QNX® Momentics® Development Suite

IDE 4.0.1 User’s Guide

For Windows®, Linux®, Solaris™, and QNX® Neutrino® hosts
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>About This Guide</strong></td>
<td>xxi</td>
</tr>
<tr>
<td>How to use this guide</td>
<td>xxiii</td>
</tr>
<tr>
<td>Assumptions</td>
<td>xxv</td>
</tr>
<tr>
<td>Note to Windows users</td>
<td>xxvii</td>
</tr>
<tr>
<td>Technical support options</td>
<td>xxvii</td>
</tr>
<tr>
<td><strong>IDE Concepts</strong></td>
<td>1</td>
</tr>
<tr>
<td>What is an IDE?</td>
<td>3</td>
</tr>
<tr>
<td>An IDE for building embedded systems</td>
<td>3</td>
</tr>
<tr>
<td>Starting the IDE</td>
<td>4</td>
</tr>
<tr>
<td>Starting the IDE for the first time</td>
<td>5</td>
</tr>
<tr>
<td>Starting the IDE from the command line</td>
<td>7</td>
</tr>
<tr>
<td>Workbench</td>
<td>8</td>
</tr>
<tr>
<td>Using the QNX Help system</td>
<td>9</td>
</tr>
<tr>
<td>Navigating the Help</td>
<td>11</td>
</tr>
<tr>
<td>Help bookmarks</td>
<td>12</td>
</tr>
<tr>
<td>Tips and tricks</td>
<td>12</td>
</tr>
<tr>
<td>Perspectives</td>
<td>13</td>
</tr>
<tr>
<td>Views and editors</td>
<td>14</td>
</tr>
<tr>
<td>Views</td>
<td>14</td>
</tr>
<tr>
<td>Editors</td>
<td>14</td>
</tr>
<tr>
<td>Projects and workspace</td>
<td>15</td>
</tr>
<tr>
<td>Specifying a workspace location</td>
<td>16</td>
</tr>
<tr>
<td>How the IDE characterizes projects using natures</td>
<td>17</td>
</tr>
<tr>
<td>Host and target computers</td>
<td>18</td>
</tr>
</tbody>
</table>
2 Preparing Your Target 25
Host-target communications 27
   IP communications 28
   Serial communications 30
   qconn over Qnet 31
   Installing the qconn update 31
   Example: Debugging via PPP 32
Connecting with Phindows 35

3 Developing C/C++ Programs 37
The C/C++ Development perspective 39
   Wizards and Launch Configurations 40
Controlling your projects 40
   Opening files 41
   Closing projects 41
   Opening projects 41
   Filtering files 42
      Outlines of source and binary files 43
Creating projects 44
   Choosing an indexer 46
Building projects 47
   Enabling the autobuild feature 47
   Building all projects 49
Cleaning and rebuilding all projects 49
Building selected projects 50
Cleaning selected projects 50
Autosave before building 50
Configuring project build order 51
Creating personal build options 53
Adding a \texttt{use} message 54
Running projects 55
Deleting projects 59
Writing code 59
\hspace{1em} C/C++ editor layout 59
\hspace{1em} Finishing function names 60
\hspace{1em} Inserting code templates 61
\hspace{1em} Adding \texttt{include} directives 64
Hover help 65
Adding and removing comments in code 66
Customizing the C/C++ editor 66
Using other editors 68
Creating files from scratch 69
More development features 70
\hspace{1em} Tracking remaining work 70
Code synopsis 73
Checking your build 74
Accessing source files for functions 76
Opening headers 76

4 Managing Source Code 79
\hspace{1em} CVS and the IDE 81
\hspace{1em} \hspace{1em} Local history feature 82
\hspace{1em} \hspace{1em} Project files (\texttt{.project} and \texttt{.cdtproject}) 82
\hspace{1em} \hspace{1em} Core Eclipse documentation on using CVS in the IDE 82
Importing existing source code into the IDE 84
Projects within projects 85
5 Debugging Programs 117

Introduction 119

Debugging your program 120
   Building an executable for debugging 120
   Launching your program 121

Controlling your debug session 124
   Using the controls 126
   Debug launch controls 130
   Disassembly mode 131

More debugging features 132
   Inspecting variables 132
   Using breakpoints and watchpoints 136
   Evaluating your expressions 142
   Inspecting your registers 143
   Inspecting a process’s memory 144
   Inspecting shared-library usage 147
6 Building OS and Flash Images 153
Introducing the QNX System Builder 155
    Toolbar buttons 158
    Binary Inspector 159
    Boot script files 160
    QNX System Builder projects 161
    The scope of the QNX System Builder 161
Overview of images 162
    The components of an image, in order of booting 162
    Types of images you can create 166
    Project layout 171
    Overview of workflow 173
Creating a project for an OS image 173
Creating a project for a flash filesystem image 175
Building an OS image 175
    Create new image 176
    Combine images 177
Downloading an image to your target 180
    Downloading via a serial link 180
    Downloading via TFTP 183
    Downloading using other methods 185
Configuring your QNX System Builder projects 186
    Managing your images 187
    Configuring image properties 190
    Configuring project properties 198
Optimizing your system 202
    Optimizing all libraries in your image 203
    Optimizing a single library 204
    Restoring a slimmed-down library 205
Moving files between the host and target 205
Moving files to the target 207
Moving files from the target to the host 207

7 Developing Photon Applications 209
What is PhAB? 211
PhAB and the IDE 212
Using PhAB 213
Creating a QNX Photon Appbuilder project 213
Closing PhAB 214
Reopening PhAB 214
Editing code 214
Building a QNX Photon Appbuilder project 215
Importing an existing standalone QNX Photon Appbuilder project 216
Starting Photon applications 217

8 Profiling an Application 219
Introducing the Application Profiler 221
Types of profiling 222
Profiling your programs 223
Building a program for profiling 224
Running and profiling a process 226
Profiling a running process 228
Postmortem profiling 231
Controlling your profiling sessions 233
Understanding your profiling data 235
Usage by line 235
Usage by function 237
Usage by thread 238
Call counts 239

9 Using Code Coverage 243
Code coverage in the IDE 245
   Types of code coverage 245
   How the coverage tool works 246
Enabling code coverage 248
   Enabling code coverage for Standard Make projects 248
   Starting a coverage-enabled program 249
Importing `gcc` code coverage data from a project 252
Associated views 253
Code Coverage Sessions View 254
   Examining data line-by-line 255
Code Coverage Properties View 259
Code Coverage Report View 260
   Generating a report 261
   Changing views 261
   Printing a report 262
   Saving a report 262
   Refreshing a report 262

10 Finding Memory Errors 263
Introduction 265
   Memory management in QNX Neutrino 266
   What the Memory Analysis perspective can reveal 273
Analyzing your program 276
Interpreting errors during memory analysis 282
   Illegal deallocation of memory 286
   NULL pointer dereference 288
   Buffer overflow 291
   Using freed memory 294
   Reading uninitialized memory 296
   Resource (memory) leaks 297
   Functions checked for memory errors during memory analysis 299
Error message summary (memory analysis) 300
Memory analysis GUI flags and corresponding environment variables 302
Using a file to log the trace 304
Analyzing a running program 305
  Launching with debug malloc 306
  Attaching to a running process 306
Analyzing shared objects 307
Associated views 308
Memory analysis session 308
Memory analysis editor 313

11 Getting System Information 331
Introduction 333
What the System Information perspective reveals 334
  Key terms 335
Logging system information 337
  Viewing captured system information 339
Associated views 341
Controlling your system information session 342
Examining your target system’s attributes 346
Watching your processes 348
Examining your target system’s memory (inspecting virtual address space) 355
Tracking heap usage 359
Examining process signals 363
Getting channel information 363
Tracking file descriptors 366
Tracking resource usage 367
Tracking the use of adaptive partitioning 373

12 Analyzing Your System with Kernel Tracing 383
Introducing the QNX System Profiler 385
Before you begin 387
Configuring a target for system profiling 389
  Launching the Log Configuration dialog 390
  Selecting options in the wizard 391
Capturing instrumentation data in event log files 394
Viewing and interpreting the captured data 396
  **Trace Search** panel 399
  Properties view 400
Filtering a profile 400
System Profiler use cases 401
  Locating sources of high CPU usage 401
  Mapping and isolating client CPU load from server CPU load 407
  Examining Interrupt Latency 411
  Locating Events of Interest 418
Associated views 426
System Profiler editor 427
Bookmarks view 435
Client/Server CPU Statistics view 435
Condition Statistics view 436
CPU Migration pane 441
Event Owner Statistics view 442
General Statistics view 443
Raw Event Data pane 443
Overview view 444
Partition Summary pane 445
Thread State Snapshot pane 446
Timeline State Colors pane 447
Trace Event Log view 448
Why Running? pane 449

13 **Common Wizards Reference** 451
Introduction 453
Creating a C/C++ project 456
  How to create a C/C++ project 457
  Tabs in the New C/C++ Project wizard 461
Creating a target 472
Converting projects 474
  Converting to a QNX project 474
  Completing the conversion 475
Importing projects 487
  Importing an existing container project into a workspace 489
  Importing an existing project into a workspace 491
  Importing external features 493
  Importing external plugins and fragments 493
  Importing a file system 493
  Importing GCC coverage data from a project 497
  Importing a QNX Board Support Package 499
  Importing a QNX mkifs Buildfile 503
  Importing a QNX source package 504
  Importing a team project set 505
  Importing an archive file 506

14 Launch Configurations Reference 509
What is a launch configuration? 511
Types of launch configurations 511
Running and debugging the first time 513
  Debugging a program the first time 514
  Running a program the first time 515
Running and debugging subsequent times 517
  Launching a selected program 517
  Launching from a list of favorites 517
  Launching the last-launched program 518
Setting execution options 518
  Main tab 519
  Arguments tab 521
A Tutorials 535
Before you start... 537
Tutorial 1: Creating a Standard Make C/C++ project 537
Tutorial 2: Creating a QNX C/C++ project 540
Tutorial 3: Importing an existing project into the IDE 541
Tutorial 4: Importing a QNX BSP into the IDE 543
  Step 1: Use File→Import... 543
  Step 2: Select the package 544
  Step 3: Select the source projects 545
  Step 4: Select a working set 546
  Step 5: Build 548

B Where Files Are Stored 551

C Utilities Used by the IDE 555

D What’s New 559
What’s new in IDE 4.0.1? 561
  Support for Windows Vista 561
  IDE supports self-hosted Neutrino and Solaris 562
  Eclipse updates 562
  General improvements 562
  Improvements to upgrading your software 562
  Download Center updates 563
  Documentation updates 563
What was new in IDE 4.0? 563
Application Profiler perspective 563
System Profiler perspective 564
Memory Analysis perspective 568
System Information perspective 570
System Builder perspective 572
Managed Make for QNX projects 572
C/C++ development 573
Debugging 581
What was new in the IDE in 6.3.0 SP3? 582
What was new in the IDE in 6.3.0 SP2? 582
  General IDE 582
  C/C++ user interface 589
  C/C++ debug and launch 592
  C/C++ project configuration and build 598
  C/C++ editing and source navigation 602
  QNX Momentics tools 608
What was new in the IDE in 6.3.0 SP? 615

E Migrating from Earlier Releases 617
Introduction 619
From 6.3.0 SP1, SP2, or SP3 to IDE 4.0 620
From 6.3.0 to 6.3.0 Service Pack 2 or Service Pack 3 620
  Migrating your workspace 621
  Migrating your projects 621
  Migration issues 623
From 6.2.1 to 6.3.0 627
  Migrating your workspace 627
  Migrating your projects 629
From 6.2.0 to 6.3.0 631
  Migrating your projects 631

Glossary 633
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 Workbench window. 9
This chapter explains how to set up host-target communications. 27
This chapter shows you how to create and manage your C or C++ projects. 39
The C/C++ Editor. 60
The Tasks view lets you track your tasks. 72
This chapter describes managing source code from within the IDE. 81
The Check Out As wizard. 93
This chapter shows you how to work with the debugger. 119
Use the QNX System Builder to create OS and flash images for your target. 155
Typical boot order. 163
Use the PhAB visual design tool to develop Photon apps. 211
This chapter shows you how to use the application profiler. 221
The QNX Application Profiler perspective. 222
Use the Code Coverage tool to help test your code. 245
Use the QNX Memory Analysis perspective to solve memory problems. 265
Process memory layout on an x86. 268
This chapter shows you how to work with the System Information perspective. 333
Use the System Profiler to analyze your system via instrumentation. 385
The System Profiler’s Go To Event command. 398
The System Profiler’s Client/Server CPU Statistics view. 436
The System Profiler’s CPU Migration pane. 441
The System Profiler’s Raw Event Data view. 444
The System Profiler’s Overview view. 444
The System Profiler’s Partition Summary pane. 446
The System Profiler’s Thread State Snapshot view. 447
The System Profiler’s Timeline State Colors view. 448
The System Profiler’s Why Running? view. 449
This chapter describes the IDE’s wizards. 453
The Import wizard. 488
Importing a container project. 490
Selecting container components to import. 491
Importing an existing project. 492
Importing code from the filesystem. 494
The Select Types dialog lets you filter imported files by selecting one or more extensions. 495
Browsing for a project or folder. 496
Importing resources. 497
Importing GCC coverage data. 498
Referenced projects and comments. 499
Importing a BSP. 500
Selecting source projects from a BSP archive. 501
Selecting a working set from a BSP archive. 502
Importing a mkifs .build file. 503
Importing a QNX Source Package. 504
Importing a Team Project Set. 506
Importing code from a ZIP archive. 507
You must set up a Launch Configuration before you can run or debug a program. 511
Here are several tutorials to help you get going with the IDE.  
537  
This appendix shows you where to find key files used by the IDE.  
553  
This appendix lists the utilities used by the IDE.  
557  
This chapter identifies the changes made to the QNX Momentics IDE.  
561  
The new Progress view, showing the progress of a CVS checkout and a Workspace build background operation.  
584  
You can easily migrate your old workspace and projects to this release.  
619
About This Guide
About This Guide

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

This release of the IDE is based on Eclipse 3.2. 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>Run through the IDE tutorials</td>
<td>Tutorials</td>
</tr>
<tr>
<td>Learn about migrating from earlier versions of the IDE</td>
<td>Migrating from Earlier Releases</td>
</tr>
<tr>
<td>Connect your host and target</td>
<td>Preparing Your Target</td>
</tr>
<tr>
<td>Create projects</td>
<td>Developing C/C++ Programs</td>
</tr>
<tr>
<td>Create Photon projects</td>
<td>Developing Photon Applications</td>
</tr>
<tr>
<td>Compile your code</td>
<td>Developing C/C++ Programs</td>
</tr>
<tr>
<td>Import existing code into the IDE</td>
<td>Managing Source Code</td>
</tr>
<tr>
<td>Debug your program</td>
<td>Debugging Programs</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>
</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.

---

<table>
<thead>
<tr>
<th>To:</th>
<th>Go to:</th>
</tr>
</thead>
<tbody>
<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>
<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>Learn how to use one of the IDE’s wizards</td>
<td>Common Wizards Reference</td>
</tr>
<tr>
<td>Look up a keyboard shortcut</td>
<td>IDE Concepts</td>
</tr>
<tr>
<td>Learn where the IDE stores important files</td>
<td>Where Files Are Stored</td>
</tr>
<tr>
<td>Learn what utilities the IDE uses</td>
<td>Utilities Used by the IDE</td>
</tr>
<tr>
<td>Learn about what’s new in this release</td>
<td>What’s New in the IDE</td>
</tr>
<tr>
<td>Find the meaning of a special term used in the IDE</td>
<td>Glossary</td>
</tr>
</tbody>
</table>
Assumptions

- You’re familiar with the architecture of the QNX Neutrino RTOS.
- You can write code in C or C++.

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

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

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.

Technical support options

To obtain technical support for any QNX product, visit the Support + Services area on our website (www.qnx.com). You’ll find a wide range of support options, including community forums.

QNX, Neutrino, Photon, Photon microGUI, Momentics, and Aviage are trademarks, registered in certain jurisdictions, of QNX Software Systems GmbH & Co. KG. and are used under license by QNX Software Systems International Corporation. All other trademarks belong to their respective owners.
Chapter 1
IDE Concepts

In this chapter...

What is an IDE? 3
Starting the IDE 4
Workbench 8
Using the QNX Help system 9
Perspectives 13
Views and editors 14
Projects and workspace 15
Host and target computers 18
Target agent (the qconn daemon) 18
Launcher 19
Resources 19
Wizards 19
Keyboard shortcuts 19
Preferences 20
Version coexistence 20
What is an IDE?

Welcome to the Integrated Development Environment (IDE), a powerful set of tools in the QNX Momentics 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). Plugins from most vendors should 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 components 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 require that you abandon the standard QNX tools and **Makefile** structure. On the contrary, it relies on those tools. If you continue to build 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 doing.

Starting the IDE

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

- On Solaris, you must start the IDE from the command-line:
  
  ```
  $QNX_HOST/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 because the IDE won’t be able to attach to your Photon session, and it will fail to start.

Starting the IDE for the first time

The first time you start the IDE on Windows, the Workspace Launcher dialog prompts you for a location to store your `workspace`. All of your IDE projects are stored in this directory.

```
Workspace Launcher

Select a workspace

QNX Momentics IDE stores your projects in a folder called a workspace. Choose a workspace folder to use for this session.

Workspace: C:/QNX632/ide4-workspace

Use this as the default and do not ask again

OK Cancel
```

*Selecting a workspace directory.*

By default, the IDE offers to put your workspace in `home_directory/ide4-workspace` on Neutrino, Linux, and Solaris, and `C:/QNX632/ide4-workspace` on Windows.

To store your workspace in another location:
Starting the IDE

Click Browse... and select a directory for your workspace.

To continue loading the IDE, click OK.

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

To change the default workspace location on Neutrino, Linux, and Solaris, launch qde with the -data workspace_path option, where workspace_path is the location to your working directory.

The IDE welcomes you

After you choose a workspace location, the IDE displays a Welcome window with several options that help introduce you to the QNX Momentics IDE:

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image.png" alt="Icon" /></td>
<td><strong>Information</strong>: Provides links to overviews within the IDE, including the Documentation Roadmap, Team Support (an important topic if you use CVS), Workbench Basics, and the C/C++ Online Documentation.</td>
</tr>
<tr>
<td><img src="image.png" alt="Icon" /></td>
<td><strong>Tutorials</strong>: Provides links to the tutorials for building your first QNX applications.</td>
</tr>
</tbody>
</table>
Starting the IDE

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Samples Icon" /></td>
<td><strong>Samples</strong>: Provides links to samples to help you explore QNX Momentics by installing various prefabricated samples.</td>
</tr>
<tr>
<td><img src="image" alt="What's New Icon" /></td>
<td><strong>What’s New</strong>: Provides links to documents describing the new features in this release, and information about migrating from a previous release.</td>
</tr>
<tr>
<td><img src="image" alt="Open Workbench Icon" /></td>
<td><strong>Open Workbench</strong>: Opens the workbench window and minimizes the Welcome page.</td>
</tr>
</tbody>
</table>

You can return to this Welcome window at any time by choosing **Help → Welcome**.

Starting the IDE from the command line

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

1. For Windows, navigate to the directory where the **qde.exe** executable is located (for example, for Windows hosts, \C:/QNX632/host/win32/x86/`usr/qde/eclipse`), and run the following command:
   ```
   qde
   ```
2. For all other hosts (Neutrino, Linux, and Solaris), navigate to the directory where the **qde** script resides. Run this command:
   ```
   ./qde
   ```
Don’t run the `eclipse` command, even though it may seem to work. Always use `qde` instead, because it configures the proper QNX-specific environment.

For details about directing the IDE at a particular `workspace` location, see the topic “Specifying a workspace location” in this chapter.

For more information about 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 start the QNX Momentics IDE, it shows the Welcome window. To advance directly to the workbench, choose the `workbench` icon at the right.
Using the QNX Help system

For details about the Workbench menu, see Reference→User interface information→Workbench menus in the Workbench User Guide. For a basic tutorial about using the workbench UI, see Getting Started→Basic tutorial→The Workbench in the Workbench User Guide.

Using the QNX Help system

The IDE includes a help system that is an HTML server which runs in its own window, separate from the workbench. This means that the help system isn’t a perspective or a view.

To open the IDE Help:
Using the QNX Help system

From the main menu, select Help→Help Contents.

If you’re using an external web browser, the IDE starts a web server running on a random port (alternatively, you can specify a port in the Preferences window using Window→Preferences…→Help→Help Server). You can access the help system through this port using any web browser; you aren’t limited to the browser launched by the IDE.

If your help system uses a random port, opening and closing the help several times can confuse some anti-spyware programs, and they may conclude that the help system is a malignant program trying to do nefarious activities with your computer. At this point, the anti-spyware program will either block the program from opening ports (which will disable the Online Help), or warn you about the strange activity.

Unless you’re extremely low on RAM, keep the Online Help system open until you’re finished using the IDE. This prevents any anti-spyware programs from confusing the IDE with a virus, and also lets you refer to the online documentation quickly whenever you need to.
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. 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 about using the workbench. Although some of the workbench topics are covered briefly here in this IDE User’s Guide, you can find complete documentation about using the workbench in the Eclipse Workbench User Guide.

*QNX Momentics*

The QNX documentation set includes the following titles:

- *A Roadmap to QNX Momentics*
- Dinkum library documentation
- High Availability Toolkit
- Phindows for QNX Neutrino
- Photon Multilingual Input
- Photon microGUI for QNX Neutrino
- Power Management
- System Analysis Toolkit
- Documentation for DDKs, and much, much more

*QNX Momentics IDE User’s Guide*

Featuring this User’s Guide. Describes the QNX Integrated Development Environment, how to set up and start using the tools to build QNX-based target systems, etc.
Using the QNX Help system

QNX Momentics IDE Cheat Sheets

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 selected a “placeholder” title page. Simply expand the title entry to see its contents.

Help bookmarks

To 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 the function.

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 most often is the C/C++ editor.

The *editor area* is a section of the workbench window reserved for editors. 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).
Projects and workspace

You can use a different text editor other than the one included with the IDE; however, you will lose the integration of the various views and perspectives. For example, within the text editor of the IDE, you can:

- set breakpoints and then see them in the Breakpoints view
- assign “to-do” markers on particular lines and see them in the Tasks view
- obtain context sensitive help as you pause your cursor over a function name in your code.

For more information about the features of the text editor included with the IDE, see Concepts→Editors and Reference→Preferences→Text Editors in the Workbench User’s Guide.

To use an alternate editor, we recommend that you:

1. Edit your files outside of the IDE.
2. Ensure that you save your files in the correct workspace location. For example, on Windows configurations, you might use the following location:
   
   `C:/QNX632/ide4-workspace/project_name`

3. To refresh the resources, 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 are generic containers for your source code, Makefiles, and binaries. Before you perform 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.
Projects and workspace

Throughout this guide, we use the term “C/C++” as shorthand to cover both C and C++ projects; however, 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 change that file, and how you change it.

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

Specifying a workspace location

To redirect the IDE to reference different workspaces:

From the directory where the qde.exe executable (Windows) or the qde script (all other hosts) resides, run the following command:

