
wizards 367–393
are categorized according to
  natures 367
  conversion 491
  creating “nature-free” files,
    folders, or projects 368
  creating a new project 369
  how to access 367
  Make Project Migration 491
  New Project 77, 376
  New QNX Target System
    Project 388
  Set QNX Build
    Environment 106
    simple 368
  workbench 10
    menus 11
  Workbench Build Behavior field
    (New Project wizard) 381
  Workbench User Guide 71
    references to CVS in 112
  working directory
    on target machine 443
  Working Set Name (BSP import wizard) 138
  workspace
    .metadata folder 383, 473
    defined 5, 16
    migrating from 6.2.0 to
      6.3.0 493
    migrating from 6.2.1 to
      6.3.0 489
    refreshing 15
    specifying where the IDE should
      look for a 16

X

XIP 220

Z

zooming (in System Profiler editor) 361