```
./qde -data path_to_workspace
```

where path_to_workspace is the location of your working directory.

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 might cause problems with some tools, such as the make utility.

For Unix-type hosts (such as Solaris, Linux, and QNX Neutrino), filenames are case-sensitive, but for Windows they’re not. For this reason, don’t use case alone to distinguish files and projects. For example, Hello.c and hello.c refer to the same file in Windows, but they are two separate files for a Unix-type system.
How the IDE characterizes projects using *natures*

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.

QNX C or C++ projects assume the QNX recursive *Makefile* hierarchy to support multiple target architectures; Standard Make C/C++ projects do not.

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

The *natures* inform 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 (for example, a list of QNX System Builder projects won’t contain any C++ library projects).

The following table contains 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>
<tr>
<td>QNX C Project</td>
<td>C, QNX C</td>
</tr>
<tr>
<td>QNX C Library Project</td>
<td>C, QNX C</td>
</tr>
<tr>
<td>QNX C++ Project</td>
<td>C, C++, QNX C</td>
</tr>
<tr>
<td>QNX C++ Library Project</td>
<td>C, C++, QNX C</td>
</tr>
<tr>
<td>QNX System Builder Project</td>
<td>QNX System Builder</td>
</tr>
</tbody>
</table>

*continued…*
Host and target computers

The IDE saves these natures and other information in the files called `.project` and `.cdtproject` in each project. To ensure that these natures persist in your source control system, such as CVS, include these files when you commit the 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 computers

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

If you’re running the IDE on a QNX Neutrino PC (self-hosted), your target computer may also be the host computer. In this case, you must continue to run *qconn*, even though your host computer is essentially connected 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 launcher of the IDE what program to run, the target to run it on, the 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, as required.

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 guides you through all of the steps to gather required information before creating a project. For more information about wizards, see the Common Wizards Reference chapter in this guide.

Keyboard shortcuts

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

Reference→Preferences→Keys

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 (in the Window menu) lets you customize the behavior of your environment, such as when to build your projects, which target processors to build for, and how to open new perspectives.

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

Version coexistence

The QNX Momentics 6.3 development suite lets you install and work with multiple versions of Neutrino (from 6.2.1 and later) — you can choose the version of the OS from which you want to build programs.

When you install QNX Momentics, you receive 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 computer, the default location is /etc/qnx.
QWinCfg for Windows hosts

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

Launch QWinCfg using the Start menu (e.g. All Programs→QNX Momentics 6.3.2→Configuration).

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

qconfig utility for non-Windows hosts

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

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

- If you specify the -e option, you can configure the environment for building software for a specific version of the operating system. For example, if you’re using the Korn shell (ksh), you can configure your computer as follows:

  eval ‘qconfig -n "QNX 6.3.2 Install" -e’

In the previous example, notice that you must use the “back tick” character (‘), not the single quote character (‘).

When the IDE starts, it uses your current qconfig choice as the default version of the operating system. If you haven’t specified a version, the IDE chooses an entry from the directory identified by QNX_CONFIGURATION environment variable. If you want to override the IDE, choose an appropriate build target.

Coexistence and PhAB

If you’re going to create Photon applications for QNX Neutrino 6.3.2 and 6.2.1 using PhAB, you need to use the older version of PhAB to create your application resources.
Version coexistence

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:

3. Clear the **Use default** check box.

4. Change the **Path to Photon Appbuilder** to `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 → Preferences**).

2. Select **QNX**.

3. From the **Select Install** list, select the OS version you want to build for.

4. Click **Apply**, then click **OK**.
Environment variables

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

**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**  A 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.
Chapter 2
Preparing Your Target

In this chapter...

- Host-target communications 27
- Connecting with Phindows 35
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*), for more information about *qconn*, see “Target agent (the *qconn* daemon)” in the IDE Concepts chapter.

Ensure that you occasionally check the Download Center on our website for updated versions of *qconn*, or you can use the IDE Software Updates manager (*Help → Software Updates*).
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.

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.

If your target’s qconn is out of date, its listing in the Target Navigator view will notify you to check the target properties:
For more information, see “Installing the \texttt{qconn} update,” later in this chapter.

When you set up a launch configuration, select \texttt{C/C++ QNX QConn (IP)}. (See the Launch Configurations Reference chapter in this guide for more information.)
Host-target communications

The \texttt{pdebug} command must be present on the target system in \texttt{/usr/bin} for all debugging sessions; \texttt{qconn} launches it, as required. The \texttt{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 \texttt{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. Typically, Intel x86-based machines use the \texttt{devc-ser8250} driver.

2. Once the serial driver is running, you’ll see a serial device listed in the \texttt{/dev} directory. To confirm it’s running, enter:

   \begin{verbatim}
   ls /dev/ser*
   \end{verbatim}

   You’ll see an entry such as \texttt{/dev/ser1} or \texttt{/dev/ser2}.

3. Type the following command to start the pseudo-terminal communications manager (\texttt{devc-pty}):

   \texttt{devc-pty \\&}

4. Type the following command to start the debug agent (this command assumes that you’re using the first serial port on your target):

   \texttt{pdebug /dev/ser1 \\&}

   The target is now fully configured.

5. Determine the serial port parameters by entering the following command (again, this command assumes the first serial port):

30 Chapter 2 • Preparing Your Target
Host-target communications

```
stty /dev/ser1
```
This command produces a lot of output. Look for the `baud=baudrate` entry; you’ll need this information to properly configure the host portion of the connection.

When you set up a launch configuration, select C/C++ QNX PDebug (Serial). For information about launch configurations, see the “Launch Configurations Reference” chapter in this guide.

**qconn over Qnet**

Suppose you have two targets running Neutrino, such that:

- The first target can communicate with the IDE host via TCP/IP.
- The second target can communicate with the first target via Qnet.

To connect to the second target with the IDE, all you need to do is start `qconn` on the second target, and instruct it to use the IP stack of the first target, like this:

```
SOCK=/net/firstTargetName qconn
```

If you want to start `qconn` like this every time you boot the second target, add this command to the file named `/etc/rc.d/rc.local`. For more information about starting Neutrino, see the Controlling How Neutrino Starts chapter of the Neutrino *User’s Guide*.

**Installing the qconn update**

After you’ve installed the IDE, you may need to update `qconn` on your target systems to take advantage of some additional features. The IDE will work with older versions of `qconn`, but not all features will be available.

Only users with system administrator privileges can perform updates to `qconn`.

To update `qconn` on your development system:
1. In the IDE, select Help→Software Updates→Qconn Updates...
2. Click OK to let the IDE update qconn on your host.

If you already have the latest version of qconn, or the next time you choose Software Updates→Qconn Updates... from the Help menu, the IDE offers to uninstall the qconn update.

After you update qconn on your Development system, you then need to update the version of qconn on your target system. How you do this depends on your target system; you might have to build a new image, or you might simply have to copy the new version to your target.

For example, to copy a new version of qconn to a target system:

1. Use slay qconn on the target to stop any existing qconn.
2. Copy $QNX_TARGET/target/usr/sbin/qconn to your target system’s /usr/sbin directory.
3. Ensure that the qconn in the target’s /usr/sbin directory is executable; if it isn’t, use chmod +x to make it executable.
4. On the target, launch the new qconn.

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 communicate with each other through a null-modem serial cable.

Setting up your target

To configure your target for PPP:

1. Create a file named options in the directory
   /etc/ppp/options that contains the following lines:
Host-target communications

```
debug
57600
/dev/ser1
10.0.0.1:10.0.0.0
```

The number 57600 refers to the baud rate. You may need to use a different value if you encounter problems using this value.

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

3. Type the following command to start the PPP daemon:
   ```
   pppd
   ```

4. Type the following command to start the `qconn` target agent:
   ```
   qconn
   ```

**QNX Neutrino host**

To configure your QNX Neutrino host for PPP:

1. Create a file named `options` in the directory `/etc/ppp/options` that contains the following lines:
   ```
   debug
   57600
   /dev/ser1
   10.0.0.1:10.0.0.0
   ```

   The number 57600 refers to the baud rate. You may need to use a different value if you encounter problems using this value.

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

3. Type the following command to start the PPP daemon with the `passive` option:
   ```
   pppd passive
   ```
Host-target communications

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:

   ![New Connection Wizard](image)

3. Select Connect directly to another computer, then click Next.
4. When prompted for the role of your target, choose Guest; click Next.
5. Name your connection (e.g. “ppp_biscayne”), and then click Next.
6. When prompted to select a device, choose Communications Port (COM1), and then click Next.
Connecting with Phindows

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

8 If you want Windows to create a desktop shortcut, click the option on the last page of the wizard.

9 Click Finish.

10 In the Connect name_of_target dialog, enter your user ID and password, then select Properties.

11 Select the Options tab.

12 Disable 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 file /etc/inetd.conf and add the following line (or remove the comment character if the line already exists):
   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.
Connecting with 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 39
Controlling your projects 40
Creating projects 44
Building projects 47
Running projects 55
Deleting projects 59
Writing code 59
More development features 70


The C/C++ Development perspective

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

The C/C++ Development perspective is where you develop and build your projects. As mentioned in the IDE 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 doc set: 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

<table>
<thead>
<tr>
<th>For information on the Navigator’s:</th>
<th>See these sections in the Workbench User Guide:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toolbar and icons</td>
<td>Concepts→Views→Navigator view</td>
</tr>
<tr>
<td>Right-click context menu</td>
<td>Reference→User interface</td>
</tr>
<tr>
<td></td>
<td>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 features of the IDE for the first time, you’ll be able to create, build, and run your programs very quickly. For details about wizards and launch configurations, 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 it is configured specifically for C and C++
Controlling your projects

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

- presents the project’s executables as if they reside in a subdirectory called Binaries
- for a library project, presents the project’s libraries as if they reside in subdirectory called Archives
- hides certain files
- includes Build Project and related commands in its right-click menu if auto-build is disabled
- gives an outline of C/C++ source files 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.

**Closing projects**

If you aren’t working on the project but would like to keep it in the IDE, you can close it:

In the C/C++ Projects or Navigator view, right-click your project, then select Close Project.

**Opening projects**

To open a closed project:

In the C/C++ Projects or Navigator view, right-click your project, then select Open Project.

The project opens — you can see it in the C/C++ Projects view.
Controlling your projects

Filtering files

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

1. In the C/C++ Projects view, click the menu dropdown button (_disc).  
2. Select Filters... The C Element Filters dialog appears:

   ![C Element Filters dialog](image)

   - Name filter patterns (matching names will be hidden): 
   - The patterns are separated by comma ("*" = any string, ? = any character)
   - Select the elements to exclude from the view:
     - * (all files)
     - Archive files
     - Closed projects
     - Executable files
     - Non-C projects
     - Non-C resource
     - Object files
     - Shared object files
   - Filter description:

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 itself and shows only the files you haven’t filtered.

Outlines of source and binary files

The C/C++ Projects view shows you an outline of the C/C++ source and header files in your project:

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

The C/C++ Projects view shows you the outlines of binary files as well. You can examine the structure of executables or object files to see the elements that you defined or used in the file, as well as implicitly used elements, 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 information about the QNX recursive Makefile hierarchy, see the Conventions for Makefiles and Directories chapter in the Neutrino Programmer’s Guide.

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

Use the New Project wizard whenever you want to create a new project in the IDE.

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:
Creating projects

(You can use the dropdown menu to create different types of C/C++ projects.)

The New Project wizard appears.

There are actually several ways to open the New Project wizard. For details about the wizards, see the Common Wizards Reference chapter in this guide.

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

Although the wizard allows it, don’t use any of the following characters in your project name:

| ! $ ( ) ‘ : ; \ ’ * [ ] # ~ = % < > { } |

Using these characters in a project name will cause problems later.

3 Click Next.

4 Select the Build Variants tab.

5 If you want to build both debug and release variants of your program, select the build variant that matches your target type, such as X86 (Little Endian) or PPC (Big Endian). If you don’t want to build both debug and release variants, expand the item for your target type and disable Debug or Release.

6 Click Finish. The IDE creates your project and displays the source file in the editor.
Choosing an indexer

The C/C++ search features use an index of symbols (from the source code in your projects, and from external header files) to find the locations of declarations and references.

To specify the indexer:

- for a project, open the project’s Properties dialog, and then select C/C++ Indexer from the list on the left.
- as the default indexer for new projects, choose Window→Preferences…, expand C/C++ in the list on the left, and then choose Indexer.

In either case, you have the following choices:

**Full indexer** Parsed your project in the same manner that a compiler does. It starts at each compilation unit and parses that file and all files that it includes. This provides the most accurate index information. For large projects using complex C++ code, this indexer can slow down the IDE.

**Fast indexer** Similar to the Full indexer, except it parses each header file included in a project only once. For example, if a header file is included by two compilation units, the parsing of the second unit reuses the results of parsing the first unit. This is similar to how precompiled headers work. The indexing of large projects using the Fast indexer uses fewer resources than the Full indexer, but the resulting index isn’t quite as accurate.

**No indexer** Disables indexing completely. This also disables much of the search functionality.
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.

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** Build the project from scratch (i.e. `make clean all`).

### Enabling the autobuild feature

By default, the IDE builds your project only when you tell it to, but it can also build 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.

To enable autobuilding:

1. From the main menu, select **Window→Preferences**.
2. In the left pane, expand **General** and select **Workspace**.
3. In the right pane, enable the **Build automatically** option.
4. Click **OK** to save and apply your preferences.

The IDE now builds your projects every time you change a file or other resource.
C/C++ projects have their own autobuild setting. To turn this on:

1. Right-click the C/C++ project, then choose Properties from the menu.
2. Select C/C++ Make Project, and then select the Make Builder tab:

3. Enable the Build on resource save (Auto Build) option in the Workbench Build Behavior section.
Building projects

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

Building all projects

The IDE lets you manually choose to build 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 build all your open projects:

From the main menu, select Project→Build All.

Cleaning and rebuilding all projects

To rebuild all projects, you should clean the projects first, and then run a build:

1 From the main menu, select Project→Clean... The IDE displays the Clean dialog:
Building projects

2 Select the projects you want cleaned, and check the **Start a build immediately** box.

3 Click **OK**.

**Building selected projects**

To build selected projects:

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

**Cleaning selected projects**

To clean selected projects:

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

**Autosave before building**

The IDE automatically saves all your changed resources before you do a manual build. To turn off this feature:

1 From the main menu, select **Window→Preferences**.
2 In the left pane, expand **General**, and then select **Workspace**.
3 In the right pane, disable the **Save automatically before build** option.

4 Click **OK** to save and apply your preferences.

The IDE no longer saves your resources before it builds your project.

**Configuring project build order**

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

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, expand General, expand Workspace, and select Build Order.
3. Disable the Use default build order option.
4. Select a project from the list, then click the Up or Down buttons to position the project where you want it to appear in the build order list.
5. When you’re done, click Apply, then OK.
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 you invoke 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 from 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 the `Build Make Target`... item. The Build Targets dialog appears.
3. Select your custom target, then click `Build`.

Chapter 3 • Developing C/C++ Programs  53
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 receive 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`, which are 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 entry for `usemsg` in the Utilities Reference.

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. Locate the `USEFILE` entry.
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 as follows:

```
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 command 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. For information about configuring your target, see the “Preparing Your Target” chapter in this guide.

After you build a 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...
Running projects

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 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.
Running projects

5 Type a name for your target.

6 Type 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 project:

---

**Tip**: 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, expand the “Binaries” folder.

2 Right-click on your project’s executable file.

3 Select Run As→C/C++ QNX QConn.
Running projects

4 In the Name field, give your launch configuration a name.
5 In the Target Options section, select your target.
6 Click the Run button.

Your program runs, and the IDE displays its output (if any) in the Console view.
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 the deletion.

2. Decide whether you want to delete only the project framework, or the project and 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 marker bar 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.
Writing code

The C/C++ Editor.

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.

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.
Writing code

2 Press Ctrl-Space. (Or, right-click near the cursor and select Content Assist.) A menu with the available functions appears:

3 You can do one of the following:
   - Continue typing. The list shortens.
   - Scroll with the up and down arrows. Press Enter to select the function.
   - Scroll with your mouse. Double-click a function to insert it.
   - Close the Content Assist window by pressing Esc.

Inserting code templates

The IDE includes another code-completion feature that can insert a template for pre-defined code structures, 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 display at the top of the list:
Writing code

The IDE lets you enable as many of these templates as you require, 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++ → Editor → Templates**.
3. To edit a template, select it, and then click **Edit**.
4 To add your own template, click New. A dialog for adding new templates displays:

5 In the Name field, type a name for this code template.

6 In the Pattern field, type the code used to form the template. Click Insert Variables to insert variables into the template code.

7 Click OK when finished.

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 information:
Writing code

Adding and removing comments in code

You can easily add comments using either the C or C++ style, even to large sections of code. You can add two forward slash characters (//) at the beginning of lines to comment out large sections, even when they have other comment characters, such as /* */.

When you remove comments from 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 remove comments from selected lines — if you highlight a partial line, the editor comments out the entire line, not just the highlighted section.

To comment or remove comments from a block of code:

1. In the C/C++ editor, highlight a section of code to comment or remove comments. For a single line, position your cursor anywhere 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 can do all this using C/C++ Editor or Text Editors preference dialogs.

66 Chapter 3 • Developing C/C++ Programs
Writing code

Changing fonts, background color, or the display line numbers

To change fonts, background color, or show line numbers:

1. Open the Text Editor preferences dialog.
2. Select Window → Preferences.
3. In the left pane, select General → Editors → Text Editors.

Changing tabs into spaces, highlighting, and folding options

To change syntax highlighting, convert tabs into spaces, and change the folding options, open the C/C++ editor preferences dialog:

1. Select Window → Preferences.
2. In the left pane, select C/C++ → Editor.
Writing code

Setting Content Assist options

To set Content Assist options, open the Content Assist preferences dialog:

1. Select Window → Preferences.
2. In the left pane, select C/C++ → Editor → Content Assist.

Using other editors

To use a different text editor than the one that’s included with 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 obtain 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 externally, or within the IDE.

Using an external editor outside of the IDE

You can edit your code with an editor started externally from 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 the changes you’ve made using an external 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).

Using another editor within the IDE

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

If you have multiple associations for the same file extension, the IDE always opens the same editor, which is marked as the default in the list of editors for this extension, but other associations continue to be available. To open a file in an alternative editor, right-click on the file you want to open and select **Open with → My editor**. This change means that you are also changing the default editor for this extension.

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.

For more information about file associations, follow these links in the Eclipse Workbench User Guide: **Reference → Preferences → File Associations**.

**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:
3. Enter (or select) the name of the folder where you want the file to reside.
4. Name your file, then click **Finish**.

Now, you will see an empty text editor window, ready for you to begin working on your new file. Also, in the title bar above the editor, your filename appears highlighted in blue.
More development features

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

Tracking remaining work

The Problems view provides you with a list of errors and warnings related to your projects. These are typically syntax errors, typos, and other programming errors identified by the compiler:

<table>
<thead>
<tr>
<th>Description</th>
<th>Resource</th>
<th>Path</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Errors (4 items)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>structure has no member named 'budget_percent'</td>
<td>partitions.c</td>
<td>partitions</td>
<td>line 14</td>
</tr>
<tr>
<td>structure has no member named 'critical_budget'</td>
<td>partitions.c</td>
<td>partitions</td>
<td>line 15</td>
</tr>
<tr>
<td>structure has no member named 'name'</td>
<td>partitions.c</td>
<td>partitions</td>
<td>line 16</td>
</tr>
<tr>
<td>structure has no member named 'name'</td>
<td>partitions.c</td>
<td>partitions</td>
<td>line 22</td>
</tr>
<tr>
<td>Warnings (1 item)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>warnings: 'partition_ID' might be used uninitialized in this function</td>
<td>partitions.c</td>
<td>partitions</td>
<td>line 9</td>
</tr>
</tbody>
</table>

Error markers

The IDE also shows corresponding markers in several other locations:

- C/C++ Projects view — on both the file that contained the 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 containing the error or warning
More development features

opens in the editor area, with the cursor positioned on the offending line.

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:

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 options you select, the more errors and warnings the IDE detects; all of which display in the Problems view.
More development features

Tracking tasks

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

<table>
<thead>
<tr>
<th>Problems</th>
<th>Console</th>
<th>Properties</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Tasks View" /></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

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.

You can associate a task with an editable resource, for instance, to remind yourself to update a line of source code later:

To associate a task with a resource:

1. In one of the navigation views, double-click the resource that you want to associate the new task. The resource opens in the editor area.
More development features

2 Right-click in the gray border at the left of the editor area, beside the line of text or source code you want to log the new task against.

3 In the pop-up menu, select Add Task.

4 When prompted, enter a brief description about the task.

![New Task dialog](image)

Removing a task

To remove a personal task:

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

Code synopsis

The Outline view provides you with a structural view of your C/C++ source code:
More development features

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:
More development features

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. Whenever you perform a build, you can also have the Console view appear on top of the other stacked views.

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 prototype for that function (if the file is also in your project).

Press Ctrl-F3, and the editor immediately jumps to the function’s implementation (if the file is also in your project).

For more information about the C/C++ Development perspective, see the link Help→Help Contents in the C/C++ Development User Guide.

Opening headers

To open a header file, right-click the file’s name in the Outline view (for example stdio.h), and then choose Open.
More development features

Many of the enhanced source navigation (including opening header files) and code development accelerators available in the C/C++ editor are extracted from the source code. To enable these features and provide the most accurate data representation, the project must be properly configured with the include paths and define directives used to compile the source.

For QNX projects, the standard include paths and definitions are set automatically based on the compiler and architecture. You can set additional values in the project’s properties.

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

To set the include paths and define directives 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 then click **Next**.

The **Compiler/Architecture Selection** panel appears:

![Compiler/Architecture Selection Panel](image)

3 Select the appropriate Compiler, Language, and Architecture for your project, and then click **Finish**.
Chapter 4
Managing Source Code

In this chapter...

CVS and the IDE 81
Importing existing source code into the IDE 84
Using container projects 97
Importing a BSP or other QNX source packages 104
Exporting projects 110
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.
CVS and the IDE

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 information about the IDE’s local history feature, see this link 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 of these files with your project when you commit your changes into source control.

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

The following 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→Creating a CVS repository location</td>
</tr>
<tr>
<td>Check code out of CVS</td>
<td>Tasks→Working in the team environment 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, particularly 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</td>
<td>Tasks→Working in the team environment with CVS→Version control life cycle: adding and ignoring resources</td>
</tr>
<tr>
<td>committed to CVS</td>
<td></td>
</tr>
<tr>
<td>Create and apply a patch</td>
<td>Tasks→Working in the team environment with CVS→Working with patches</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>Track code changes that haven’t been</td>
<td>Tasks→Working with local history,</td>
</tr>
<tr>
<td>committed to CVS</td>
<td>especially the Comparing resources</td>
</tr>
<tr>
<td>View an online FAQ about the CVS</td>
<td>Reference→Team Support with CVSS</td>
</tr>
<tr>
<td>Repository Exploring perspective</td>
<td></td>
</tr>
</tbody>
</table>

Importing existing source code into the IDE

As with many tasks within the IDE, there’s more than one method 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, as well as files from 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 information about linked resources, follow these links in the Workbench User Guide: Concepts→Workbench→Linked resources.

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.
Importing existing source code into the IDE

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

```
all:
    jam -fbuild.jam

clean:
    jam -fbuild.jam clean
```

Projects within projects

If you have an existing source hierarchy that looks something like this:

```
SourceProjectA
  ComponentA
    subdir1
    subdir2
  ComponentB
    SubcomponentC
      subdir3
      subdir4
    SubcomponentD
      subdir5
      subdir6
```

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 the IDE doesn’t support nesting projects as such. So how would you import such a source tree? You would use the following four step process:

1. Step 1: Creating an initial project for your source code
2. Step 2: Creating a new project for each existing project or component in your source code tree
Importing existing source code into the IDE

3 Step 3: Linking the projects to a directory in the source tree
4 Step 4: Building the component project in the linked folder

For information about container projects, see “Using container projects” in this chapter.

Step 1

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

1 Select File→New→Project.…
2 Select the type of project (e.g. Standard Make C project).
3 Name your project (e.g. EntireSourceProjectA).
4 To tell the IDE where the resources reside in the filesystem (since they don’t reside in your workspace), disable the Use Default Location option.
5 In the Location field, type 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:
Step 2

Now, you need to create an individual project (via File→New→Project...) for each of the existing projects (or components) in your source tree. In this example, you create a separate project for each of the following source components:

- ComponentA
- ComponentB
- SubcomponentC
- SubcomponentD
Importing existing source code into the IDE

To create individual projects:

1. Select File→New→Project…
2. Select the type of project (e.g. Standard Make C project).
3. In the Name type a descriptive name for your project (e.g. Project_ComponentA).
4. Enable the Use default location option because you want the IDE to create a project in your workspace for this and all the other components that comprise your project EntireSourceProjectA. In the next step, you’ll link 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, you’ll link each individual project in the IDE to its corresponding directory in the source tree:

To link projects:

1. Select File→New→Folder.
2. Make sure your new project (Project_ComponentA) is selected as the parent folder.
3. Type a name for the folder (e.g. ComponentA).
4. Click Advanced>>.
Importing existing source code into the IDE

5. Enable the **Link to folder in the file system** option.

6. Type the path to that folder in your source tree (or use **Browse... to locate and select one**).

7. Click **Finish**. Now, your `Project_ComponentA` project should show a folder called `ComponentA`; the contents of which actually reside in your source tree.

**Step 4**

Now, you 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.
Importing existing source code into the IDE

2 Select C/C++ Make Project.

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

Now, when you start 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. If you change a file or other resource in one location, the duplicate resource is also affected. For example, 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!

**Files System 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 from Windows Explorer, then drag them into the project. The files are copied into your project workspace.

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

**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

Importing 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:
   2a Right-click the new project in the Navigator or C/C++ Projects view and choose Properties.
   2b Click the Build Variants tab, which displays a warning: At least one platform should be selected
   2c 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

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:
   - as a Standard Make C/C++ Project – if you need to create your own Makefile to integrate with an existing build process

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

4 Click Next to continue.

5 If you’re creating or checking out a QNX project:

5a Right-click the new project in the Navigator or C/C++ Projects view and choose Properties.

5b Click the Build Variants tab, which displays a warning:

At least one platform should be selected

5c Select one or more of the build variants, then click OK.

6 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.

2 Choose Check out into an existing project, and then click Next. The IDE displays the Check Out Into dialog:
Importing existing source code into the IDE

3 Select an existing project from the list, and 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
- a QNX Source Package
Importing existing source code into the IDE

- 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 method of linking resources 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 select either New→File or New→Folder.
   The New File or New Folder dialog appears.

2. In the Name field, type a new name for the file or folder.

3. Click Advanced >>, and enable either the Link to file in the file system or Link to folder in the file system option.

4. Type the full path to the file or folder, or click Browse... to select a specific file or folder.

5. Click Variables... to define any path variables for use in the file or folder path:
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. The IDE doesn’t support the creation of projects in projects.

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 build for each of those projects.

Each build configuration may contain a different list and combination 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. Click Next.
3. Name the container.
4. Click Next.
5. In the New Project dialog, click Add Project....
6. Now select all the projects (which could be other containers) that you want to include in this container:
Using container projects

Each subproject has an entry for make targets under the Target field. You can click on an entry to get a menu that lets you change the selection. 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.

7 If the project is a QNX C/C++ project, you can click in its Variant entry to 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
Using container projects

just the variants you’ve selected). Note that the concept of variants makes sense only for QNX C/C++ projects.

8 If you wish, click in the **Stop on error** column to control whether the build process for the container stops at the first subproject to have an error or continues to build all the remaining subprojects.

9 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. Note that the working set and the container have the same name.

If you later add elements to or remove elements from a container project, the working set isn’t updated automatically.

10 Click **Finish**. The IDE creates your container project.

To select a working set, click the down-arrow at the top of the C/C++ Projects view pane, and then select the working set you want.

**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.

100 Chapter 4 ● Managing Source Code
Using container projects

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:

- 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 to build
Using container projects

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.

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.
Using container projects

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** to save your changes and close the dialog.

6 Click **Build** or **Cancel** in the Build Container Configuration 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.

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 are 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 a:</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>

**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 generate 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.
2. Select the .zip source archive you want to add.
Importing a BSP or other QNX source packages

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.

Step 4: Select a working set

The last page of the import wizard lets you name your source projects. You can specify:
Importing a BSP or other QNX source packages

- 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.

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

If you import dual-endian BSPs, the wizard displays this informational message:

Some CPU folders in the project tree that were created as part of the import process have extra content (platform-specific source files and/or headers such as primcell.h).

If you add other platforms to your BSP project later, you should probably also provide extra platform-dependent content.

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 an archive file 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.
- Export to an archive file using the **Export...** command.

### Using the Export... command

The Export wizard helps you export an entire project to your filesystem or an archive file.

To export one or more projects:

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

    The Export wizard appears:
To export your project to the filesystem, choose **General**→**File System**. To export your project to an archive file, choose **General**→**Archive File**. Click **Next**.

The Export wizard’s next panel appears:
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:

3
Exporting projects

Click one or more extensions, then click **OK** to filter the selected files in the Export wizard.

4. Type the name of the directory (if you’re exporting to the filesystem) or file (if you’re exporting to an archive file) in the input field.

5. When you’re done, click **Finish**.
If you export more than one project, and you import from the resulting filesystem or archive file, you’ll get one project containing all of the projects you exported.
Chapter 5
Debugging Programs

In this chapter...
Introduction 119
Debugging your program 120
Controlling your debug session 124
More debugging features 132
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 that original source. In the IDE, you’ll see different icons: a running
Debugging your program

man for executables that weren’t compiled for debugging, or a bug for those that were.

The IDE debugger uses GDB as the underlying debug engine. It translates each GUI action into a sequence of GDB commands, and 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 much more information and control by building debug variants of the executables. To build an executable with debugging information, you must pass the -g option to the compiler. If you’re using a QNX Make project, the filename for the debug variant has _g appended to it.

To specify the -g option from the project options:

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.

### Launching your program

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:
Debugging your program

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.
Debugging your program

For Neutrino, if you want to perform a local debug of your project, you can’t use the default QNX GDB debugger option. Instead, select GDB Debugger from the Debugger list.

4  Make sure Stop at main() on startup is set.

5  Click Apply.

6  Click Debug.

If launching a debugging session doesn’t work when connected to the target with qconn, ensure that pdebug is on the target, and it is located in one of the directories in the PATH that qconn uses (typically /usr/bin).
Controlling your debug session

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.

- The IDE removes terminated debugging sessions from the Debug view when you launch a new session. This frees resources on your development host and your debugging target. You can retain the completed debug sessions by unchecking the Remove terminated launches when a new launch is created box in 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 the 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:
Controlling your debug session

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

<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>
Controlling your debug session

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.

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.
## 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</td>
<td>![Resume Icon]</td>
<td>F8</td>
<td>Run the process freely from the current point.</td>
</tr>
<tr>
<td>Suspend</td>
<td>![Suspend Icon]</td>
<td></td>
<td>Regain control of the running process.</td>
</tr>
<tr>
<td>Terminate</td>
<td>![Terminate Icon]</td>
<td></td>
<td>Kill the process.</td>
</tr>
<tr>
<td>Restart</td>
<td>![Restart Icon]</td>
<td></td>
<td>Rerun the process from the beginning.</td>
</tr>
<tr>
<td>Resume without signal</td>
<td>![Resume without signal Icon]</td>
<td></td>
<td>Resume the execution of a process without delivering any 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>

*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>Toggle Instruction Stepping</td>
<td>![Icon]</td>
<td></td>
<td>Toggle what all the operators work on (machine instructions or source code).</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:

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.

To see a list of the currently active hotkeys, press Ctrl-Shift-L or choose Help→Key Assist. To customize the debug hotkeys:

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

2 Choose General→Keys in the list on the left.

3 Select a command from the list and click Edit.

4 To assign this command to a new hotkey, click in the Name field in the Key Sequence area of the Keys pane, and then press the key(s) for your new hotkey.

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

6 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. If the program never hits that line, the program runs until it finishes or hits another breakpoint.

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.

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.
Controlling your debug session

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>![Icon]</td>
<td>Kill the selected process.</td>
</tr>
<tr>
<td>Terminate &amp; Remove</td>
<td>![Icon]</td>
<td>Kill the selected process and remove it from the Debug view.</td>
</tr>
<tr>
<td>Terminate All</td>
<td>![Icon]</td>
<td>Kill all active processes in the Debug view.</td>
</tr>
<tr>
<td>Disconnect</td>
<td>![Icon]</td>
<td>Detach the debugger (i.e. <code>gdb</code>) from the selected process (useful for debugging attached processes).</td>
</tr>
<tr>
<td>Remove All Terminated Launches</td>
<td>![Icon]</td>
<td>Clear all the killed processes from the Debug view.</td>
</tr>
<tr>
<td>Restart</td>
<td>![Icon]</td>
<td>Restart the process.</td>
</tr>
</tbody>
</table>
Controlling your debug session

The debugger keeps the project’s files open while the program is running. Be sure to terminate the debug session before you try to rebuild the project, or else the build will fail.

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:

If you click in this view or use the **Toggle Instruction Stepping** icon ( ) to give focus to this view, the operators (e.g. Run to Line) operate on machine instructions instead of the source code.
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>
<tr>
<td>Inspect a process’s memory</td>
<td>Memory</td>
</tr>
<tr>
<td>Inspect shared library usage</td>
<td>Modules</td>
</tr>
<tr>
<td>Monitor signal handling</td>
<td>Signals</td>
</tr>
<tr>
<td>View your output</td>
<td>Console</td>
</tr>
<tr>
<td>Interact 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:
More debugging features

At the bottom of the view, the Detail pane displays the value of the selected variable.

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 yellow (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.

You can decide whether or not to display the variable type (e.g. `int`) by clicking the **Show Type Names** toggle button ( ). The **Show Type Names** button is unavailable when columns are displayed.

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

Tracking all the variables can reduce your program’s performance.

**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 ( ).

2. Select one or more symbols in the Global Variables dialog.
More debugging features

3 Click OK to add the selected global variables to the Variables view.

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:
More debugging features

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

   ![Set Value dialog box](image)

   Enter a new value for argc:

   1

   OK  Cancel

2. Enter the new value in the field.

You can also change a variable’s value in the Detail pane at the bottom of the Variables view. Click the value, change it, and 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. This can also improve your program’s performance.

To enable or disable a variable:

   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 box]

**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:

![Breakpoints view]

136  Chapter 5  •  Debugging Programs
More debugging features

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. For example:

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 is suspended before that line of code is executed.
More debugging features

To add a breakpoint:

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

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:

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.
More debugging features

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 or:
- 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 **Properties**... item. (For breakpoints only, in the C/C++ editor, right-click the breakpoint and select **Breakpoint Properties**...)

2 Use the **Common** panel to modify the watchpoint's behavior.
More debugging features

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.

3. To restrict the breakpoint to specific threads, make sure they’re selected in the **Filtering** panel:
More debugging features

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 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, and then select Remove or Remove All.
More debugging features

Evaluating your expressions

The Expressions view lets you evaluate and examine the value of expressions. To display this view, choose **Windows → Show View → Expressions**.

![Expressions view](image)

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, and then choose **Add Watch Expression**.
More debugging features

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

3 Click OK. The expression and its value appear in the Expressions view. When the debugger suspends the program’s 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. To display this view, choose Windows→Show View→Registers.

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

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. To display this view, choose Windows→Show View→Memory. The view is initially empty, but after you’ve added an item to monitor and specified the output format, this view will look something like this:
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’s Monitors pane, click the Add Memory Monitor button ( ).
3. In the *Enter address or expression to monitor* field, type the address or expression, and then select OK.

**Configuring output format**

You can display memory in hexadecimal or ASCII, or as signed or unsigned integers:

1. In the Memory view’s Renderings pane, click the Add Rendering(s) button ( ). The Add Memory Rendering dialog appears:
More debugging features

2 From the dropdown menu, select the memory or expression you wish to add a new memory rendering for, or click Add New… to create a new memory or expression monitor.

3 Click, Shift-click, or Ctrl-click to choose one or more formats from the Memory Rendering(s) list.

4 Click OK. Each format you’ve chosen appears in a separate tab in the Memory view’s Renderings pane.

Changing memory

To change a process’s memory:

1 Follow the procedures for inspecting a process’s memory and configuring the output format.

2 In the Memory view’s Renderings pane, click in a cell, type the new value for the memory, and then press one of the following:
   - Enter to submit the change
More debugging features

- Esc to leave the memory unchanged

The changed memory appears in red.

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

Inspecting shared-library usage

The Modules view shows you information about the shared libraries for the session you select in the Debug view. The view shows the name, base address, and size of each library. To display this view, choose **Windows→Show View→Modules**.

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. To display this view, choose **Windows→Show View→Signals**.
More debugging features

The view contains the following fields:

**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.

**Description**  
A brief description of 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, and then choose **Signal Properties...** from the menu.
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.

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. If the program isn’t suspended, click the **Suspend** button in the Debug view.

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

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.
More debugging features

Viewing your output

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

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

- standard output (black)
- standard error (red)
- standard input (green)

If you’re connecting to your target via qconn, your process’s output all appears in the same color because qconn combines standard output and standard error into one stream.

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**.

You can have more than one Console view, which is useful if you’re working on more than one application at once:

- To create a new console, use the Open Console icon ( ).
More debugging features

- To display a different console, use the Display Selected Console icon ( ).
- To reduce switching between consoles, use the Pin Console icon ( ).

Remember that you can copy data from the console and paste it elsewhere.

Interacting with GDB

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

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. It appears as soon as the data is sent to it.

To switch to the QNX GDB Console view:

1. In the Debug view, select a debug session.
2. Click the arrow beside the Display selected console button ( ).
3. Choose the console whose name includes gdb. For example:
More debugging features

The Console view changes to the QNX GDB Console view.

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):

To enter commands, you must be on the last line of the Console view.
Chapter 6

Building OS and Flash Images

In this chapter...

Introducing the QNX System Builder 155
Overview of images 162
Creating a project for an OS image 173
Creating a project for a flash filesystem image 175
Building an OS image 175
Downloading an image to your target 180
Configuring your QNX System Builder projects 186
Optimizing your system 202
Moving files between the host and target 205
Introducing the QNX System Builder

Use the QNX System Builder to create OS and flash images for your target.

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, libraries) you want to incorporate into your system image. As you add a
Introducing the QNX System Builder

component, 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](image)
- ![Add a new shared library](image)
Introducing the QNX System Builder

Add a new DLL.

Add a new symbolic link.

Add a new file.

Add a new inline file (i.e. a file whose contents are specified in the buildfile).

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:
Introducing the QNX System Builder

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.

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.
Introducing the QNX System Builder

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 such as 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:
Overview of images

- 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 a multicore 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, and then loads and transfers control to the microkernel/process manager (procnto), the core runtime component of the QNX Neutrino OS.

3. The procnto module then runs the boot script, which performs any final setup required and runs your programs.

At reset, a typical processor has only a minimal configuration that lets code be executed from a known linearly addressable device (e.g.
Overview of images

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.

---

If you’re creating your own startup variant, its name must start with startup or the QNX System Builder perspective won’t recognize it.
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 multicore machines, etc.).

If you’re creating your own `procnto` variant, its name must start with `procnto`- or the QNX System Builder perspective won’t recognize it.

Boot script

If you want your system to load any drivers or to run your program automatically after powering 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

- create adaptive partitions, run programs in them, and set their parameters:

```plaintext
# Create an adaptive partition named "Debugging" with a budget of 20%:
sched_aps Debugging 20

# Start qconn in the Debugging partition:
[sched_aps=Debugging]/usr/sbin/qconn
```
Overview of images

# Use the recommended security level for the partitions:
aps modify -s recommended

For more information about these commands, see the Adaptive Partitioning User's Guide.

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:

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.
### Overview of images

#### 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>
<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-prpcm800.prepboot</td>
</tr>
<tr>
<td>.srec</td>
<td>Suffix for S-record format file</td>
<td>ifs-malta.srec</td>
</tr>
</tbody>
</table>
Overview of images

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 an .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:

- 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.
Overview of images

- 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 Filesystems 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:

Chapter 6 • Building OS and Flash Images  169
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 an existing BSP, or compile your own IPL from your own assembler and C source.

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.
Overview of images

**CAUTION:** 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.

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</strong> directory</td>
<td>The images and generated files that the IDE creates when you build your project.</td>
</tr>
</tbody>
</table>

*continued…*
## Overview of images

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Overrides</strong> directory</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 <code>Overrides/image_name</code> directory to easily test a change to your build. The <code>image_name</code> subdirectory is created automatically, and you must populate it with the override files your image needs.</td>
</tr>
<tr>
<td><strong>Reductions</strong> directory</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 <code>Reductions/image_name</code> directory (where <code>image_name</code> is the name of the image being built).</td>
</tr>
<tr>
<td><code>.project</code> file</td>
<td>Information about the project, such as its name and type. All IDE projects have a <code>.project</code> file.</td>
</tr>
<tr>
<td><code>.sysbldr_meta</code> file</td>
<td>Information about the properties specific to a QNX System Builder project. This file describes where the IDE looks for files (including the <code>Overrides</code> and <code>Reductions</code> directories), the location of your IPL file, how the IDE includes <code>.efs</code> files, and the final format of your <code>.ifs</code> file.</td>
</tr>
</tbody>
</table>

*continued...*
Creating a project for an OS image

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>project.bld file</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 file</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.
Creating a project for an OS image

4 At this point, you can either import an existing buildfile (as shipped with your QNX BSPs) or select a generic type (e.g. "ppcbe").

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 `$QNX_TARGET/processor/boot/build/` on your host.

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. “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

To build your QNX System Builder projects using the standard Eclipse build mechanism:

From the main menu, select **Project**→**Build Project**.

You can also build projects using the context menu:

1. In the Navigator or System Builder Projects view, right-click the project.
2. Select **Build Project**.

The System Builder Console view shows the output produced when you build your images:
Building an OS image

Output can come from any of these utilities:

- `mkefs`
- `mkifs`
- `mkimage`
- `mkrec`
- `objcopy`

For more information, see their entries in the *Utilities Reference*.

You can clear the view by clicking the **Clear Output** button ( ).

**Create new image**

You can create a new image for your QNX System Builder project by using the **Add New Image** icon ( ) in the System Builder editor’s toolbar:

1. Click the **Add New Image** icon in the toolbar.

The IDE displays the Create New Image dialog box:
Building an OS image

2 Use the Create New Image dialog to:

- **Duplicate Selected Image** — create a duplicate of the currently selected image with the given name.
- **Import Existing IFS Buildfile** — generate the new IFS image using an existing buildfile.
- **Import Existing EFS Buildfile** — generate the new EFS image using an existing buildfile.
- **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 image(s)** icon (●) to:

- add an IPL to the start of your image
Building an OS 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 Images view, select your image.
2. Click the Combine image(s) icon (”).
3. In the Create New Image dialog box, check Add IPL.
Building an OS image

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.

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 Create New Image dialog, check Append Image(s).

2 In the Align to 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 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 ROM size field, enter the size of the ROM.
Downloading an image to your target

4 In the **Format** field, select the format from the dropdown menu (e.g. SREC, Intel hex records, binary.)

5 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:
Downloading an image to your target

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.

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:

   ![Terminal settings menu]

   You can now interact with your target by typing in the view.
Downloading an image to your target

- Under Solaris, the Terminal view’s **Device** menu may list all available devices twice 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; you can safely ignore the message.
  ```

- By default on Linux hosts, the owner (**root**) and the group (**uucp**) have read-write permission on all `/dev/ttyS*` serial devices; users outside this group have no access.

  If you’re logged in as a non-**root** user, and you aren’t a member of the **uucp** group, then the Serial Terminal view doesn’t display any serial devices to select from, since you don’t have access rights to any of them. To work around this problem, add non-**root** users to the **uucp** group.

When a connection is made, the **Send File** button changes to its enabled state ( ), 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 ( ).

3. In the Select File to Send dialog, enter the name of your file (or click **Browse**).

4. Select a protocol (e.g. `sendnto`).
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 ( ).
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 → TFTP Server → User Search Paths**.
3. Click **New**, and then select your directory from the Add New Search Path dialog.
4. Click **OK**.
5. 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 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.
Configuring your QNX System Builder projects

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:
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:
Configuring your QNX System Builder projects

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 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
   - Inline File
   - Directory

2. Select an item (e.g. Binary) from the list.
Configuring your QNX System Builder projects

3 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.)

4 Click **OK**. The QNX System Builder adds the item to your image, as you can see in the **Images** pane.

Deleting files

To delete 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 **OK**. 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

To delete directories:

In either the Images or Filesystem pane, right-click your directory and select **Delete**.
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:
### Configuring your QNX System Builder projects

#### Chapter 6: Building OS and Flash Images

![Properties window]

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
<td></td>
</tr>
<tr>
<td>Create Image?</td>
<td>Yes</td>
</tr>
<tr>
<td>Remove File Time Stamps?</td>
<td>No</td>
</tr>
<tr>
<td>Image Name</td>
<td>bios</td>
</tr>
<tr>
<td>CPU Type</td>
<td>x86</td>
</tr>
<tr>
<td>Page Align Image?</td>
<td>No</td>
</tr>
<tr>
<td>Image Mount Point</td>
<td></td>
</tr>
<tr>
<td>Default Target Location</td>
<td>/proc/boot</td>
</tr>
<tr>
<td>Compressed</td>
<td>UCL</td>
</tr>
<tr>
<td>Boot Script</td>
<td>bios.bsh</td>
</tr>
<tr>
<td><strong>Directories</strong></td>
<td></td>
</tr>
<tr>
<td>Default Permissions</td>
<td>777</td>
</tr>
<tr>
<td>Default User ID</td>
<td>0</td>
</tr>
<tr>
<td>Default Group ID</td>
<td>0</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td></td>
</tr>
<tr>
<td>Auto Link Shared Libs?</td>
<td>Yes</td>
</tr>
<tr>
<td>Create startup sym file?</td>
<td>No</td>
</tr>
<tr>
<td>Create proc sym file?</td>
<td>No</td>
</tr>
<tr>
<td>Procinto</td>
<td>procinto</td>
</tr>
<tr>
<td>Procinto Arguments</td>
<td>-v</td>
</tr>
<tr>
<td>Procinto $PATH</td>
<td>/proc/boot</td>
</tr>
<tr>
<td>Procinto $LD_LIBRARY_PATH</td>
<td>/proc/boot</td>
</tr>
<tr>
<td>Use APS?</td>
<td>No</td>
</tr>
<tr>
<td>Startup</td>
<td>startup-bios</td>
</tr>
<tr>
<td>Startup Arguments</td>
<td></td>
</tr>
<tr>
<td>Boot File</td>
<td>bios</td>
</tr>
<tr>
<td>Image Address</td>
<td>Default</td>
</tr>
<tr>
<td>RAM Address</td>
<td>Default</td>
</tr>
<tr>
<td><strong>Combine</strong></td>
<td></td>
</tr>
<tr>
<td>IPL file</td>
<td></td>
</tr>
<tr>
<td>Pad IPL to</td>
<td></td>
</tr>
<tr>
<td>Align File System to</td>
<td></td>
</tr>
<tr>
<td>Offset</td>
<td>0x0</td>
</tr>
<tr>
<td>ROM size</td>
<td></td>
</tr>
<tr>
<td>Combined Image format</td>
<td></td>
</tr>
<tr>
<td>Images to combine with</td>
<td></td>
</tr>
<tr>
<td>Mount unbootable IF5s?</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Configuring your QNX System Builder projects

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.
Configuring your QNX System Builder projects

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).

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.

Default Target Location

The default path where the filesystem is located when the boot process completes. By default, the location is `/proc/boot`.

Chapter 6 • Building OS and Flash Images 193
Configuring your QNX System Builder projects

**Image Mount Point**

The path where the filesystem is mounted in the filesystem. By default, the location is 
/proc/boot.

**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 execute-in-place (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.).
Configuring your QNX System Builder projects

**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.).

**Startup Arguments**
Command-line arguments for the startup program.

**Use APS?**
If Yes, the System Builder adds the **aps** module to the invocation of *procnto* in the OS image. You need this module if you want to use adaptive partitioning on the target system.

To create partitions and run programs in them at boot time, add the appropriate commands to the image’s .bsh file. For more information, see “Boot script” earlier in this chapter.
Configuring your QNX System Builder projects

**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.

**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 4 GB 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.
Configuring your QNX System Builder projects

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 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.
Configuring your QNX System Builder projects

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
- Builders
- Project Preferences
Configuring your QNX System Builder projects

- Refactoring History
- 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:

![Image of Search Paths pane]

The IDE provides separately configurable search paths for:

- binaries
Configuring your QNX System Builder projects

- 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 Path — a path with a $WORKSPACE/projectName prefix

   Another 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
   - DLLs
   - Other Files
   - System Files

2 Select a path, then click one of these buttons:
   - Move Up
   - Move Down
   - Remove Selected

**CAUTION:** The *Overrides/image_name* and * Overrides* directories must be first ones in the list. The *Reductions/image_name* and *Reductions* directories, which are listed in the **Shared Libraries** tab, must be next 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:

- CPU
- CPUDIR
- PLATFORM
- PROJECT
- QNX_TARGET
Optimizing your system

- QNX_TARGET_CPU
- VARIANT
- WORKSPACE

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:

   **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 nonfunctional. The order of operations is key!
Optimizing your system

To ensure that your image works and is as efficient as possible, you should select all three options.

4 Click Next. On the next three pages, 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 reduced libraries appear in your project’s Reductions/image_name directory.

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.

4 In the System Optimizer, select the Apply diet(s) system wide option.
Moving files between the host and target

5 Click **Next**. In the next few pages, the Dietician shows the unnecessary libraries, any additional needed libraries, and the libraries that can be optimized.

6 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 **Reducations/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 **Reducations** directory, and then rebuild your image using a standard, “full-weight” shared library.

To restore a library to its original state:

1 In the Navigator or System Builder Projects view, open the **Reducations** directory in your project. This directory contains the reduced versions of your libraries.

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 original 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.
Moving files between the host and target

If you haven’t yet created a target system, you can do so right from within the Target File System Navigator view.

To create a target system:

Right-click anywhere in the view, then select **Add New Target**.

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:

1. From the main menu, select **Window → Show View → Other...**
2. Select **QNX Targets**, then double-click Target File System Navigator.

The view displays the target and directory tree in the left pane, and the contents of the selected directory in the right pane:
Moving files between the host and target

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**.

**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.

Drag-and-drop 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.

Drag-and-drop is not supported on Neutrino hosts.
Chapter 7
Developing Photon Applications

In this chapter…

- What is PhAB? 211
- Using PhAB 213
- Starting Photon applications 217
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?

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, expand **QNX**.
3. Select **Photon Appbuilder Project**, and then click **Next**.
4. Name your project. If you don’t want to use the default location for the project, specify a different one.
5. Click **Next**.
6. Select your target architecture.

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.

7. Click **Finish**.

The IDE creates your project, then launches PhAB. (In Windows, the IDE also creates a **Console for PhAB** window.)
Using PhAB

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, select it in the **C/C++ Projects view**, open the **Project** menu, and then 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.
Using PhAB

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. Select the **Commands** tab.
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.
Using PhAB

**Importing an existing standalone QNX Photon Appbuilder project**

If you have an application written in the standalone version of QNX Photon Appbuilder, you can import it into an empty IDE-based PhAB project. Once the project is imported, you can edit the source and build the project in the IDE. To import a standalone PhAB project:

1. Open the project in standalone PhAB and make sure it uses an Eclipse Project directory structure. To do this, select **Project → Convert to Eclipse Project**.

   If this option is grayed out, the project already uses an Eclipse Project directory structure.

2. Save the project.

3. Run the IDE and create a new Photon Appbuilder Project. This will run PhAB automatically, but for now cancel and exit out of PhAB. This creates a new, empty PhAB project in your workspace.

4. Select **File → Import...**, and choose **File system** to import resources from the local filesystem. Browse to the project saved earlier by standalone PhAB, and import the following four items:
   - the *abapp.dfn* file
   - the *abapp.wsp* file
   - the *src* directory and contents
   - the *wgt* directory and contents

5. At this point you can rebuild the project using the IDE, and launch PhAB from the IDE to edit the 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 221
Profiling your programs 223
Controlling your profiling sessions 233
Understanding your profiling data 235
Introducing the Application Profiler

This chapter shows you how to use the application profiler.

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 unintrusively, 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; for qcc, see the Utilities Reference.

**Profiling your programs**

Whether you plan to do profiling in real time or postmortem (using a gmon.out file), you should build your programs with profiling enabled before starting a profiling session.

This section includes these topics:

- Building a program for profiling
- Running and profiling a process
- Profiling a running process
Profiling your programs

- 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.

Profiling is handled by functions in `libc`; be sure to check our website for `libc` updates from time to time.

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 level)</td>
<td>Yes</td>
<td>Yes (function level)</td>
<td>Yes</td>
</tr>
<tr>
<td>sampling</td>
<td></td>
<td></td>
<td></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>
<tr>
<td>profiling</td>
<td></td>
<td></td>
<td></td>
<td></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.
Profiling your programs

8 Rebuild your project.

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 Application Profiler tool.
5 Click OK.
6 On the launcher, click the Application Profiler tab:
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.

**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.

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**.
Profiling your programs

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:

When you profile a running process, you can’t use the Console view in the IDE to interact with this process. If your running process requires user input through the Console view, use a shell to interact with the process.

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 **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
Profiling your programs

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**, and then click **Debug**. The Select Process dialog is displayed, showing all of the currently running processes:

![Select Process Dialog]

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:

```
QCONN_PROFILER=/dev/profiler ./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. <strong>prof201 [C/C++ QNX QConn (IP)]</strong>)</td>
<td>![image]</td>
</tr>
<tr>
<td>Profiled program</td>
<td>Project name and start time (e.g. <strong>prof201 on localhost pid 4468773 (3/4/03 12:41 PM)</strong>)</td>
<td>![image]</td>
</tr>
<tr>
<td>Application Profiler instance</td>
<td>Program name and target computer (e.g. <strong>Application Profiler Attached to: prof201 &lt;4468773&gt; on 10.12.3.200</strong>)</td>
<td>![image]</td>
</tr>
<tr>
<td>Executable</td>
<td></td>
<td>![image]</td>
</tr>
<tr>
<td>Shared libraries</td>
<td></td>
<td>![image]</td>
</tr>
<tr>
<td>DLLs</td>
<td></td>
<td>![image]</td>
</tr>
<tr>
<td>Unknown</td>
<td></td>
<td>![image]</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:

1. In the Application Profiler view, select a launch configuration.
Understanding your profiling data

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:

1 Launch a profile session for a debuggable (i.e. _g) executable.
Understanding your profiling data

2 In the Application Profiler view, select your program by selecting an Application Profiler instance ( ) 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:

```
uint64_t
func1 (uint64_t n) /* 33.3% of to
{
    volatile uint64_t a, b, c;

    if (optv)
        printf("%s: Now in func1()\n", program);

    for (a = 0; a < 2*n; a++) /* 50% of func
    for (b = 0; b < n; b++) {} /* 25% of func
    for (c = 0; c < n; c++) {} /* 25% of func
    return 1;
}

uint64_t
func2 (uint64_t n) /* 66.7% of to
{
    volatile uint64_t d, e;
```

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:
Understanding your profiling data

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:

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:
Understanding your profiling data

The view lists all the functions called in the selected item. For each function, this view displays:

- 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 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:

Select the **Use/Calls since last reset** item from the dropdown menu in the title bar for the Sampling Information view.

**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:
Understanding your profiling data

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 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:

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:
Understanding your profiling data

**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.

**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.
Understanding your profiling data

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.
In this chapter...

Code coverage in the IDE 245
Enabling code coverage 248
Importing gcc code coverage data from a project 252
Associated views 253
Code Coverage Sessions View 254
Code Coverage Properties View 259
Code Coverage Report View 260
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
Code coverage in 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:
- 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 an 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 you can later import the data into the IDE code coverage tool. For information about importing gcc coverage data from a project, see “Importing gcc code coverage data from a project” later in this chapter.

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.
Enabling code coverage

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 Clean....

7. Be sure that your project is selected, and check the Start a build immediately box, then click OK to rebuild your 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.
Enabling code coverage

For example, your **Makefile** might look something like this:

```makefile
objects:=Profile.o main.o
CC:=qcc -Vgcc_ntox86
CFLAGS:=-g -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) $ˆ -o $@ $(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:
Enabling code coverage

Enable GCC 3 Coverage metrics collection

Check this if your application was compiled with gcc 3.3 or later. The default is to collect code coverage information from applications compiled with gcc 2.95.

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.
### Importing gcc code coverage data from a project

**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.

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. toolsselection.gif

8 Click **Apply**.

9 Click **Run** or **Debug**.

### Importing gcc code coverage data from a project

If you launched a code coverage-**enabled** build process and chose to disable code coverage in the launch configuration, the process writes the coverage information to a data file (`.da`) at run time, rather than read it from the process’s data space. This means that you can choose to import this data into the IDE Code Coverage tool at a later time.

To import gcc code coverage data from a project:

1 Create and build a project with code coverage selected. For information about enabling code coverage, see “Enabling code coverage” earlier in this chapter.

2 Create a launch configuration where code coverage is **disabled**.

3 Run this configuration.

4 Observe the target’s directory using the **Target File System Navigator** tab in the **Tasks** view (bottom of the Workbench window) in the location where the file `project_name.da` resides.
By default, you will not have the Target File System Navigator tab in your Tasks view. To add this tab to your view:

4a Select Window→Show View→Other.
4b Expand QNX Targets.
4c Select Target File System Navigator.
4d Click OK.

5 For the target, right-click on the file project_name.da and select Copy to→Workspace.

6 In the Select Target Folder window, specify a folder location to copy the file, and click OK.

The project_name.da will be visible under the C/C++ tab for the corresponding project.

7 On the QNX Code Coverage tab, select File→Import→GCC Code Coverage Data from Project.

8 Specify the name of the session, click Browse to locate a project, and then click OK.

9 Click Finish.

Now, the Code Coverage tab displays the session name and imported gcc code coverage data for the selected project.

Associated views

The QNX Code Coverage perspective includes the following views:

- Code Coverage Sessions View (controlling your session and examining data line-by-line)
- Code Coverage Properties View (seeing your coverage at a glance)
- Code Coverage Report View (examining your coverage report)
The Code Coverage Sessions view lets you control and display multiple code-coverage sessions:

The view displays the following as a hierarchical tree for each 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. <code>ccov102_factor [GCC Code Coverage] (7/2/03 2:48 PM)</code>)</td>
<td>![icons]</td>
</tr>
<tr>
<td>Project</td>
<td>Project name and amount of coverage (e.g. <code>ccov102_factor [ 86.67% ]</code>)</td>
<td></td>
</tr>
</tbody>
</table>

continued...
The IDE uses several icons in this view:

<table>
<thead>
<tr>
<th>Icon</th>
<th>Icon Color</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>☐</td>
<td>White</td>
<td>No coverage</td>
</tr>
<tr>
<td>☢</td>
<td>Yellow</td>
<td>Partial coverage</td>
</tr>
<tr>
<td>☣</td>
<td>Green</td>
<td>Full (100%) coverage</td>
</tr>
<tr>
<td>'?'</td>
<td></td>
<td>Missing or out-of-date source file</td>
</tr>
</tbody>
</table>

The IDE also adds a coverage markup icon ( sorter ) 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 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
Code Coverage Sessions View

(✓) 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:

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:

Generating a report

To generate a report, 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...
browser. Using an external browser lets you compare several reports simultaneously.

Changing views

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 General→Web Browser.
3  In the right pane, enable/disable the Use external Web browser check box.
4  Click OK.

Printing a report

To print a report:

In the Code Coverage Report view’s title bar, click the Print button ( ).

Saving a report

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./neutrino

Refreshing a report

To refresh a report:

In the Code Coverage Report view’s title bar, click the Refresh button ( ).

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:
Opening a file in the Code Coverage perspective

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.

Displaying coverage information for a specific session

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 (/gif) overlays the source file icon.

Displaying coverage information when opening a file

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.

Removing coverage markers

To remove all coverage markers:

In the Code Coverage Sessions view’s title bar, click the Remove All Coverage Markers button (/gif).

Code Coverage Properties View

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:
Code Coverage Report View

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:
Generating a 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 browser. Using an external browser lets you compare several reports simultaneously.

Changing views

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 **General**→**Web Browser**.

3 In the right pane, enable/disable the **Use external Web browser** check box.

4 Click **OK**.

**Printing a report**

To print a report:

In the Code Coverage Report view’s title bar, click the **Print** button ( 

**Saving a report**

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.

**Refreshing a report**

To refresh a report:

In the Code Coverage Report view’s title bar, click the **Refresh** button ( 

Chapter 10
Finding Memory Errors

In this chapter...

Introduction 265
Analyzing your program 276
Interpreting errors during memory analysis 282
Error message summary (memory analysis) 300
Memory analysis GUI flags and corresponding environment variables 302
Using a file to log the trace 304
Analyzing a running program 305
Analyzing shared objects 307
Associated views 308
Memory analysis session 308
Memory analysis editor 313
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.
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 4 KB each). To determine the size for your system, use the `sysconf(_SC_PAGESIZE)` function.

As you’ll see when you use the Memory Information view of the QNX System Information perspective, the IDE categorizes your program’s virtual address space as follows:

- program
- stack
- shared library
- objects
- heap
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:
Introduction

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.
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 isn’t 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 4 KB blocks, but allocations are typically much smaller. Since it would be wasteful to use 4 KB 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 4 KB increments:

![Diagram showing memory mapping]

**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 unintrusively 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_g.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 NUL 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.

Be sure to occasionally check the Download Center on our website for updated versions of the debug `malloc` library.

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.

The following tabs in the Memory Analysis editor can help you find and fix memory leaks:

- **Allocations tab** — shows you all the instances where your program allocates, reallocates, and frees memory. The view lets you hide allocations that have a matching deallocation; the remaining allocations are either still in use or forgotten.

- **Errors tab** — shows you all memory errors, including leaks (unreachable blocks).
For more information about the memory analysis editor, see Viewing memory analysis data (memory analysis editor) later in this chapter.
For detailed descriptions about memory errors, see “Interpreting errors during memory analysis”.

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. For more information about how to interpret memory errors during memory analysis, see the topic “Interpreting errors during memory analysis” later in this chapter.

In the event of a memory error, the IDE can:

- stop your program’s execution and let you see all the allocations that led up to the error
- report the error and try to continue
- launch the debugger

The resulting action that the IDE takes depends on the setting that you chose in the Memory Analysis Tooling part of the launch configuration (see “Analyzing your program,” later in this chapter).

The Memory Analysis editor’s Errors and Statistics tabs display memory errors and — if possible — the exact line of source code that generated each error. The Trace tab lets you find the prior call that accessed the same memory address, even if your program made the call days earlier. For more information about the memory analysis editor, see Viewing memory analysis data (memory analysis editor) later in this chapter. For detailed descriptions about memory errors, see “Interpreting errors during memory analysis”.

Chapter 10 • Finding Memory Errors 275
Analyzing your program

To learn more about the common causes of memory problems, see the topic Heap Analysis: Making Memory Errors a Thing of the Past chapter of 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 Memory Analysis Tooling:
5 Click **OK**.

6 Click the **Memory Analysis Tooling** tab.
Analyzing your program

To configure the Memory Analysis settings for your program, click the disclosure triangle for the appropriate set of options:

- **Memory Errors**
  This group of configuration options controls the Memory Analysis tool’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**
Analyzing your program

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 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 attempts 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.

**Perform leak check every (ms)**
Specify how often you want to check for leaks. Note that this checking comes with a performance penalty.

**Perform leak check when process exits**
When checked, look for memory leaks when the process exits, before the operating system cleans up the process’s resources.

- **Memory Tracing**
  This group of configuration options controls the Memory Analysis tool’s memory tracing features.

**Enable memory allocation/deallocation tracing**
When checked, trace all memory allocations and deallocations.
Analyzing your program

Limit back-trace depth to
Specify the number of stack frames to record when tracing memory events.

Minimum allocation to trace
The size, in bytes, of the smallest allocation to trace.
Use 0 to trace all allocations.

Maximum allocation to trace
The size, in bytes, of the largest allocation to trace. Use 0 to trace all allocations.

Perform tracing every (ms)
How often to collect information about your program’s allocation and deallocation activity. When setting this, consider how often your program allocates and deallocates memory, and for how long you plan to run the program.

Memory Snapshots
Controls the Memory Analysis tool’s memory snapshot feature.

MemorySnapshots
Check this to enable memory snapshots.

Perform snapshot every (ms)
Specify the number of milliseconds between each memory snapshot.

Bins counters (comma separated) ex: 2,4,6,8,...
A comma-separated list of the memory bins you want to trace.

Library search paths
A list of the libraries that you want to have backtrace information for. For more information, see “Analyzing shared objects,” later in this chapter.

Target Settings
These settings let you specify details about how memory debugging will be handled on the target system.
Analyzing your program

**Malloc library:**

The full path *on the target* to the memory-debugging library, usually

$\texttt{QNX\_TARGET/target\_architecture/usr/lib/libmalloc\_g.so}$.

- **Send traces to:**
  
The full path to the device that will receive trace messages. The default is `/dev/dbgmem`. You can also log traces to a file on the target; for more information, see “Using a file to log the trace,” later in this chapter.

- **Send events to:**
  
The full path to the device that will receive memory events. The default is `/dev/dbgmem`.

- **Create control thread**
  
Enable this to use a separate thread for memory tracing operations.

For example, if you attach the Memory Analysis tool to a program that uses `fork`, it creates a second thread; however, `fork` only works with single threaded programs.

Consequently, the control thread of the Memory Analysis tool must be disabled.

- **Use dladdr to find dll names**
  
Check this if you’d like to get backtrace information from shared objects that were built with debugging information.

For more information, see “Analyzing shared objects,” later in this chapter.

- **Show debug output on console**
  
Enable this to show messages from the memory-debugging library in the Console view.

- **Data Collection**

This setting lets you choose the type of database used to register the memory traces and events:

- **Neutrino Derby Data Collection**
  
A file database that’s slower, but uses less memory.
Interpreting errors during memory analysis

- **NTO HSQLDB Data Collection**
  An in-memory database. It’s fast, but before grabbing large traces, you should make sure the Java VM has enough memory. You can use the `-Xmx` option in `qde.ini` to specify the amount of memory for the Java VM. You must restart the IDE for this option to take effect.

  For example, to collect 2 million events and memory traces, you’ll need about 2 GB of memory; specify `-Xmx2048m`. The default size is 512 MB.

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, Debug, or Profile**. The IDE starts your program and lets you analyze your program’s memory.

Don’t run more than one Memory Analysis session on a given target at a time, because the results may not be accurate.

---

**Interpreting errors during memory analysis**

Although the QNX Memory Analysis perspective shows you how your program uses memory, and can quickly direct you to memory errors in your development and testing environments, you need to understand the types of memory errors that you might run into.

During memory analysis, you may encounter the following types of memory errors:

- runtime errors
  - illegal deallocation of memory
  - null pointer dereference
  - buffer overflow
Interpreting errors during memory analysis

- illegal function arguments
  - using freed memory
  - reading uninitialized memory
- resource (memory) leaks

When memory errors occur, the IDE can:

- report the error by logging the memory error, and then continue with the process
- launch the debugger and stop at the code that resulted in the error
- terminate the process

Regardless of which course of action the IDE takes, all of the error and non-error events are available to you in the Trace Event Log.

To include the Trace Event Log in your Task view:

- Click Windows→Show Views→Other.
- Expand QNX System Profiler.
- Select Trace Event Log.
- Click OK.

To enable memory analysis:

1. From an existing launch configuration, select the Tools tab.
2. Select Add/Delete Tool.
3. Select Memory Analysis Tooling and click OK.
Interpreting errors during memory analysis

4 Select any required memory tools and specify any desired options for that tool.

After you configure the IDE for memory analysis, you can begin to use the results to identify the types of memory errors in your programs, and then trace them back to your code.

To view the memory errors identified by the IDE, and then navigate to those errors:

1 After enabling Memory Analysis in a launch configuration, run that configuration.

2 In the Session view, double-click your desired launch configuration.

A dialog with the same name will contain a list of memory errors that the IDE encountered in your program.
Interpreting errors during memory analysis

In addition, an editor with multiple tabs will open at the bottom of the Workbench window.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Description</th>
<th>Pointer</th>
<th>TrapFunc...</th>
<th>Threethread</th>
<th>PID</th>
<th>TID</th>
<th>Operation</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f080</td>
<td>strcpy</td>
<td>5093724</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f090</td>
<td>strcpy</td>
<td>5129222</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f0a0</td>
<td>strcpy</td>
<td>521921</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f0b0</td>
<td>strcpy</td>
<td>5242920</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f0c0</td>
<td>strcpy</td>
<td>122039</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f0d0</td>
<td>strcpy</td>
<td>54913</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f0e0</td>
<td>strcpy</td>
<td>540316</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f0f0</td>
<td>strcpy</td>
<td>556915</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f000</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f001</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f002</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f003</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f004</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f005</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f006</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f007</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f008</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f009</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f00a</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f00b</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f00c</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f00d</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f00e</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f00f</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f010</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f011</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f012</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f013</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f014</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f015</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f016</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f017</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f018</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f019</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f01a</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f01b</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f01c</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f01d</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f01e</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
<tr>
<td>ERROR</td>
<td>pointer points to heap but not to a user writable area</td>
<td>0x0004f01f</td>
<td>strcpy</td>
<td>577574</td>
<td>577574</td>
<td>l</td>
<td>malloc</td>
<td>in use</td>
</tr>
</tbody>
</table>

| 3 | Click the Errors tab. |
Interpreting errors during memory analysis

4 In the window that lists the types of errors, select an error. Notice that the information in the Errors tab dynamically updates to reflect the error that you’ve selected.

5 On the Errors tab, double-click on an error to navigate to that error in the code editor.

6 Modify the code, as required, to correct the memory error for the selected error.

Illegal deallocation of memory

The illegal deallocation of memory occurs when a `free()` operation is performed on a pointer that doesn’t point to an appropriate heap memory segment. This type of error can occur when you attempt to do any of the following activities:

- free a NULL pointer (not detected)
- free a pointer to stack or static memory
- free a pointer to heap memory that does not point to the beginning of an allocated block
- perform a double free (when `free()` is performed more than once on the same memory location)

Consequences

The illegal deallocation of memory can generate the following runtime errors:

- memory corruption (a stack, heap, or static segment)
- immediate segmentation fault

Detecting the error
Interpreting errors during memory analysis

For instructions about enabling error detection in the IDE, see “Enabling error detection for the illegal deallocation of memory”.

In the QNX IDE, the memory analysis feature detects this error (if error detection is enabled), and it traps the illegal deallocation error when any of the following functions are called:

- `free()`
- `realloc()`

**Enabling error detection for the illegal deallocation of memory**

To enable error detection for the illegal deallocation of memory:

1. In the Launch Configuration window, select the Tools tab.
2. Select the Enable error detection checkbox.
3. Select the Enable check on realloc()/free() argument checkbox.
4. Click OK.

**Message returned to the QNX IDE**

In the IDE, you can expect the message for this type of memory error to include the following types of information and detail:

- Message: **Pointer does not point to heap area**
- Severity: ERROR
- Pointer: 0 (typically 0 for most messages)
- TrapFunction: shows the `free()` or `realloc()` function where the error occurred.
- Operation: displayed, where applicable

For a list of error messages returned by the Memory Analysis tool, see “Error message summary (memory analysis)”.
Interpreting errors during memory analysis

How to address the illegal deallocation of memory

To help address this memory problem, try the following:

- Add a condition to test that when a NULL is a pointer, to verify that it can be freed.
- Don’t free stack and static memory. Ensure that the same pointer can never point to different types of memory.
- Never reassign an allocated pointer (except for a NULL or other allocation). If you need to iterate over allocated memory, use another pointer (alias), or just use an index.
- Nullify the pointer immediately after deallocation, unless it is a local variable which is out of scope.

Example

The following code shows an example of the illegal deallocation of memory:

```c
int main(int argc, char ** argv){
    char * str = "";
    if (argc>1) {
        str = malloc(10);
        // ...
    }
    printf("Str: %s\n",str);
    free(str);
    return 0;
}
```

NULL pointer dereference

A NULL pointer dereference is a sub type of an error causing a segmentation fault. It occurs when a program attempts to read or write to memory with a NULL pointer.
Interpreting errors during memory analysis

Consequences

Running a program that contains a NULL pointer dereference generates an immediate segmentation fault error.

For instructions about enabling error detection in the IDE, see “Enabling error detection for a NULL pointer dereference”.

When the memory analysis feature detects this type of error, it traps these errors for any of the following functions (if error detection is enabled) when they are called within your program:

- `free()`
- memory and string functions:
  
  - `strcat()`
  - `strndup()`
  - `strnccat()`
  - `strcmp()`
  - `strncmp()`
  - `strcpy()`
  - `strncpy()`
  - `strlen()`
  - `strchr()`
  - `strrchr()`
  - `index()`
  - `rindex()`

The memory analysis feature doesn’t trap errors for the following functions when they are called:

- `memccpy()`
- `bcopy()`
- `memcmpl()`
- `bzero()`
- `memset()`
- `bcmp()`

Enabling error detection for a NULL pointer dereference

To enable error detection for the NULL pointer dereference:

1. In the Launch Configuration window, select the Tools tab.
2. Select the Enable error detection checkbox.
Interpreting errors during memory analysis

3 To detect the passing of a zero (0) pointer to string and memory functions, select **Verify parameters in string and memory functions**.

4 To detect the freeing of a zero (0) pointer, select **Enable check on realloc()/free() argument**.

Message returned to the QNX IDE

In the IDE, you can expect the message for this type of memory error to include the following types of information and detail:

- **Message**: various types of messages expected
- **Severity**: ERROR
- **Pointer**: 0
- **TrapFunction**: shows the memory or string function where the error occurred.
- **Operation**: displayed, where applicable

For a list of error messages returned by the Memory Analysis tool, see “Error message summary (memory analysis)”.

How to address a NULL pointer dereference

You can perform an explicit check for NULL for all pointers returned by functions that can return NULL, and when parameters are passed to the function.

Example

The following code shows an example of a NULL pointer dereference:

```c
int main(int argc, char ** argv){
    char buf[255];
    char * ptr = NULL;
    if (argc>1) {
        ptr = argv[1];
    }
```
Interpreting errors during memory analysis

```c
strcpy(str, ptr);
return 0;
```

Buffer overflow

A buffer overflow error occurs when a program unintentionally writes to a memory area that’s out of bounds of the buffer it intended to write to.

Consequences

A buffer overflow generates the following runtime errors:

- memory corruption (with an unpredictable failure in the future)
- segmentation fault

Detecting the error

The Memory Analysis tool can detect a limited number of possible buffer overflows with following conditions:

- when the overflow buffer belongs to the heap area
- when the overflow occurred within the block’s memory overhead (typically, the overflow is over by 1, and the overflow is trapped in the `free()` function)
- when the overflow is corrupting the heap. Typically, with a large enough index (or negative index), you can write data into next block area, thereby making all of the heap unusable. This error is trapped in the following allocation functions: `malloc()`, `calloc()`, `realloc()`, `free()`
- when the overflow occurred in a library function:

  ```c
  strchr()    strncmp()    strcmp()    strpbrk()
  strcpy()    strcmp()    strchr()    strspn()
  strdup()    strncpy()    strrchr()   strcspn()
  strncat()   strlen()    rindex()    strstr()
  ```

Chapter 10 • Finding Memory Errors 291
Interpreting errors during memory analysis

strtok() memmove() memset() bcmp()
memccpy() memcpy() bcopy()
memchr() memcmp() bzero()

Enabling error detection

To enable error detection for a buffer overflow or underflow:

1. In the Launch Configuration window, select the **Tools** tab.
2. Select **Enable error detection** checkbox.
3. To detect an immediate overflow, select **Verify parameters in string and memory functions**.
4. To detect a small overflow in block’s memory overhead area, select **Enabled bounds checking (where possible)**.
5. To detect a corrupted heap, caused by overflowing other regions, select **Perform full integrity check on every allocation/deallocation**.

Message returned to the QNX IDE

In the IDE, you can expect the message for this type of memory error to include the following types of information and detail:

- **Messages**
  - *allocator inconsistency* - Malloc chain is corrupted, pointers out of order
  - *allocator inconsistency* - Malloc chain is corrupted, end before end pointer
  - *pointer does not point to heap area*
  - *possible overwrite* - Malloc block header corrupted
  - *allocator inconsistency* - Pointers between this segment and adjoining segments are invalid
Interpreting errors during memory analysis

- data has been written outside allocated memory block
- pointer points to heap but not to a user writable area
- allocator inconsistency - Malloc segment in free list is in-use
- malloc region doesn’t have a valid CRC in header

- Other parameters
  - Severity: ERROR
  - Pointer: pointer that points outside of buffer
  - TrapFunction: memory or string function where the error was trapped (the error can also occur before the actual function in error)
  - Operation: UNKNOWN, malloc, malloc-realloc, calloc — how memory was allocated for the memory region we are referencing
  - State: In Use or FREED

For a list of error messages returned by the Memory Analysis tool, see “Error message summary (memory analysis)”.

How to address buffer overflow errors

Locate the code where the actual overflow occurred. Ensure that the size of the memory region is always accompanied by the pointer itself, verify all unsafe operations, and that the memory region is large enough to accommodate the data going into that location.

Example

The following code shows an example of a buffer overflow trapped by a library function:

```c
int main(int argc, char ** argv){
    char * ptr = NULL;
```
Interpreting errors during memory analysis

```c
ptr = malloc(12);
strcpy(ptr,"Hello World!");
return 0;
}
```

The following code shows an example of a buffer overflow trapped by a post-heap check in a `free()` function:

```c
int main(int argc, char ** argv){
    char * ptr = NULL;
    ptr = malloc(12);
    ptr[12]=0;
    free(pre);
    return 0;
}
```

Using freed memory

If you attempt to read or write to memory that was previously freed, the result will be a conflict and the program will generate a memory error. For example, if a program calls the `free()` function for a particular block and then continues to use that block, it will create a reuse problem when a `malloc()` call is made.

Consequences

Using freed memory generates the following runtime errors:

- memory corruption (results in an unpredictable future failure)
- random data read — when the heap is re-used, other data can be in that location

Detecting the error

The Memory Analysis tool can detect only a limited number of situations where free memory is read/written with following conditions:

- where library functions read a pointer that are already known to be free, those functions are:
Interpreting errors during memory analysis

strcat()  strlen()  strcsnp()  memcmpp()
strdup()  strchr()  strstr()  memset()
strncat()  strchr()  strtok()  bcopy()
strncmp()  index()  memccpy()  bzero()
strcpy()  strchr()  memchr()  bcmp()
strncpy()  rindex()  memmove()  bcmp()

- The newly allocated block contains altered data; it was modified after deallocation. The memory errors are trapped in the following memory functions:

malloc()  calloc()  realloc()  free()

Enabling error detection

To enable error detection when using freed memory:

1. In the Launch Configuration window, select the Tools tab.
2. Select the Enable error detection checkbox.
3. To detect usage of freed memory, select Verify parameters in string and memory functions.
4. To detect writing to a freed memory area, select Enabled bounds checking (where possible).

Message returned to the QNX IDE

In the IDE, you can expect the message for this type of memory error to include the following types of information and detail:

- Messages: data in freed memory block has been modified
- Severity: ERROR
- Pointer: not specified
- TrapFunction: shows the memory or string function where the error occurred (where the error was trapped).
Interpreting errors during memory analysis

- Operation: **In Use** or **Free** — indicates whether the memory region is being used or is available.

For a list of error messages returned by the Memory Analysis tool, see “Error message summary (memory analysis)”.

**How to address freed memory usage**

Set the pointer of the freed memory to NULL immediately after the `free()`, unless it is a local variable that goes out of the scope in the next line of the program.

**Example**

The following code shows an example using already freed memory:

```c
int main(int argc, char ** argv){
    char * ptr = NULL;
    ptr = malloc(13);
    free(ptr);
    strcpy(ptr,"Hello World!");
    return 0;
}
```

**Reading uninitialized memory**

If you attempt to read or write to memory that was previously freed, the result will be a conflict and the program will generate a memory error because the memory is not initialized.

**Consequences**

Using an uninitialized memory read generates a random data read runtime error.

**Detecting the error**

Typically, the IDE does not detect this type of error; however, the Memory Analysis tool does trap the condition of reading uninitialized data from a recently allocated memory region.

For a list of error messages returned by the Memory Analysis tool, see “Error message summary (memory analysis)”.

296  Chapter 10 • Finding Memory Errors
Interpreting errors during memory analysis

How to address random data read issues

Use the calloc() function, which always initializes data with zeros (0).

Example

The following code shows an example of an uninitialized memory read:

```c
int main(int argc, char ** argv){
    char * ptr = NULL;
    ptr = malloc(13);
    if (argc>1)
        strcpy(ptr,"Hello World!");
    ptr[12]=0;
    printf("%s\n",ptr);
    return 0;
}
```

Resource (memory) leaks

Memory leaks can occur if your program allocates memory and then does not free it. For example, a resource leak can occur in a memory region that no longer has references from a process.

Consequences

Resource leaks generate the following runtime errors:

- resource Exhaustion
- program termination

Detecting the error

This error would be trapped during the following circumstances:

- a typical program exit (versus an abnormal program exit/termination)
- routine investigation (set by the programmer or tester) at regular intervals
Interpreting errors during memory analysis

Enabling error detection

In the IDE, you can expect the message for this type of memory error to include the following types of information and detail:

1. In the Launch Configuration window, select the **Tools** tab.
2. Select the **Perform leak check when process exits** checkbox.
3. Optional: Specify how often to check for leaks in the **Perform leak check every (ms)** field. The minimum depends on target speed; however, on average, it should be no less than 100 ms.

Message returned to the QNX IDE

In the IDE, you can expect the message for this type of memory error to include the following types of information and detail:

- Message: varies
- Severity: LEAK
- Pointer: lost pointer
- TrapFunction: blank
- Operation: `malloc()`, `realloc()`, `alloc()`, `calloc()` — how memory was allocated for this leak
- State: empty

For a list of error messages returned by the Memory Analysis tool, see “Error message summary (memory analysis)”.

How to address resource (memory) leaks

To address resource leaks in your program, ensure that memory is deallocated on all paths, including error paths.
Interpreting errors during memory analysis

Example

The following code shows an example of a memory leak:

```c
int main(int argc, char ** argv){
    char * str = malloc(10);
    if (argc>1) {
        str = malloc(20);
        // ...
    }
    printf("Str: %s\n",str);
    free(str);
    return 0;
}
```

Functions checked for memory errors during memory analysis

During memory analysis, the following functions are checked for memory errors:

- string functions:
  - `strcat()`, `strncpy()`, `strpbrk()`
  - `strdup()`, `strlen()`, `strspn()`
  - `strncat()`, `strchr()`, `strcspn()`
  - `strcmp()`, `strchr()`, `strstr()`
  - `strcpy()`, `strncpy()`, `strtok()`

- memory copy functions:
  - `memccpy()`, `memcpy()`, `bcopy()`
  - `memchr()`, `memcmp()`, `bzero()`
  - `memmove()`, `memset()`, `bcmp()`

- allocation functions:
## Error message summary (memory analysis)

The following table shows a summary of potential error messages you might encounter during memory analysis:

<table>
<thead>
<tr>
<th>Message</th>
<th>Caused by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>no errors</td>
<td></td>
<td>The heap memory is corrupted.</td>
</tr>
<tr>
<td>allocator inconsistency -</td>
<td>A buffer overflow</td>
<td>A buffer overflow occurred in the heap.</td>
</tr>
<tr>
<td>Malloc chain is corrupted, pointers out of order</td>
<td></td>
<td>The heap memory is corrupted.</td>
</tr>
<tr>
<td>allocator consistency -</td>
<td>A buffer overflow</td>
<td>A buffer overflow occurred in the heap.</td>
</tr>
<tr>
<td>Malloc chain is corrupted, end before end pointer</td>
<td></td>
<td>The heap memory is corrupted.</td>
</tr>
<tr>
<td>pointer does not point to heap area</td>
<td>The illegal deallocation of memory.</td>
<td>You attempted to free non-heap memory.</td>
</tr>
<tr>
<td>possible overwrite - Malloc block header corrupted</td>
<td>A buffer overflow</td>
<td>A buffer overflow occurred in the heap.</td>
</tr>
<tr>
<td>allocator inconsistency -</td>
<td>A buffer overflow</td>
<td>A buffer overflow occurred in the heap.</td>
</tr>
<tr>
<td>Pointers between this segment and adjoining segments are invalid</td>
<td>The heap memory is corrupted.</td>
<td></td>
</tr>
</tbody>
</table>

*continued...*
## Error message summary (memory analysis)

<table>
<thead>
<tr>
<th>Message</th>
<th>Caused by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data has been written outside allocated memory block</td>
<td>A buffer overflow occurred in the heap.</td>
<td>The program attempted to write data to a region beyond allocated memory.</td>
</tr>
<tr>
<td>data in free’d memory block has been modified</td>
<td>Attempting to use memory that was previously freed.</td>
<td>The program is attempting to write to a memory region that was previously freed.</td>
</tr>
<tr>
<td>data area is not in use (can’t be freed or reallocated)</td>
<td>A buffer overflow occurred in the heap.</td>
<td>The heap memory is corrupted.</td>
</tr>
<tr>
<td>unable to get additional memory from the system</td>
<td>All memory resources are exhausted.</td>
<td>There are no more memory resources to allocate.</td>
</tr>
<tr>
<td>pointer points to the heap but not to a user writable area</td>
<td>A buffer overflow occurred in the heap.</td>
<td>The heap memory is corrupted.</td>
</tr>
<tr>
<td>allocator inconsistency - Malloc segment in free list is in-use</td>
<td>A buffer overflow occurred in the heap.</td>
<td>The heap memory is corrupted.</td>
</tr>
<tr>
<td>malloc region doesn’t have a valid CRC in header</td>
<td>A buffer overflow occurred in the heap.</td>
<td>The heap memory is corrupted.</td>
</tr>
<tr>
<td>free’d pointer isn’t at start of allocated memory block</td>
<td>An illegal deallocation of memory.</td>
<td>An attempt was made to deallocate the pointer that shifted from its original value when it was returned by the allocator.</td>
</tr>
</tbody>
</table>
Memory analysis GUI flags and corresponding environment variables

The following table shows a summary of Memory Analysis Tool (MAT) graphical user interface options (flags) and their corresponding environment variables:

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Where to find in MAT GUI</th>
<th>What option to set</th>
</tr>
</thead>
<tbody>
<tr>
<td>MALLOC_TRACEBT=5</td>
<td>Memory Tracing→Enable memory allocation/deallocation tracing</td>
<td>Limit back-trace depth to: 5</td>
</tr>
<tr>
<td>MALLOC_CKALLOC=1</td>
<td>Memory Errors→Enable error detection</td>
<td>Enable check on realloc()/free() argument</td>
</tr>
<tr>
<td>MALLOC_CKACCESS=1</td>
<td>Memory Errors→Enable error detection</td>
<td>Verify the parameters in the string and memory functions</td>
</tr>
<tr>
<td>MALLOC_CKCHAIN=1</td>
<td>Memory Errors→Enable error detection</td>
<td>Perform a full heap integrity check on every allocation/deallocation</td>
</tr>
<tr>
<td>MALLOC_WARN=0</td>
<td>Memory Errors→Enable error detection</td>
<td>When an error is detected: report the error and continue</td>
</tr>
<tr>
<td>MALLOC_DUMP_LEAKS=1</td>
<td>Memory Errors→Enable error detection</td>
<td>Perform leak check when process exits</td>
</tr>
</tbody>
</table>

*continued...*
## Memory analysis GUI flags and corresponding environment variables

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Where to find in MAT GUI</th>
<th>What option to set</th>
</tr>
</thead>
<tbody>
<tr>
<td>MALLOC_TRACE_MIN=20</td>
<td>Memory Tracing</td>
<td>Minimum allocation to trace: 20</td>
</tr>
<tr>
<td>MALLOC_BTDEPTH=10</td>
<td>Memory Error</td>
<td>Limit back-trace depth to: 10</td>
</tr>
<tr>
<td>MALLOC_TRACE=/dev/dbgmem</td>
<td>Memory Tracing</td>
<td>Enable memory allocation/deallocation tracing</td>
</tr>
<tr>
<td>MALLOC_ERRFILE=/dev/null</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MALLOC_FATAL=0</td>
<td>Memory Errors→Enable error detection</td>
<td>When an error is detected: report the error and continue</td>
</tr>
<tr>
<td>MALLOC_FILLAREA=1</td>
<td>Memory Errors→Enable error detection</td>
<td>Enable bounds checking (where possible)</td>
</tr>
<tr>
<td>MALLOC_TRACE_MAX=30</td>
<td>Memory Tracing</td>
<td>Maximum allocation to trace: 20</td>
</tr>
<tr>
<td>MALLOC_CTHREAD=1</td>
<td>Memory Errors→Target Settings</td>
<td>Create control thread</td>
</tr>
<tr>
<td>MALLOC_EVENTFILE=/dev/dbgmem</td>
<td>Memory Errors→Target Settings</td>
<td>Send events to:</td>
</tr>
<tr>
<td>MALLOC_STAT_BINS=2,4,8,16,32</td>
<td>Memory Snapshots→Bin counters</td>
<td>N/A</td>
</tr>
<tr>
<td>LD_PRELOAD=/tmp/libmalloc_g.so</td>
<td>Target Settings</td>
<td>Malloc library: /tmp/libmalloc_g.so</td>
</tr>
</tbody>
</table>
Using a file to log the trace

You can perform memory analysis on a running program, or you can log the trace to a file on the target system. The advantage of logging the trace is that doing so frees up qconn resources; you run the process and do the analysis later.

When analyzing the data from a log file, you can’t do any backtracing.

To log the trace to a file:

1. Open the Launch configuration and go to the Memory Analysis Tooling tab.
2. Expand the Target Settings.
3. By default, the Send traces to field is set to /dev/dbgmem, which the qconn agent reads. Specify the name of the file on the target system (e.g. /tmp/log.memory) where you’d like to send the traces instead.
4. Run your application on the target.
5. Copy the file back to the host, and then choose Import libmalloc_g events from the Session view’s right-click menu.

A dialog appears:
Analyzing a running program

6 Choose an existing session or create a new one.

7 Browse to the file that you copied from the target, and then click OK. The IDE reparses the file for viewing.

You can also do this on the command line by setting the appropriate environment variables. For example:

```
LD_PRELOAD=/tmp/libmalloc_g.so MALLOC_TRACE=/tmp/log.memory
```

Analyzing a running program

You can perform memory analysis on a running program, if that program was started using the debug malloc library and the proper environment variables. Once the program is running, you can attach the Memory Analysis perspective and gather your data.

For more information, see “Attaching to a running process.” in this chapter.
Analyzing a running program

If a program uses `fork`, the control thread of the Memory Analysis tool must be disabled because when you attach the Memory Analysis tool, it creates a second thread; however, `fork` only works with single threaded programs.

To disable the control thread option for memory analysis:

1. From an existing launch configuration, select the **Tools** tab.
2. If Memory Analysis Tooling is not currently enabled, select **Add/Delete Tool**, select **Memory Analysis Tooling**, then click **OK**.
3. Expand the **Target Settings**.
4. Disable the **Create control thread** option if it is currently enabled.
5. Click **Apply**.
6. Click **Run**.

For information about the **Create control thread** option, see “Analyzing your program” in this chapter.

**Launching with debug malloc**

To start a program using the debug malloc library:

Launch the program using the **LD_PRELOAD** (set to the debug malloc library) and **MALLOC_CTHREAD** (set to 1) environment variables:

`LD_PRELOAD=/tmp/libmalloc_g.so MALLOC_CTHREAD=1 ./my_app`

**Attaching to a running process**

As mentioned above, you can analyze memory events and traces for a running process. To do this, you need to create a launch profile, as follows:
Analyzing shared objects

1  If the Run menu doesn’t include a Profile entry, add it like this:
   1a  Choose Customize Perspective ... from the Window menu.
   1b  Choose the Commands tab.
   1c  In the list of checkboxes, enable the Profile checkbox.
   1d  Click OK.

2  Choose Run→Profile....

3  Set up the launch configuration.

After launching, a dialog appears with a list of the running processes on the box. Choose the process you want to attach to; the Session view then lists it. When you select the process in the Session view, the editor displays the information about it.

When you’re done, disconnect from the process and let it continue.

Analyzing shared objects

In order to analyze shared objects, you must set up the Memory Analysis Tooling tab in your launch configuration:

- To analyze all shared objects, expand the Target Settings and enable Use dladdr to find dll names.
- To analyze specific shared objects, expand Library search paths and fill in the paths to the directories containing the shared objects:
You can also check the **Recurse** button to search all subdirectories under a given path.

In the Session View, you can expand your session, expand your process, and then select a shared object to view its memory events and traces in a new tab in the editor.

**Associated views**

The QNX Memory Analysis perspective includes the following views:

- Memory analysis session
- Memory analysis editor

**Memory analysis session**

The Session view lets you manage your memory analysis sessions. You can load these sessions into the Memory Analysis perspective, so you can search for memory management errors in your application.
The view lists all of the memory analysis sessions that you’ve created in your workspace while running programs with the Memory Analysis tool active. Each session is identified by a name, date stamp, and an icon that indicates its current state.

The icons indicate:

- ![Icon](image) This memory analysis session is open and can be viewed in the Memory Analysis editor.
- ![Icon](image) This session is closed and cannot currently be viewed.
- ![Icon](image) This session is still running on the target; you can select the session and view its incoming traces.

If the session is running, you may need to close and reopen the tabs at the top of the editor periodically to refresh the information in the Errors and Statistics panes.

![Icon](image) The traces and events are being indexed. This icon appears only if you stop the memory analysis session or your process...
Memory analysis session

terminates. If your process terminates, the running icon may still be displayed while the database is registering the events and traces; when this is done, the indexing icon appears. Wait until indexing is finished, or the information might be incomplete.

Right-clicking on a connected session displays a menu with several options:

- View
- Disconnect
- Import libmalloc_g events
- Delete
- Rename
- Properties

Right-clicking on a disconnected session displays a menu with several options:

- Connect
- Import libmalloc_g events
- Delete
- Rename
- Properties

Connecting to a session

Memory analysis sessions must be “connected” before they can be viewed in the Memory Analysis editor. To connect a session:

1. Right-click the session in the Session view.
2. Choose Connect from the pop-up menu.

After a moment, the session is connected.
Deleting a session

To delete a session:

1. Right-click the session in the Session view.
2. Choose Delete from the pop-up menu.

The IDE deletes the memory analysis session.

Disconnecting from a session

To disconnect a session and recover the resources it uses while connected:

1. Right-click the session in the Session view.
2. Choose Disconnect from the pop-up menu.

After a moment, the session is disconnected.

Displaying information about a session

To view information about a session:

1. Right-click the session in the Session view.
2. Choose Properties from the pop-up menu.

The IDE displays a Properties dialog for that memory analysis session:
Memory analysis session

Filtering information for a session

Occasionally, there may be too much information in a Memory Analysis session, and you might want to filter some of this information to narrow down your search for memory errors, events, and traces.

To filter out Memory Analysis session information:

1. Expand your Memory Analysis session in the session view.
2. Select specific session components, such as library and/or thread, that you want to filter on. You can double-click any of the session components to open a corresponding Memory Analysis Allocations pane containing memory events and traces that belong to the selected component.
Renaming a session

To rename a memory analysis session:

1. Right-click the session in the Session view.
2. Choose Rename from the pop-up menu.
   The IDE displays the Rename Session dialog.

3. Enter a new name for the session, then click OK to change the session’s name. Click Cancel to leave the name unchanged.

Viewing a session

To view a connected session in the Memory Analysis editor:

   Double-click the session in the Session view.

   The IDE opens the memory analysis session in the Memory Analysis editor.

Import libmalloc_g events

You’ll use this item after you’ve logged trace events to a file on the target system and copied the file to your host system. For more information, see “Using a file to log the trace” in the Finding Memory Errors chapter.

Memory analysis editor

When you view a connected memory analysis session, the Memory Analysis perspective opens that session in the main editor area of the IDE:
Memory analysis editor

The top half of the window shows details for the data selected in the bottom half, which is an overview of the entire memory analysis session data set:
Memory analysis editor

The details include a table of information about the allocations. If you select an allocation, a vertical line indicates its position in the Details chart, and the backtrace (if available) is displayed below the table. If you click on a backtrace, the editor displays the associated source code (if available) in another tab.

The icons in the table indicate the type of allocation or deallocation:

- Allocation with a matching deallocation.
- Deallocation with a matching allocation.
- Allocation without a matching deallocation.
- Deallocation without a matching allocation.

The Allocations Overview can be very wide, so it could be broken into pages. You can use the Page field to move from one page to another, and you can specify the number of points to display on each page.
Memory analysis editor

If the process does many allocations and deallocations, it could take some time for the traces and events to be registered, indexed, and displayed.

The tabs at the bottom let you switch between several different data views:

- **Allocations** — trace information about allocations and deallocations.
- **Errors** — leaks and other memory errors.
- **Bins** — counters that track the general size of allocations and deallocations.
- **Bands** — counters that track the allocator’s preallocated memory bands.
- **Usage** — information about the application’s memory usage over time.
- **Trace Details** — replays the details of the selected bins, bands, or usage, synchronized with the allocations.
- **Statistics** — detailed information about every memory event.
- **Settings** — settings for the running process. This tab is displayed only if the process is still running.

**Selecting data**

To select data in the overview:

Click and drag over the region you’re interested in.

The Memory Analysis perspective updates the details to reflect the data region you’ve selected.
Memory analysis editor

Controlling the view

The Memory Analysis editor has several icons that you can use to control the view:

<table>
<thead>
<tr>
<th>Use this icon:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Set the Chart and Detail Pane to a horizontal layout, one beside the other</td>
</tr>
<tr>
<td></td>
<td>Set the Chart and Detail Pane to a vertical layout, one above the other</td>
</tr>
<tr>
<td></td>
<td>Display the Detail Pane if it’s currently hidden</td>
</tr>
<tr>
<td></td>
<td>Hide the Detail Pane so the Chart pane has more display room</td>
</tr>
<tr>
<td></td>
<td>Hide the Chart pane so the Detail Pane has more display room</td>
</tr>
<tr>
<td></td>
<td>Toggle the Overview pane on and off</td>
</tr>
</tbody>
</table>

Controlling the overview

You can control the Overview pane through its context menu, which is displayed when you right-click on the Overview pane:
Memory analysis editor

This menu includes:

**By Timestamp**
Display the events sorted by their timestamp. Because several memory events can occur with the same timestamp, this might present the events in a confusing order (for example, a buffer’s allocation and deallocation events could be shown in the wrong order if they happen during the sampling interval).

**By Count**
Display events sorted by their event index. This is the default ordering in the Overview pane.

**Filters...**
Filter the displayed events by size, type, or both. You can also hide the matching allocations and deallocations, so that you see only the unmatched ones:

![Memory Events Filter Dialog](image)

- **Hide matching allocation/deallocation pair**
- **From allocation size:**
  - 0
- **To allocation size:**
  - 0

**Show memory events of type:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc</td>
<td>memory allocation</td>
</tr>
<tr>
<td>calloc</td>
<td>memory allocation (served)</td>
</tr>
<tr>
<td>realloc-alloc</td>
<td>memory allocation (during reallocation)</td>
</tr>
<tr>
<td>realloc-free</td>
<td>memory deallocation (during reallocation)</td>
</tr>
<tr>
<td>free</td>
<td>memory deallocation</td>
</tr>
</tbody>
</table>

[318] Chapter 10 • Finding Memory Errors
**Memory analysis editor**

**Zoom In**  
Zoom in on the selected range of events.

**Zoom Out**  
Zoom out to the set of memory events that you previously zoomed in on.

**Controlling the detail pane**

You can control the Detail pane through its context menu:

1. Right-click on the Detail pane.

2. Choose a graph from the **Chart Types** menu:
   - **BarChart** — a plain bar chart

   ![BarChart](image)

   - **BarChart_3D** — a 3D bar chart

   ![BarChart_3D](image)
Memory analysis editor

- **Differentiator** — a plain differentiator chart

- **Differentiator _3D** — a 3D differentiator chart

**Allocations tab**

As described above, the Allocations pane shows you allocation and deallocation events over time. Select a range of events to display a chart and details for those events.

**Errors tab**

The Errors pane shows any memory errors (in red) or leaks (in blue) detected while collecting statistics. Select a line in the top pane to see a function backtrace in the lower pane.
Bins tab

The allocator keeps counters for allocations of various sizes to help gather statistics about how your application is using memory. Blocks up to each power of two (2, 4, 8, 16, etc. up to 4096) and “large” blocks (anything over 4 KB) are tracked by these counters.

The Bins pane shows the values for these counters over time:
You can disable a counter’s display by clicking its circle above the pane’s title. Click the circle again to enable that counter’s display.

When the Bins pane is displayed, the Chart pane shows allocations and deallocations for each bin at the time selected in the Detail pane. The Detail pane lists memory events for the selected region of the Bins pane.

The Bins pane includes these additional buttons:

- **Play**: Play the selected range of the Use Bins; the Bins Statistics chart displays the usage dynamically.
- **Stop**: Stop the playing.

Tracing can be slow, and it may also change the timing of the application, because of the logging that’s done for each allocation and deallocation.
deallocation. You might want to do a first pass with the bins snapshots enabled to determine the “hot spots or ranges”, and on the second pass reduce the tracing to a certain range (minimum, maximum) to filter and reduce the log set.

**Bands tab**

For efficiency, the QNX allocator preallocates “bands” of memory (small buffers) for satisfying requests for small allocations. This saves you a trip through the kernel’s memory manager for small blocks, improving your performance.

The bands handle allocations of up to 16, 24, 32, 48, 64, 80, 96, and 128 bytes in size, and activity in these bands is shown on the Bands pane:
Memory analysis editor

Usage tab

The Usage pane shows your application’s overall memory usage over time.

Trace Details tab

This tab plays back the allocations, synchronized with the bins, bands, or usage, depending on what you last selected. You can use this display to look for memory leaks (e.g. bins where the number of allocations is much greater than the number of deallocations):
Statistics tab

The Statistics pane gives you several different statistics views for the Memory Analysis session.

The Allocations pane shows the number of calls to different kinds of allocations, plus a count of each allocation for a given number of bytes:
Memory analysis editor

The Backtraces pane shows you a list of memory event points in your application. Select one to display a function backtrace for that event:
The Outstanding traces pane shows allocations that weren’t deallocated when the trace ended. These aren’t necessarily errors.
The Errors pane shows you a list of the different types of memory error encountered while running your application.
Settings tab

This tab lets you change the settings for a running process.
Memory analysis editor

For information about the settings, see “Analyzing your program” in the Finding Memory Errors chapter.

The additional icons (from left to right) let you:

- get leaks
- get snapshots
- get traces
- apply the settings
Chapter 11

Getting System Information

In this chapter...

Introduction 333
What the System Information perspective reveals 334
Logging system information 337
Associated views 341
Controlling your system information session 342
Examining your target system’s attributes 346
Watching your processes 348
Examining your target system’s memory (inspecting virtual address space) 355
Tracking heap usage 359
Examining process signals 363
Getting channel information 363
Tracking file descriptors 366
Tracking resource usage 367
Tracking the use of adaptive partitioning 373
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 4 GB. 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
Logging system information

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.

Logging system information

You can gather system information from a Neutrino target and log it to a file, and then view it later in the IDE. Here’s how:

1. Right-click your target in the Target Navigator, and then choose Log With...→Log from the menu.

2. Select System Information Logging Configuration, and then select the New launch configuration icon to create a Log configuration.

3. In the Main tab of the log configuration, select the location where you’d like to store the log file.

4. Select the mode to use:
   - Snapshot mode collects all the requested data, and then stops.
Logging system information

- Continuous mode collects the data, and then continues to collect any changes to the data for the requested period of time at an interval provided (the default is 1 second).

5 Select the Neutrino target and optionally the processes you wish to collect data for.

6 If you wish, switch to the **Logging Options** tab and select the level of information you require:

7 Select **Log**.

Here are a few things to consider when setting up your log configuration:
• Some types of data require other data. For example, if you want to collect any of the process-level data, you must select Processes in the list of system-level data, as shown in the example of the Logging Options tab above. Similarly, if you want to collect thread-level data, you must select Threads in the list of process-level data.

• If you select specific processes for logging, the IDE doesn’t log process data for any new processes (e.g. process IDs show as -1). If you wish to log all processes, including those created during the logging operation, don’t select any processes in the process-selection area on the Main tab of the log configuration.

**Viewing captured system information**

Once the logging process has begun, you’ll see a progress monitor for it in the Progress view and the lower right progress area of the main IDE window. You can cancel the logging at any time through the Progress view.

When the logging operation finishes, the IDE presents the captured data as a target in the System Information History View. This view behaves the same way as the Target Navigator view; selecting the target or one or more processes causes the System Information views to display the corresponding data from the log.
Logging system information

To view the data captured over a period of time in continuous mode, drag the time index slider at the bottom of the System Information History view to the point in time where you’d like to view the data; the views update to display the data at that point in time.

To view a log file from a previous logging session, select the Search log files button in the toolbar area of the System Information History View. This presents you with a dialog showing a list of the log files the IDE has found:
In this dialog, you can set search paths for the IDE to use to find log files that you can load into the System Information perspective. By default any log configurations that you've used to gather information are displayed, along with any log files that were created using it. To load a log file, select it in the tree, and then select **Open Log**; when the loading is complete, the data from the log file appears as a target in the System Information History view.

**Associated views**

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>
</tbody>
</table>
### Controlling your system information session

<table>
<thead>
<tr>
<th>To:</th>
<th>Use this view:</th>
</tr>
</thead>
<tbody>
<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>
<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>
<tr>
<td>Examine your system’s use of adaptive partitioning</td>
<td>APS View</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 ( chí ) 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
Controlling your system information session

Target Navigator are gray when their data is stale and needs refreshing.

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:

- Connection Information
- Malloc Information
- Memory Information
- Process Information
- Signal Information

To control the display in the Information views:

In the Target Navigator view, expand a target and select a process:

![Target Navigator](image)

The currently-displayed Information view is updated to show information about the selected process.
Controlling your system information session

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.

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.

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
Watching your processes

Click the Highlight button (📝) in the view’s toolbar to highlight changes to the display since the last update.

You can change the highlight color in the Colors and Fonts preferences (Window → Preferences → General → Appearance → Colors and Fonts).

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 a multicore system, the pane lists CPU information for each core or processor.

System Memory pane

The System Memory pane displays your system’s total memory and free memory in numerical and graphical form.

Processes panes

The Processes panes display the process name, code and data size, the data usage delta, total CPU usage since starting, the CPU usage delta, and the process’s start date and time for the processes running on your selected target. The panes let 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 state of each thread:
Watching your processes

The Process Information view includes the following panes:

- Thread Details
- Environment Variables
- Process Properties

Chapter 11 • Getting System Information 349
Watching your processes

Click the Highlight button ( ) in the view’s toolbar to highlight changes to the display since the last update.

You can change the highlight color in the Colors and Fonts preferences (Window→Preferences→General→Appearance→Colors and Fonts).

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:
Watching your processes

3 You can:

- Add entries to the view by selecting items from the Available Items list and clicking **Add**.
- 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.

If you right-click on a thread in the Thread Details pane, the menu includes items that let you specify the thread’s priority and scheduling algorithm, name, CPU affinity, and inherited CPU affinity:
Watching your processes

Setting the priority and scheduling algorithm:

For more information about the available priorities and scheduling algorithms, see “Thread scheduling” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

If you’ve installed the Core OS 6.3.2 on the target, you can give the thread a name:
Watching your processes

You can also set the runmask that the thread’s children will inherit:

and its own runmask:
Watching your processes

For more information, see the Multicore Processing User’s Guide.

If you right-click on a process in the target navigator or the Thread Details pane, you get similar options, except for setting the thread name. The settings you make apply to all of the process’s threads.

**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`
Examining your target system’s memory (inspecting virtual address space)

 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 system’s memory
(inspecting virtual address space)

The following 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)
- Memory Information view (for examining virtual address space)

Virtual address space

The Memory Information view displays the memory used by the process you select in the Target Navigator view:
Examining your target system's memory (inspecting virtual address space)

The view shows the following major categories of memory usage:

- **Stack (red)**
  - guard (light)
  - unallocated (medium)
  - allocated (dark)

- **Program (royal blue)**
  - data (light)
  - code (dark)
Examining your target system’s memory (inspecting virtual address space)

- 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>The name of the category.</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 mmap() function’s flags and prot arguments in the Neutrino Library Reference.</td>
</tr>
</tbody>
</table>
Examining your target system's memory (inspecting virtual address space)

Offset The memory block’s offset into shared memory, which is equal to the `mmap()` function’s `off` argument.

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.

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
Tracking heap usage

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.

Tracking heap usage

The following 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)
- 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:
Tracking heap usage

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**

---

360 Chapter 11 • Getting System Information
Tracking heap usage

- 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 4 KB (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.
Tracking heap usage

Total `malloc` and frees

The total number of calls that effectively allocate or free memory. For example, if your program 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 $130/160$. 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 1 KB. Thus, two 512-byte allocations made over several seconds trigger one update.
Examining process signals

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.

Examining process signals

The Signal Information view shows the signals for the processes selected in the Target Navigator view.

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.
Getting channel information

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:
Tracking file descriptors

<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>

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>devc-con (?)</td>
<td>lrw</td>
<td>0</td>
<td>/dev/cont1</td>
</tr>
<tr>
<td>1</td>
<td>devc-con (?)</td>
<td>lrw</td>
<td>0</td>
<td>/dev/cont1</td>
</tr>
<tr>
<td>2</td>
<td>devc-con (?)</td>
<td>lrw</td>
<td>0</td>
<td>/dev/cont1</td>
</tr>
<tr>
<td>3s</td>
<td>procrire (1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2s</td>
<td>devc-ser/8250(77835)</td>
<td>--</td>
<td>0</td>
<td>/dev/ser1</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.
Tracking resource usage

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.

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

## Getting System Information

<table>
<thead>
<tr>
<th>Process Name</th>
<th>Start Time</th>
<th>CPU Usage</th>
<th>Uptime (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>procinfo (1)</td>
<td>Mon Oct 23 13:43:15 EDT 2006 8m 35s 653ms</td>
<td>98.78%</td>
<td></td>
</tr>
<tr>
<td>lin (2)</td>
<td>Mon Oct 23 17:43:22 EDT 2006 947ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>sluggish (3)</td>
<td>Mon Oct 23 13:43:16 EDT 2006 1ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>mqueue (12255)</td>
<td>Mon Oct 23 13:43:19 EDT 2006 947ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>pt-bios (5)</td>
<td>Mon Oct 23 13:43:16 EDT 2006 8ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>devfs-odb (6)</td>
<td>Mon Oct 23 13:43:16 EDT 2006 481ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>devvc-conv (7)</td>
<td>Mon Oct 23 13:43:19 EDT 2006 11ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>pipe (4104)</td>
<td>Mon Oct 23 13:43:19 EDT 2006 2ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>fusefs (237577)</td>
<td>Mon Oct 23 17:43:24 EDT 2006 4ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>dhcp_client (102410)</td>
<td>Mon Oct 23 17:43:20 EDT 2006 1ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>devvc-server (77835)</td>
<td>Mon Oct 23 17:43:20 EDT 2006 2ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>devvc-fdc (156508)</td>
<td>Mon Oct 23 17:43:22 EDT 2006 2ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>devvc-par (77337)</td>
<td>Mon Oct 23 17:43:20 EDT 2006 947ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>login (307214)</td>
<td>Mon Oct 23 17:43:27 EDT 2006 2ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>iso-net (77839)</td>
<td>Mon Oct 23 17:43:20 EDT 2006 2s 681ms</td>
<td>0.51%</td>
<td></td>
</tr>
<tr>
<td>devchk (294428)</td>
<td>Mon Oct 23 17:43:27 EDT 2006 191ms</td>
<td>0.04%</td>
<td></td>
</tr>
<tr>
<td>spoolder (91225)</td>
<td>Mon Oct 23 17:43:20 EDT 2006 13ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>devvc-ply (106514)</td>
<td>Mon Oct 23 17:43:22 EDT 2006 3ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>random (110311)</td>
<td>Mon Oct 23 17:43:22 EDT 2006 3ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>duniper (122908)</td>
<td>Mon Oct 23 17:43:22 EDT 2006 947ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>qconn (139285)</td>
<td>Mon Oct 23 17:43:22 EDT 2006 1s 867ms</td>
<td>0.36%</td>
<td></td>
</tr>
<tr>
<td>login (13306)</td>
<td>Mon Oct 23 17:43:22 EDT 2006 3ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>login (139267)</td>
<td>Mon Oct 23 17:43:22 EDT 2006 2ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>inetd (139230)</td>
<td>Mon Oct 23 17:43:22 EDT 2006 2ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>login (139289)</td>
<td>Mon Oct 23 17:43:22 EDT 2006 1ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>Photon (198444)</td>
<td>Mon Oct 23 17:43:22 EDT 2006 11ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>pfmon (217117)</td>
<td>Mon Oct 23 17:43:23 EDT 2006 26ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>iso-graphics (266271)</td>
<td>Mon Oct 23 17:43:25 EDT 2006 495ms</td>
<td>0.19%</td>
<td></td>
</tr>
<tr>
<td>pwn (459556)</td>
<td>Mon Oct 23 17:43:46 EDT 2006 3ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>shell (480352)</td>
<td>Mon Oct 23 17:43:47 EDT 2006 49ms</td>
<td>0.01%</td>
<td></td>
</tr>
<tr>
<td>bloader (487457)</td>
<td>Mon Oct 23 17:43:48 EDT 2006 457ms</td>
<td>0.09%</td>
<td></td>
</tr>
<tr>
<td>xserver (487465)</td>
<td>Mon Oct 23 17:43:48 EDT 2006 3ms</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>server (487459)</td>
<td>Mon Oct 23 17:43:48 EDT 2006 4ms</td>
<td>0.00%</td>
<td></td>
</tr>
</tbody>
</table>
Click the Highlight button ( ) in the view’s toolbar to highlight changes to the display since the last update.

You can change the highlight color in the Colors and Fonts preferences (Window → Preferences → General → Appearance → Colors and Fonts).

General Resources display

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

<table>
<thead>
<tr>
<th>Process Name</th>
<th>CPU Usage (%)</th>
<th>Data</th>
<th>File Descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td>procinfo (1)</td>
<td>99.10</td>
<td>0</td>
<td>68</td>
</tr>
<tr>
<td>tinit (2)</td>
<td>0.00</td>
<td>84k</td>
<td>0</td>
</tr>
<tr>
<td>slogger (3)</td>
<td>0.00</td>
<td>152k</td>
<td>3</td>
</tr>
<tr>
<td>inqueue (12292)</td>
<td>0.00</td>
<td>84k</td>
<td>3</td>
</tr>
<tr>
<td>ps-bsd (5)</td>
<td>0.00</td>
<td>84k</td>
<td>3</td>
</tr>
<tr>
<td>dev-log (6)</td>
<td>0.10</td>
<td>233M</td>
<td>6</td>
</tr>
<tr>
<td>devc-con (7)</td>
<td>0.00</td>
<td>120k</td>
<td>3</td>
</tr>
<tr>
<td>pipe (4104)</td>
<td>0.00</td>
<td>84k</td>
<td>3</td>
</tr>
<tr>
<td>forcelog (237577)</td>
<td>0.00</td>
<td>120k</td>
<td>7</td>
</tr>
<tr>
<td>dhcp-client (102410)</td>
<td>0.00</td>
<td>140k</td>
<td>5</td>
</tr>
<tr>
<td>devc-serv250 (77895)</td>
<td>0.00</td>
<td>116k</td>
<td>3</td>
</tr>
<tr>
<td>devc-fdi (100530)</td>
<td>0.00</td>
<td>260k</td>
<td>3</td>
</tr>
<tr>
<td>devc-par (77837)</td>
<td>0.00</td>
<td>120k</td>
<td>4</td>
</tr>
<tr>
<td>login (307214)</td>
<td>0.00</td>
<td>84k</td>
<td>3</td>
</tr>
<tr>
<td>is-net (77839)</td>
<td>0.50</td>
<td>564k</td>
<td>5</td>
</tr>
<tr>
<td>dev-hirun (204920)</td>
<td>0.00</td>
<td>12k</td>
<td>5</td>
</tr>
<tr>
<td>spooger (341315)</td>
<td>0.00</td>
<td>84k</td>
<td>4</td>
</tr>
<tr>
<td>devc-ply (105514)</td>
<td>0.00</td>
<td>216k</td>
<td>3</td>
</tr>
<tr>
<td>random (110511)</td>
<td>0.00</td>
<td>324k</td>
<td>3</td>
</tr>
<tr>
<td>dumper (122900)</td>
<td>0.00</td>
<td>84k</td>
<td>3</td>
</tr>
<tr>
<td>qcomm (139085)</td>
<td>0.20</td>
<td>288k</td>
<td>20</td>
</tr>
<tr>
<td>login (143882)</td>
<td>0.00</td>
<td>84k</td>
<td>3</td>
</tr>
<tr>
<td>logl (139267)</td>
<td>0.00</td>
<td>84k</td>
<td>3</td>
</tr>
<tr>
<td>inetlog (130288)</td>
<td>0.00</td>
<td>116k</td>
<td>7</td>
</tr>
<tr>
<td>log (139289)</td>
<td>0.00</td>
<td>84k</td>
<td>3</td>
</tr>
<tr>
<td>photon (188444)</td>
<td>0.00</td>
<td>116k</td>
<td>3</td>
</tr>
<tr>
<td>phfour (217117)</td>
<td>0.00</td>
<td>892k</td>
<td>7</td>
</tr>
<tr>
<td>iso-graphics (266271)</td>
<td>0.00</td>
<td>3816k</td>
<td>14</td>
</tr>
<tr>
<td>psaw (405534)</td>
<td>0.00</td>
<td>196k</td>
<td>5</td>
</tr>
<tr>
<td>shell (450592)</td>
<td>0.00</td>
<td>680k</td>
<td>9</td>
</tr>
<tr>
<td>blader (407457)</td>
<td>0.00</td>
<td>5420k</td>
<td>7</td>
</tr>
<tr>
<td>unsawatch (487458)</td>
<td>0.00</td>
<td>133k</td>
<td>6</td>
</tr>
<tr>
<td>seve (467459)</td>
<td>0.00</td>
<td>203k</td>
<td>6</td>
</tr>
</tbody>
</table>
Click the Highlight button (🖌️) in the view’s toolbar to highlight changes to the display since the last update.

You can change the highlight color in the Colors and Fonts preferences
(Window → Preferences → General → Appearance → Colors and Fonts).

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

<table>
<thead>
<tr>
<th>Process Name</th>
<th>Heap</th>
<th>Code</th>
<th>Data</th>
<th>Lib Code</th>
<th>Lib Data</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>procfs (1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tinit (2)</td>
<td>10K</td>
<td>12K</td>
<td>15K</td>
<td>36K</td>
<td>12K</td>
<td>8K</td>
</tr>
<tr>
<td>tulex (3)</td>
<td>60K</td>
<td>12K</td>
<td>4K</td>
<td>36K</td>
<td>12K</td>
<td>8K</td>
</tr>
<tr>
<td>pipe (4)</td>
<td>60K</td>
<td>40K</td>
<td>8K</td>
<td>36K</td>
<td>12K</td>
<td>8K</td>
</tr>
<tr>
<td>devb-aoe (5)</td>
<td>233M</td>
<td>52K</td>
<td>233M</td>
<td>62K</td>
<td>4K</td>
<td>76K</td>
</tr>
<tr>
<td>devc-rc (7)</td>
<td>100K</td>
<td>60K</td>
<td>120K</td>
<td>360K</td>
<td>12K</td>
<td>8K</td>
</tr>
<tr>
<td>pipe (104)</td>
<td>60K</td>
<td>16K</td>
<td>64K</td>
<td>360K</td>
<td>12K</td>
<td>40K</td>
</tr>
<tr>
<td>fontsleuth (237577)</td>
<td>60K</td>
<td>40K</td>
<td>120K</td>
<td>360K</td>
<td>12K</td>
<td>40K</td>
</tr>
<tr>
<td>dhcp (10010)</td>
<td>88K</td>
<td>44K</td>
<td>140K</td>
<td>475K</td>
<td>40K</td>
<td>12K</td>
</tr>
<tr>
<td>devc-sera (77535)</td>
<td>100K</td>
<td>44K</td>
<td>116K</td>
<td>360K</td>
<td>12K</td>
<td>8K</td>
</tr>
<tr>
<td>devb-fac (106508)</td>
<td>244K</td>
<td>24K</td>
<td>280K</td>
<td>528K</td>
<td>32K</td>
<td>60K</td>
</tr>
<tr>
<td>devc-ptr (77037)</td>
<td>100K</td>
<td>44K</td>
<td>120K</td>
<td>360K</td>
<td>12K</td>
<td>12K</td>
</tr>
<tr>
<td>login (307214)</td>
<td>60K</td>
<td>16K</td>
<td>84K</td>
<td>360K</td>
<td>12K</td>
<td>8K</td>
</tr>
<tr>
<td>ionet (77035)</td>
<td>520K</td>
<td>64K</td>
<td>560K</td>
<td>657K</td>
<td>32K</td>
<td>64K</td>
</tr>
<tr>
<td>devn-rmn (294926)</td>
<td>60K</td>
<td>80K</td>
<td>124K</td>
<td>1340K</td>
<td>48K</td>
<td>24K</td>
</tr>
<tr>
<td>spool (94225)</td>
<td>60K</td>
<td>16K</td>
<td>64K</td>
<td>360K</td>
<td>12K</td>
<td>8K</td>
</tr>
<tr>
<td>devc-cpy (100514)</td>
<td>200K</td>
<td>44K</td>
<td>216K</td>
<td>360K</td>
<td>12K</td>
<td>8K</td>
</tr>
<tr>
<td>random (110611)</td>
<td>202K</td>
<td>20K</td>
<td>324K</td>
<td>503K</td>
<td>26K</td>
<td>20K</td>
</tr>
<tr>
<td>cluser (12900)</td>
<td>60K</td>
<td>20K</td>
<td>84K</td>
<td>360K</td>
<td>12K</td>
<td>8K</td>
</tr>
<tr>
<td>qconn (139285)</td>
<td>188K</td>
<td>100K</td>
<td>236K</td>
<td>475K</td>
<td>40K</td>
<td>20K</td>
</tr>
<tr>
<td>log (14362)</td>
<td>60K</td>
<td>16K</td>
<td>84K</td>
<td>360K</td>
<td>12K</td>
<td>8K</td>
</tr>
<tr>
<td>login (139267)</td>
<td>60K</td>
<td>16K</td>
<td>84K</td>
<td>360K</td>
<td>12K</td>
<td>8K</td>
</tr>
<tr>
<td>inetd (139280)</td>
<td>72K</td>
<td>36K</td>
<td>116K</td>
<td>475K</td>
<td>40K</td>
<td>8K</td>
</tr>
<tr>
<td>login (139289)</td>
<td>60K</td>
<td>16K</td>
<td>84K</td>
<td>360K</td>
<td>12K</td>
<td>8K</td>
</tr>
<tr>
<td>Photon (186444)</td>
<td>100K</td>
<td>60K</td>
<td>116K</td>
<td>360K</td>
<td>12K</td>
<td>8K</td>
</tr>
<tr>
<td>pfist (217117)</td>
<td>615K</td>
<td>404K</td>
<td>924K</td>
<td>903K</td>
<td>72K</td>
<td>29K</td>
</tr>
<tr>
<td>ioperf (254671)</td>
<td>732K</td>
<td>100K</td>
<td>301K</td>
<td>1453K</td>
<td>120K</td>
<td>33K</td>
</tr>
<tr>
<td>pwm (41053)</td>
<td>132K</td>
<td>80K</td>
<td>198K</td>
<td>360K</td>
<td>12K</td>
<td>8K</td>
</tr>
<tr>
<td>shell (130502)</td>
<td>520K</td>
<td>80K</td>
<td>680K</td>
<td>1975K</td>
<td>152K</td>
<td>20K</td>
</tr>
<tr>
<td>binding (179457)</td>
<td>132K</td>
<td>12K</td>
<td>542K</td>
<td>1715K</td>
<td>72K</td>
<td>12K</td>
</tr>
<tr>
<td>winswitch (657459)</td>
<td>68K</td>
<td>1892K</td>
<td>12K</td>
<td>1448K</td>
<td>56K</td>
<td>12K</td>
</tr>
<tr>
<td>server (457459)</td>
<td>104K</td>
<td>16K</td>
<td>322K</td>
<td>1615K</td>
<td>60K</td>
<td>12K</td>
</tr>
</tbody>
</table>
Tracking the use of adaptive partitioning

Click the Highlight button ( ) in the view’s toolbar to highlight changes to the display since the last update.

You can change the highlight color in the Colors and Fonts preferences
(Window → Preferences → General → Appearance → Colors and Fonts).

To learn more about the meaning of the values shown in the Memory Resources display, see the Finding Memory Errors chapter in this guide.

Tracking the use of adaptive partitioning

This view displays information about the adaptive partitioning scheduling on the target system.

For more information about adaptive partitioning, see

- the Adaptive Partitioning chapter of the System Architecture guide

The APS view shows the budget pie chart as well as the APS System parameters and Partition Information:
Tracking the use of adaptive partitioning

If you expand the APS System information item, the view displays the following:

The Partitions item includes the following:
Tracking the use of adaptive partitioning

You can drag and drop processes or threads to move them from one partition to another. This might cause other processes or threads to move as well.

The Partition Statistics item displays the following information:

The APS Bankruptcy item displays information about bankruptcies:
Tracking the use of adaptive partitioning

The pane at the bottom of the view displays graphical information:

- Partition budgets (in percentages):

- CPU usage by partition (in percentages):
Tracking the use of adaptive partitioning

- Critical time usage (in milliseconds):

If you right-click on your target, the menu includes some options for the adaptive partitioning scheduler:
Tracking the use of adaptive partitioning

This menu includes:

- Set APS Security:
Tracking the use of adaptive partitioning

For information about the flags, see “Scheduling policies” in the entry for `SchedCtl()` in the Neutrino Library Reference.

- Set APS Parameters:
Tracking the use of adaptive partitioning

These parameters control:

- the length of the sliding averaging window over which the adaptive partitioning scheduler calculates the CPU usage
- how the scheduler handles bankruptcies. For more information, see “Handling bankruptcies” in the entry for SchedCtl() in the Neutrino Library Reference.

- Modify Existing Partition:

The partition’s budget is a percentage of CPU usage, while the critical budget is in milliseconds.
Tracking the use of adaptive partitioning

- Create New Partition:

![Create New Partition](image)

The new partition’s budget is taken from its parent partition’s budget.

You can also get information about the usage of adaptive partitioning on your system over a specified period of time through the System Profiler perspective’s Partition Summary pane. For more information, see the “Analyzing Your System with Kernel Tracing” chapter in this guide.
Chapter 12

Analyzing Your System with Kernel Tracing

In this chapter...

Introducing the QNX System Profiler 385
Configuring a target for system profiling 389
Capturing instrumentation data in event log files 394
Viewing and interpreting the captured data 396
Filtering a profile 400
System Profiler use cases 401
Associated views 426
System Profiler editor 427
Bookmarks view 435
Client/Server CPU Statistics view 435
Condition Statistics view 436
CPU Migration pane 441
Event Owner Statistics view 442
General Statistics view 443
Raw Event Data pane 443
Overview view 444
Partition Summary pane 445
Thread State Snapshot pane 446
Timeline State Colors pane 447
Trace Event Log view 448
Why Running? pane 449
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:
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 a multicore machine (by watching threads as they migrate from one CPU or core 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 Log With... → Kernel Event Trace, which initiates the Log Configuration dialog. 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.

You can gather statistics 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.

Other components help you determine why a given thread is running, examine the migration of threads from one processor or core to another, and so on. For more details, see “Associated views,” later in this chapter.

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, to capture instrumentation data for analysis, the instrumented kernel (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:

```
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:
4  Select `proccto-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 can run a command-line utility (e.g. `tracelogger`) on your target to generate a log file, and then transfer
Configuring a target for system profiling

that log file back to your development environment for analysis. Or, you can capture events directly from the IDE using the Log Configuration dialog.

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 Log Configuration dialog 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.

Launching the Log Configuration dialog

To launch the Log Configuration dialog:

In the Target Navigator view, right-click a target, then select Log With… → Kernel Event Trace from the menu. If you don’t have the Target Navigator view open, choose Window → Show View → Other…, then QNX Targets → Target Navigator.
Configuring a target for system profiling

If you don’t already have a target project, you’ll have to create one.

To create a target project:

1. In the Target Navigator view, right-click and select **Add New Target**.
2. Specify the required information for your 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 Log Configuration dialog 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 number of **qconn** 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:

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

A 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.
Configuring a target for system profiling

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.

If your target is running QNX Neutrino 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 Neutrino 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 Log Configuration dialog), 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.
Capturing instrumentation data in event log files

The data logged with events comes in the following modes:

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:

Chapter 12 • Analyzing Your System with Kernel Tracing 395
Viewing and interpreting the captured data

- 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.

Log files are loaded in a nonmodal activity that you can cancel. This lets you load multiple log files at the same time, and log file data can be normalized as a background task when it’s out of order. The display will show data up to the point where the log file has been processed.
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.

Searches in the System Profiler occur in the background, with results displayed as they’re found. The results of each search operation are managed separately and are “overlaid” on top of one another, allowing multiple search results to be graphically viewed together in the System Profiler timeline editor pane. You can configure the table that’s displayed in the search result view to display only the interesting trace event fields. The content of this table can be cut and pasted to the system clipboard as CSV-format data.

The tracing of kernel events from the IDE is performed as a background task. You can monitor the progress of the trace by opening the Progress view.

In the System Profiler editor’s Timeline pane, you can navigate to the next or previous event limited to only those event owners. This lets you follow a sequence of events generated by a particular set of event owners (for example finding the next event owned by a thread, or the messages generated by a client and server).

In locations where single events have been identified (for example, the Trace Log view, Search Results view), you can navigate directly to the event location in the System Profiler timeline editor pane by double-clicking. The selection marker is moved to the event location and, if possible, the specific event owner is scrolled into view in the timeline editor pane.
Viewing and interpreting the captured data

The Navigate menu contains a Go To Type command that lets you jump directly to a specific type to allow developers to collaborate more easily with one another. Navigating by type provides direct navigation to the following: `namespace`, `typedef`, `class`, `enum`, `struct`, and `union`.

![Open Type](image)

The System Profiler’s Go To Event command.

These components of the QNX System Profiler are described in detail later in this chapter:

- System Profiler editor
Viewing and interpreting the captured data

- Bookmarks view
- Client/Server CPU Statistics view
- Condition Statistics view
- CPU Migration pane
- Event Owner Statistics view
- General Statistics view
- Raw Event Data pane
- Overview view
- Partition Summary pane
- Thread State Snapshot pane
- Timeline State Colors pane
- Trace Event Log view
- Why Running? pane

There are a number of additional components outside of the editor that you can use to examine the event data in more detail:

**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 $\text{MsgSend}$ events whose calling function corresponds to $\text{mmap()}$). 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.
Filtering a profile

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 is 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 in the Preferences dialog (choose Window→Preferences→QNX→System Profiler).

Properties view

This 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.

Filtering a profile

The IDE lets you filter profile data so that you can look at a subset of the captured information. You can specify filtering on the following items:

- processes
- events
- saved filters

To filter profile data:

1. After you’ve begun running your process(es) and started kernel logging for a project, you can select System Profiler→Display→Switch Pane→Timeline to change to the Timeline editor state.

2. Right-click on the Timeline canvas and select Filter.

3. Specify your desired filtering options on the following tabs:
Notice that the Timeline will dynamically change (for the specified time range) based on the filtering selections you make.
System Profiler use cases

- On the **Owners** tab, select only those processes that you want to observe system profile data for. Click **Deselect All** to quickly deselect all of the processes, and then you can select only those that you want to monitor.
- On the **Events** tab, you can specify the events that you want to filter on, such as interrupts, communication, kernel calls, and various other events. Click **Deselect All** to quickly deselect all of the events, and then you can select only those that you want to monitor.
- On the **Saved Filters** tab, you can filter in or out based on custom filters; these are pre-existing filters within the IDE. For example, if you select **CPU Usage** and then click the **Show Only** button, the IDE will filter out any event owners that didn’t use CPU time. Click the **Add** button to add the currently selected item to the list of items being filtered; the results will dynamically display in the Timeline. Click **Remove** to filter out the selected item; its corresponding data will be removed from the Timeline.

System Profiler use cases

This section describes some cases where you’d use the System Profiler:

- locating sources of high CPU usage
- mapping and isolating client CPU load from server CPU load
- examining interrupt latency
- locating events of interest

Locating sources of high CPU usage

In many cases you want to know where in time your CPU cycles are being consumed and who is doing the consuming. The System Profiler provides several tools to help extract this information and “drill down” to quickly and easily determine the source and distribution of CPU consumption.
System Profiler use cases

Requirements

To extract CPU usage metrics using the System Profiler tools, the captured log file must contain at a minimum, the Neutrino RUNNING thread state. If the RUNNING thread state is logged in wide mode, then additional information regarding CPU usage distribution over priority and partitions can also be calculated.

If you need to determine the CPU load caused by interrupts, then you must also log the Interrupt Entry/Exit events.

Procedure

To start, open the target log file in the System Profiler editor. By default the initial display should show the Summary editor pane; if this isn’t the case, then you can get to the Summary editor pane via the menu item System Profiler→Display→Switch Pane→Summary.

The Summary editor pane shows a high-level overview of the log file contents:
The System Activity section displays the distribution of time spent in the log file, carved into these broad categories:
System Profiler use cases

Idle  The amount of time that the idle thread(s) spent running in this log file.

Interrupts  The amount of time that has been spent servicing hardware interrupts in this log file.

System  The amount of time that has been spent making kernel calls (measured between kernel entry and exit events). This time doesn’t include any of the time spent handling hardware interrupts.

User  The amount of time that nonidle threads spend in the Neutrino RUNNING state, minus the time spent performing kernel calls or in interrupt handlers.

Using these metrics, you can get a rough estimate of how efficiently your system is performing (e.g. amount of idle time, ratio of system to user time, possible interrupt flooding).

The distribution of CPU usage over the time of the entire log file is displayed graphically in the Process & Thread Activity section overlaid with the volume of events that have been generated. This same data is also available as the Overview view accessed via Window→Show View→Other …→Overview.
System Profiler use cases

The peaks of this display indicate areas of particularly intense CPU usage and are the areas of most interest.

To focus on the particular threads that are causing these spikes, switch the editor display pane to the CPU Usage editor pane. You can do this via the menu item System Profiler→Display→Switch Pane→CPU Usage or directly using the editor pull down.

The CPU Usage editor display charts the CPU usage of consuming elements (threads and interrupts) over time and provides a tabular display showing the sum of this usage categorized by CPU, priority, or partition.
System Profiler use cases

By selecting multiple elements in the table, you can “stack” the CPU usage to see how threads and interrupts are interacting. For example, selecting the first few nonidle CPU consumers in this example provides the following display:

By selecting a region of the display, you can zoom in to the area of interest to further drill down into selected areas to better examine the profile of the CPU execution. As the display zooms in, the editor panel’s time bar is updated to show the new range of time being examined.
System Profiler use cases

This example has shown the CPU usage for process threads, but this technique applies equally well to individual interrupt handlers, which show up as CPU consumers in the same manner as threads.

The CPU Usage editor pane lets you isolate and assign CPU consumption behavior to specific threads very quickly and easily. With this information, you can generally use a more specialized, and application centric, tool such as the Application Profiler to look more closely at execution behavior and to drill down directly to the application source code.

**Mapping and isolating client CPU load from server CPU load**

There are many cases where high CPU load is traced back to server activity. However, in most cases what is required to reduce this CPU load isn’t to make the servers more efficient, but to look more closely at the client activity that is causing the servers to run.

**Requirements**
System Profiler use cases

Make sure you’ve read and understood “Locating sources of high CPU usage” before examining this use case.

In addition to the Neutrino RUNNING thread state, the log must contain the communication events SEND/RECEIVE/REPLY | ERROR. These communication events are used to establish the relationship between clients and servers.

Procedure

QNX Neutrino systems rely heavily on client/server communications most often performed via message passing. In these systems, clients send requests to servers asking them to do work on their behalf such as shown:

Here, A’s real CPU usage would be considered to be 2 units of time, B’s as 10, and C’s as 2 units of time. Since B and C are both acting as servers, they really execute only when there are clients generating requests for action. Most standard CPU Usage metrics don’t take this type of “on behalf of” work into consideration. However, if the goal of a kernel log file investigation is to locate the source or sources of CPU load, then this type of metric is invaluable in assigning “blame” for high CPU usage.

The System Profiler provides the Client/Server CPU Statistics view to help extract this type of “on behalf of” metric. You can activate this view via the Window→Show View→Other...→Client/Server CPU Statistics.

Once activated, the Client/Server CPU Statistics are gathered on demand, by default, targeting the full range of the target log file:
The default display of this view shows the simplified view that displays the RUNNING time (slightly different from the CPU Usage in that it doesn’t remove the time spent interrupted by interrupt handlers) that CPU consumers are consuming directly, indirectly, and summed together as a total:

In this case, it’s clear that while the qconn- Thread 1 isn’t consuming much CPU on its own, it’s imposing a significant amount of time on the system indirectly. If you compare this data to what the CPU Usage editor pane displays, you’ll see the difference in what’s reported:
System Profiler use cases

In the CPU Usage table, procnto- Thread 8 ranks ahead of qconn- Thread 1 in its usage. However, procnto is a pure server process, so we know that it consumes no CPU resources without being solicited to do so. We suspect that perhaps qconn- Thread 1 is driving procnto- Thread 1.

We can confirm this suspicion by looking at which servers qconn- Thread 1 is imposing CPU usage on. You can configure the Client/Server CPU Usage view to display all of the CPU consumers that are being imposed on (and by whom) by selecting Show all times from the view’s dropdown menu:

![Client/Server CPU Usage view](image)

The Client/Server CPU Usage view table changes to show all of the imposed-on servers that clients are communicating with. The servers are listed in the columns and the clients in the Owner column. Note
that this may result in a table with many columns (imposed on servers):

<table>
<thead>
<tr>
<th>Owner</th>
<th>Total Time</th>
<th>Self Time</th>
<th>Imposed Time</th>
<th>devcpty - Thread 1</th>
<th>Photo...</th>
<th>io-n...</th>
<th>proc...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Photon - Thread 1</td>
<td>3ms</td>
<td>3ms</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>devb-Idle - Thread 6</td>
<td>337us</td>
<td>337us</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>devc-pty - Thread 1</td>
<td>1ms</td>
<td>1ms</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>font-utils - Thread 3</td>
<td>44us</td>
<td>44us</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>in-graphics - Thread 3</td>
<td>3ms</td>
<td>337us</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>in-net - Thread 2</td>
<td>6ms</td>
<td>6ms</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>in-net - Thread 6</td>
<td>3ms</td>
<td>3ms</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>in-net - Thread 7</td>
<td>472us</td>
<td>472us</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>proc-to-int - CPU 1 idle</td>
<td>26us</td>
<td>26us</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>proc-to-int - Thread 2</td>
<td>20us</td>
<td>20us</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>proc-to-int - Thread 6</td>
<td>974us</td>
<td>974us</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>pthread - Thread 1</td>
<td>3ms</td>
<td>1ms</td>
<td>1ms</td>
<td>175us</td>
<td>331us</td>
<td>0ns</td>
<td>948us</td>
</tr>
<tr>
<td>pthread - Thread 1</td>
<td>3ms</td>
<td>2ms</td>
<td>1ms</td>
<td>150us</td>
<td>369us</td>
<td>0ns</td>
<td>975us</td>
</tr>
<tr>
<td>pvm - Thread 4</td>
<td>10ms</td>
<td>10ms</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>proc - Thread 1</td>
<td>4ms</td>
<td>4ms</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>proc - Thread 2</td>
<td>1ms</td>
<td>1ms</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>proc - Thread 5</td>
<td>2ms</td>
<td>2ms</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>proc - Thread 5</td>
<td>27ms</td>
<td>16ms</td>
<td>1ms</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>15ms</td>
</tr>
<tr>
<td>random - Thread 3</td>
<td>260us</td>
<td>260us</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
</tr>
<tr>
<td>server - Thread 1</td>
<td>208us</td>
<td>129us</td>
<td>79us</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>79us</td>
</tr>
<tr>
<td>shell - Thread 1</td>
<td>2ms</td>
<td>1ms</td>
<td>646us</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>646us</td>
</tr>
<tr>
<td>shell - Thread 2</td>
<td>4ms</td>
<td>2ms</td>
<td>1ms</td>
<td>0ns</td>
<td>0ns</td>
<td>0ns</td>
<td>227us</td>
</tr>
</tbody>
</table>

Here we can see that in fact nearly all of the time that **proc-to-int - Thread 8** is spending consuming CPU is due to requests coming from **qconn - Thread 1**, with only a minimal amount being imposed on it by another **qconn** thread, **qconn - Thread 6**.

This is to be expected, since in order to query the system, the **qconn** process must communicate with the kernel to extract the system state and all the process and thread information.

**Examining Interrupt Latency**

There are several different types of interrupt latency that you can measure in a system:
System Profiler use cases

- the time from the HW signal generation to the start of software processing
- the time it takes before a non-OS control function can be invoked in response to the interrupt
- the time it takes for a user thread to be activated in response to this type of external event

The System Profiler, as a type of software logic analyzer, helps you look at the timing of activities once the interrupt has been acknowledged by the operating system. In order to accurately measure the time between the signal generation and the acknowledgment of it, you need additional hardware tools.

Requirements

To measure interrupt service time (the time taken for the operating system to acknowledge the interrupt, handle it, and return to normal processing), you must log the Neutrino Interrupt Entry/Exit events.

If you're interested in the time from the operating system’s acknowledgment to a service handling routine, then you also need to capture the Interrupt Handler Entry/Exit events in the log file.

To properly gauge the latency in triggering a response in user code, you should also log the Neutrino thread READY and RUNNING states, in addition to the communication PULSE events, since these are often used to trigger a user application’s behavior in response to an interrupt.

Procedure

Interrupt activity is best viewed in the System Profiler editor using the Timeline editor pane. Open the target log file in the System Profiler editor. Switch to the Timeline editor pane via the menu item System Profiler→Display→Switch Pane→Timeline.

You should see a display that resembles the following. The details will of course be different, but the layout similar:
System Profiler use cases

This display shows the various event owners/sources (interrupts, interrupt handlers, processes and threads) as a tree with their associated events arranged horizontally as a timeline.

If you’ve logged Interrupt Handler Entry/Exit events, then you should be able to expand the interrupt entries to show the various handlers (more than one handler can be attached to service an interrupt source), such as the following:

Here you can see that the io-net process has attached to Interrupt 0x8c and that procnto has attached to Interrupt 0x800000000, which on this system is the timer interrupt firing once every millisecond or so.

You can determine how many interrupt events are occurring in this log file by using the General Statistics view. This view is part of the default System Profiler perspective, and you can also access it via Window→Show View→Other...→General Statistics.
System Profiler use cases

If you use the refresh button, this view extracts the event statistics for the entire log file (default), or for just the selected area if specified. Doing that results in the following display:

This table provides a breakdown for all of the event sources, showing the number of raw events and also the maximum, minimum, average, and total duration of the various Neutrino thread states in this log file.

If you’re interested in only the events associated with the timer interrupt (Interrupt 0x80000000), you can select that event owner in the Timeline editor pane:
Next, uncheck the **Show statistics for all elements** check box at the bottom of the General Statistics view:

The General Statistics view tables will show the content limited to just the selected event owners.

Using this technique, you can get an estimate of the rough order of magnitude of how many events you’re looking at in a log file, and in the case of interrupts, you can see some of the statistics about what the maximum, minimum, average, and total times spent were.

This display also lets you drill down further into the results, by allowing navigation in the Timeline editor pane directly to the maximum and minimum times, where you can look at the exact timing sequences. To do this, select one of the entries in the States table, and then right-click or use the toolbar to jump to the appropriate selection.

To look at the timing sequence of an interrupt, you usually have to zoom in on the timeline a significant amount to achieve an adequate level of visual detail, since interrupt processing is in general fast compared to the length of the log files. If you zoom into an area where a networking interrupt is being processed, the Timeline editor pane will change to look something like:
At this level of granularity, it also helps to see the trace event log concurrently with the Timeline editor pane. This is part of the standard System Profiler perspective, and you can access it using **Window→Show View→Other…→Trace Event Log.** The Trace Event Log and the Timeline editor pane are synchronized; when you change your cursor in the editor, the selection in the Trace Event Log view also changes and vice versa.

The selection synchronization is shown here. In the Trace Event Log view, we’ve selected the **Interrupt 0x8c** Entry event through to the **Interrupt 0x8c** Exit event. This represents the start to end of the processing of the interrupt event. In the timeline display, this selection is made and the timing measurement of 11.304 microseconds is displayed:
So the total interrupt handling time from start to end of the operating system interrupt service routine, including the event handler was 11.304 microseconds. If you want to just look at the handling time for interrupt handler attached by the \texttt{io-net} process, you can see that this time is only 8 microseconds. These times represent the earliest and latest points in time that can be measured before entering/exiting control of the software.

You can also see in this example that the \texttt{io-net} interrupt handler is returning a pulse that’s triggering something in the user’s application (event 13515) and that an \texttt{io-net} thread is then scheduled to service that request. You can also measure this latency to determine how long it takes to go from operating system awareness of the interrupt to eventual application processing, using the same selection technique:

There are many different choices in terms of what time ranges are of interest to measure. Here we’ve decided to measure from the time that the operating system is aware of the interrupt (event 13511) through to the point at which the user process has started to respond to the signal generated by the \texttt{io-net} interrupt handler. Since the interrupt handler communicates using a pulse (event 13515), then the earliest that the user code can respond is when the \texttt{MsgReceive()} kernel call exits (event 13519) with the received pulse. In this case, we can see that the end-to-end latency from OS awareness to the start of user processing (nonprivileged) is 46.304 microseconds:
Alternate measurements that could be of interest and that you can easily examine include:

- The time that it takes for the user process to be scheduled rather than the time for it to start processing. This would be signified by a transition of one of the receiving process’s (io-net) threads to a READY or RUNNING state (event 13516 for example). This time may be significantly different from the actual start of processing time in busy systems with execution taking place with mixed priorities.

- The time between the end of specific interrupt handler processing, and the awareness of the user process (either the scheduling or the start of processing) of the interrupt’s occurrence. This timing can be quite relevant when there are multiple interrupt-handling routines sharing the interrupt that may skew the time before the interrupt handler starts its processing of the interrupt.

**Locating Events of Interest**

Trace event log files contain a wealth of information, but unfortunately that information is often buried deep in among thousands, if not millions, of other events. The System Profiler tooling helps provide tools to reduce and remove some of this “noise” to help you focus on the areas of a log that are important to you.

**Requirements**

There are no specific requirements for this use case, but some of the topics may not apply, depending on the types of events that have been captured.
System Profiler use cases

Procedure

We’ll walk through some of the tools available to help you to reduce and filter the data contained in a trace event file. Where this information is most useful is during investigations involving the Timeline editor pane. The timeline displays information with a very fine granularity and is often the display that users turn to in order to “single step” through the execution flow of an activity of interest. To open the Timeline editor pane, select System Profiler → Display → Switch Pane → Timeline.

Timeline editor pane filters

The first level of data reduction is to use the Filters view to remove information that isn’t significant for the tracing of the problem you’re interested in. The Filters view is synchronized with the active System Profiler editor; you can display it via the menu Window → Show View → Other… → Filters or by right-clicking Filters… in the Timeline editor pane.

This view provides you with the following types of filtering:

- The Owners tab shows a list of event owners/sources, letting you select or unselect event owners to be displayed. Unselecting an event owner in the list removes that owner from the Timeline editor pane.
System Profiler use cases

- The Events tab is similar to the Owners tab, but it provides filtering capabilities for individual trace events rather than for the owners of those events.
The Saved Filters tab provides a listing of preconfigured filters that are available. These filters are often based on more sophisticated criteria for determining if events or event owners are to be displayed. The CPU Usage filter is an example of such a filter. It removes from the display any event owner that hasn’t consumed any CPU resources within the current display area. By using this filter in conjunction with zooming and searching capabilities, you can quickly reduce the overall data set.

**Trace event log filter synchronization**

By default, the Trace Event Log view presents a display that uses the same filters as the currently active editor. However, there are times when it’s useful to be able to temporarily “unfilter” the Trace Event
System Profiler use cases

Log view display to see the raw content of the log file. You can accomplish this by toggling the editor’s Synchronize button on the Trace Event Log view display:

There are times when you’re looking at an event stream and want to quickly navigate through it. One mechanism for doing this is to move to the next or previous event, using the toolbar commands (Next, Previous, Next Event In Selection, Previous Event In Selection).

Another, more flexible, alternative is to use the Find functionality of the Timeline editor pane. Selecting Edit→Find/Replace opens a dialog similar to the one found in many text editors:

The dialog supports searching a restricted set of event owners (based on the selection made in the Timeline editor pane) as well as searching forwards and backwards through the log file. This is convenient when you know specifically what type of event you’re looking for in a sequence of events (e.g. the next RUNNING state for a thread).

The Find dialog moves the selection marker in the Timeline editor pane to the appropriate event.

Trace Search

If you need to generate a collection of events matching a particular condition, or you need to construct a more complicated expression (perhaps including event data) in order to find the events you’re looking for, you need the power of trace event conditions and the Trace Search tool.

The search tool is invoked via the menu item Search→Search. Opening this up presents a dialog similar to the following:
There are many different search mechanisms in Eclipse, but the Trace Search is the one that we’re interested in as a “super find” for the event files.

Searching is based on trace conditions. Trace conditions describe a selection criterion for matching an event and can be based on anything that an event provides (ownership, data payload, and so on).

To add a condition that will locate all of the `MsgSend()` calls that may have been made for write system calls:

1. Add a new condition via the Add button in the search dialog. This brings up a new condition dialog that you can fill in with the `MsgSendv()` kernel call and the `write()` function entry to match. When matching string values (such as function names), the matching is done based on a regular-expression match.
Once you’ve defined the condition, it shows up in the Defined Conditions table shown in the Trace Search panel. You can combine individual conditions to form Boolean expressions if required.
3 Specifying the newly created condition in the Search Expression drop-down and selecting Search automatically opens up the Search Results view. If the Timeline editor pane is open, double-clicking on a search result (assuming that the result isn’t filtered) moves the timeline selection directly to that event:

Search results are also marked in the timeline to help show the event distribution over the period of the log file:
Exporting filtered log files with Save As

Often the kernel event files that are captured are large and contain a significant amount of nonessential data for the problem at hand. Of course, this is generally only determined after the fact, once you’ve performed some basic analysis.

You can use the File→Save As menu command to create a new log file that’s based on the current log file in the System Profiler editor.

You can restrict the new log file to just the selected area (if you’ve made a selection), and you can also use the current filter settings (event and event owner) to reduce the amount of additional data that’s stored in the log file.

The new log file contains the same attribute information as the original log file (including the system version, system boot time, number of CPUs, and so on). Any event owners, such as interrupts, processes, and threads, which are referenced by events in the new log file, are synthetically created with timestamps matching the start time(s) of the new log file.

Associated views

The QNX System Profiler perspective includes the following views:

- System Profiler editor
- Bookmarks view
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:
The System Profiler editor panes include the following:

Summary pane (the default)

Displays a summary of the activity in the system, accounting for how much time is spent processing interrupts, running system- or kernel-level code, running user code, or being idle.

The IDE generates an overview of the CPU and trace event activity over the period of the log file. This overview contains the same information displayed in the Overview view.

The process activity (amount of time spent RUNNING or READY, number of kernel calls) displayed in the Summary pane contains the same information as can be extracted by drilling down...
for a particular time range of the event log using the General Statistics and Event Owner Statistics views.

CPU Activity pane
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 pane
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.

Timeline pane 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 Summary pane is the default. To choose one of the other types, right-click in the editor, then select Display→Switch Pane, or click this icon:

You can choose a specific display type by from the dropdown menu associated with this menu item or icon.

For the CPU Activity pane, 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 Chart
- Bar Chart
- Histogram
System Profiler editor

- **Area Chart**

3D versions of the charts are also available.

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 Display**, or click this icon:

![Split Display icon]

You can lock two panes to each other. From the **Split Display** submenu, choose the graph you want to display in the new pane, or click this icon:

![Lock panes icon]

You can have a maximum of four panes displayed at once.

A number of different features are available from within the editor:

**Event owner selection**

If you click on event owners, they’re selected in the editor. These selected event owners can then be used by other components of the IDE (such as **Find**).

If an owner has children (e.g. a parent process with threads), you’ll see a plus sign beside the parent’s name. To see a parent’s children, click the plus sign.

**Filters**

You can use the **Filters** view to filter out event owners and specific events. This lets you significantly cut down on the unwanted event “noise” in the
System Profiler editor

You can then save a new version of the 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, choose **Window → Show view → Other → QNX System Profiler → Filters** to open the Filters view. Select the Events tab, and then replace **type filter text** with **MsgSend**. Click **Deselect All**, and then click the applicable **MsgSend** events that your search found. The selected events are added to your display dynamically.

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. Note the **Class** and **Code** fields in the dialog are filled in with the values from the currently selected event in the Timeline pane.

Bookmarks

You can place bookmarks in the timeline editor just as you would to annotate text files. To add a bookmark, click the Bookmark icon in the toolbar ( ), 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 General Statistics views. These views can synchronize with the cursor, event owner selections, and time ranges, and can adjust their content accordingly.
System Profiler editor

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 clicking this button in the toolbar:

By default, this displays the IPC trace arrows for all event owners. You can choose to display only the arrows for the selected owners by choosing Selection from the button’s dropdown menu.

Display Event Labels

The Display Event Labels button in the toolbar lets you display labels in the timeline; open the button’s dropdown menu and select the type of labels you want to display:

- Priority Labels — display the thread’s priority
- State Labels — display the thread’s state as a label
- State Icons — display the thread’s state as an icon above the thread
- IPC Labels — add text boxes to the IPC lines to indicate which thread or process you’re communicating with
- Event Labels — display labels for kernel events, including I/O and memory events.

If you haven’t expanded a process in the display, the labels for all its threads are displayed. By default, no labels are displayed.
Types of selection

Within the editor, you can select either of the following:

- an owner (e.g. a process or thread)
- a point in time

Owners

To select a single owner, simply click the owner’s name. To unselect an owner, press and hold the Ctrl key, then click each selected owner’s name.

To select multiple owners, press and hold the Ctrl key, then click each owner’s name.

Time

To select a point in time, click an event on the timeline.

To select a range, click the start point on the timeline, then drag and release at the end point.
System Profiler editor

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 (**Zoom Level** → **Custom**)
- toolbar icons
- hotkeys (+ to zoom in; - to zoom out)

**Scrolling**

You can 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>Ctrl←</td>
</tr>
<tr>
<td>The selection to the right by one event</td>
<td>Ctrl→</td>
</tr>
<tr>
<td>The display to the left</td>
<td>←</td>
</tr>
<tr>
<td>The display to the right</td>
<td>→</td>
</tr>
<tr>
<td>The selection for the current owner to the left by one event</td>
<td>Ctrl-Shift←</td>
</tr>
<tr>
<td>The selection for the current owner to the right by one event</td>
<td>Ctrl-Shift→</td>
</tr>
</tbody>
</table>

You can use these keys to scroll through the owners:
To move the display: | Use this key:
---|---
Up by one owner | ↑
Down by one owner | ↓
Up by one page (horizontal scrollbar thumb size) | Page Up
Down by one page (horizontal scrollbar thumb size) | Page Down
To the top of the owner list | Home
To the bottom of the owner list | End

Hovering

When you pause your mouse pointer over an owner or an event, you’ll see relevant information (e.g. PID, timestamps, etc.).

Bookmarks view

Just as you can bookmark lines in a text file, in the System Profiler editor, you can bookmark particular locations and event ranges displayed, and then see your bookmarked events in the Bookmarks view.

Client/Server CPU Statistics view

The Client/Server CPU Statistics view (Window→Show View→Other…→QNX System Profiler→Client/Server CPU Statistics) tracks the amount of client/server time spent in the RUNNING state. In a message-passing system, it may be that a particular thread is consuming a large amount of CPU time, but that CPU time is being consumed based on requests from one or more clients. In these cases, in order to achieve a higher performance, the client requests on the server must be reduced (assuming that the server is identified as a bottleneck).

This panel provides a tabular display of threads that spend time in a RUNNING state (slightly different from pure CPU usage) and breaks
down the display such that for each thread there is a summary of how much time it spent directly in the RUNNING state and how much RUNNING time it “imposed” on other threads in the system and the total of those two times.

The System Profiler’s Client/Server CPU Statistics view.

You can expand the table, via the View menu, to show how much time the client imposed on various server threads. The imposed time is cumulative: if client A sends to server B, then until B replies to A, any time that B consumes is seen as “imposed on” time. If during that time B sends to server C, then server C is also billed time as “imposed on” by A. The rationale here is that B would not have engaged with server C were it not for the initial message from A.

By sorting by imposed time, it is possible to identify which clients are predominantly driving the system and which servers may be bottleneck points.

Condition Statistics view

This view provides a tabular statistical representation of particular events. The statistics can be gathered for the entire log file or for a selected range.

When you first open the Condition Statistics view, it contains no data:
You must configure conditions and the table to view condition statistics.

**Configuring conditions**

To configure conditions for the Condition Statistics view:

1. Click the **Configure Conditions...** button, or choose **Configure Conditions...** from the view’s drop-down menu.

The IDE displays the Modify Conditions dialog.
2 Click the Add button to bring up the Trace Condition Wizard. The IDE displays the Trace Condition Wizard dialog:
3 Give your condition a unique name and select the appropriate class and code. For example, select **Process and Thread** from the Class dropdown menu, then select **Mutex** under the Code dropdown menu.

4 Click **Finish**.

5 Click **OK** in the Modify Conditions dialog.

6 Click the Configure Table Condition Contents (1) button, or choose **Configure Table...** from the view’s dropdown menu. The IDE displays the Condition Selection dialog:
Condition Statistics view

7 Add a check mark beside the conditions that you want displayed in the table.

8 Press OK to confirm your selections.

You’ll need to click the Refresh button ( ) to populate this view with data.
CPU Migration pane

The CPU Migration pane provides a display that draws attention to some of the potential performance problems associated with multiple-CPU systems. This is principally concerned with CPU migration activities.

There are two migration details that are currently charted over the period of the log file:

- The first chart provides a display showing the number of CPU-scheduling migrations over time. The count is incremented each time a thread switches CPUs. The peaks in this chart indicate areas where there’s a high level of contention for particular CPUs. This type of cross-CPU migration can reduce performance because the instruction code cache is flushed, invalidated and then reloaded on the new CPU.

- The second chart provides a display showing the count of cross-CPU communication, where the sending client thread and
the receiving server thread are running on different CPUs. This type of cross-CPU communication on the initial message-sends is a potential performance problem since the data that is associated with the message-pass can’t be maintained in the processor data cache, and the caches must be invalidated, as the data transfer is moved to the new CPU.

This pane contains valid data only when the log file contains events from a system where there are multiple CPUs.

**Event Owner Statistics view**

This view provides 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 (Refresh) to populate this view with data.
General Statistics view

This view provides 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 this view with data.

Raw Event Data pane

The Raw Event Data view (Window→Show View→Other…→QNX System Profiler→Raw Event Data) lets you examine the data payload of a particular selected event. It shows a table of event data keys and their values. For example if an event is selected in the Trace Log view, rather than attempting to look at all of the data in the single line entry, you can open the Raw Event Data view to display the data details more effectively.
Overview view

The System Profiler’s Raw Event Data view.

Overview view

The Overview view (Window → Show View → Other… → QNX System Profiler → Overview) shows two charts spanning the entire log file range.

The System Profiler’s Overview view.
These charts display the CPU usage (per CPU) over time and the volume of events over time. The Overview reflects the current state of the active editor and active editor pane. You can select an area of interest in either one of the charts; then, using the right-click menu, zoom in to display only that range of events to quickly isolate areas of interest due to abnormal system activity.

**Partition Summary pane**

The Partition Summary pane provides a summary of the entire log file, focused on QNX’s adaptive partitioning technology. For each distinct configuration of partitions detected in the log file, the distribution of CPU usage used by those partitions is displayed, along with a tabular display showing the distribution of CPU usage per event owner per partition.

You can use this information in conjunction with the CPU Usage editor pane to drill down into areas of interest. This pane contains valid data only when the log file contains partition information, and the process and thread states are logged in wide mode (so the partition thread scheduling information is collected).
Thread State Snapshot pane

You can also get snapshots of the usage of the adaptive partitioning on your system through the System Information perspective’s Partition Summary view. For more information, see the “Getting System Information” chapter.

Thread State Snapshot pane

In addition to asking why a particular process’s thread may be running, developers are often faced with the task of understanding what the rest of the system is doing at a particular point in time. This question can easily be answered using the Thread State Snapshot view (Window→Show View→Other…→QNX System Profiler→Thread State Snapshot).

Like the Why Running? view, this view is keyed off of the current cursor position in the System Profiler editor Timeline pane. For a given time/position, it determines the state of all of the threads in the system and presents that list to you in such a fashion that you can then determine if this “system state” is what you’d anticipated.
Timeline State Colors pane

Note that you must click the refresh icon in the Thread State View’s toolbar to update the contents of the Thread State View when you select a point in the Timeline.

Timeline State Colors pane

You can use the Timeline State Colors view (Window→Show View→Other…→QNX System Profiler→Timeline State Colors) if you’re unfamiliar with the System Profiler timeline editor pane state colorings, or if you’d like to change the color settings to something more appropriate for your task.

The view displays a table with all of the color and height markers that are used when drawing the timeline display. These settings can be bulk imported and exported using the view’s dropdown menu based on particular task requirements. The default settings generally categorize states with similar activities together (synchronization, waiting, scheduling, etc.).
Trace Event Log view

The System Profiler’s Timeline State Colors view.

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.
Why Running? pane

The Why Running? view (Window → Show View → Other… → QNX System Profiler → Why Running?) works in conjunction with the System Profiler timeline editor pane to provide developers with a single click answer to the question “Why is this thread running?” where “this thread” is the actively executing thread at the current cursor position.

By repeating this action (or generating the entire running backtrace) developers can get a clearer view of the sequence of activities leading up to their original execution position. Not to be confused with an execution backtrace, this “running backtrace” highlights the cause/effect relationship leading up to the initial execution position.

The System Profiler’s Why Running? view.
Chapter 13

Common Wizards Reference

In this chapter...

Introduction 453
Creating a C/C++ project 456
Creating a target 472
Converting projects 474
Importing projects 487
This chapter describes the IDE's wizards.

Introduction

Wizards guide you through a sequence of tasks, such as creating a new project or converting an existing non-QNX 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:
Notice the overlap: the QNX C Project wizard appears in both C and QNX.
In the C/C++ Development perspective, you can also access the QNX C/C++ Projects wizards via the New C/C++ Project button:

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*.

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 folders and files:

   In the Navigator view, right-click the navigator pane and select **Refresh**.
Creating a C/C++ project

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
QNX C++ Project

A C or C++ project for multiple target platforms. It supports the QNX-specific project structure using common.mk files to perform a QNX recursive make. A QNX Project can automatically build either one executable or one library object (in different formats). You can switch between application or library nature by using the project properties.

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 chapter in the Neutrino Programmer’s Guide.)

Managed Make C Project
Managed Make C++ Project

A managed make project generates the Makefile for you automatically. In addition, the module.dep and module.mk files are created for every project subdirectory. These files are required for your managed make projects to build successfully.

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>Managed Make C Project</td>
<td>C</td>
</tr>
<tr>
<td>QNX C Project</td>
<td>C or QNX</td>
</tr>
<tr>
<td>Standard Make C++ Project</td>
<td>C++</td>
</tr>
<tr>
<td>Managed Make C++ Project</td>
<td>C++</td>
</tr>
<tr>
<td>QNX C++ 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:

```
| ! $ ( ) & ‘ : ; \ * ? [ ] # ^ = % < { } |
```

because they’ll cause problems later.

6. If you don’t want to use the default location for the project, specify a different one.

7. Select the type (application or one of library types):
Creating a C/C++ project

If you’re building a library, see below.

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 project

You’ll need to choose the type of library you wish to build:
Creating a C/C++ project

Static library (libxx.a)

Combine object files (i.e. *.o) into an archive (*.a) that is directly linked into an executable.

Shared library (libxx.so)

Combine object files 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 usually is a symbolic link to the latest version.

Static library for shared objects (libxxS.a)

Same as a 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.
Creating a C/C++ project

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`.

Here’s a sample `Makefile` you can use:

```makefile
CC:=qcc

hello: hello.c

all: hello

clean:
    rm -f hello.o hello
```
Creating a C/C++ project

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++ Project
Tabs:
- Options
- Build Variants
- Make Builder
- Error Parsers
- Projects

Standard or Managed Make C or C++ project
Tabs:
- Projects
- Make Builder
- Environment
- Error Parsers
Creating a C/C++ project

- Binary Parser
- Discovery Options
- C/C++ Indexer

**Build Variants tab**

The Build Variants tab lets you choose the platforms to compile executables for:

By default, *none* of the platforms are enabled. You might want to change your default preferences for all new QNX projects. To do this, open **Window → Preferences → QNX → New Project → Build Variants**.

Select the specific architecture(s) and build variant(s) you want to build your project for.

You can click the **Select All** button to enable all of the listed variants, or the **Deselect All** button to disable all of the listed variants.
Creating a C/C++ project

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 one build variant for the Indexer to use:

1. Select the build variant you want the Indexer to use.

2. Click the **Set Indexer Variant** button.
   The variant’s name changes to include “> This variant’s symbols and include paths will be used for source indexing.”

**Projects tab**

In the Referenced Projects list, you can set project dependencies for the new project. In the list of other projects in the Workbench, you can select one or more projects you want the new project to depend on. Initially, no projects will be selected.
Creating a C/C++ project

For example, if you associate myProject with mySubProject, the IDE builds mySubProject first, followed by your project. 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 first build 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`).
Creating a C/C++ project

Workbench Build Behavior

You can specify how you want the IDE to build your project. For example, you can:

- check **Build on resource save (Auto Build)** to enable automatic building
- change the name of the auto build target (the default is `all`)
- change the name of the incremental build target (the default is `all`)
- change the name of the clean target (the default is `clean`)

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

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.

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).
Creating a C/C++ project

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) to use to deal with the project’s binary objects.

For more information about indexers, see the “Choosing an indexer” chapter.
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.

At a later time, you can supply this data using the **Search Paths** item in the project properties.
Creating a C/C++ project

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.

Environment tab

If you’re building a Standard Make C/C++ project, then this tab lets you control environment variables used by the build.
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 main menu, select File→New→Project.…
2. Expand QNX category.
3. Select QNX Target System Project.
4. Click Next. The New QNX Target System Project wizard appears:

Creating a target

Chapter 13 • Common Wizards Reference
Creating a target

5 Complete the fields described below:

Target Name Type a descriptive name for your QNX Target System Project.

Hostname or IP

Enter the hostname or IP address for the target system that’s running qconn.

Port

Enter the port number for qconn. Leave this as the default (8000), if you’re running qconn with the default settings.

6 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.
Converting projects

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 \texttt{.project} and \texttt{.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 \texttt{Makefile} hierarchy (a project with a QNX nature causes the IDE to use the QNX \texttt{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.

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 into 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.
Converting projects

To open the Properties dialog for 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:

3. In the left pane, select **QNIX C/C++ Project**.

4. 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
Converting projects

Makefiles and Directories chapter 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 `usmsg` 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 these categories you select:

- General options
- Extra source paths
- Extra include paths
Converting projects

Compiler type

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
Output options  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.

Code generation  For the Optimization level, you can specify four levels: from 0 (no optimization) to 3 (most optimization). 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 pass 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.
Converting projects

- **QNX target** — You can add source from anywhere in or below the `$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 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 these categories you select:

- General options
- Extra library paths
- Extra libraries
- Extra object files
- 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 a 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** — 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 in the same way 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:

- the stripped name, the base name without the **lib** prefix (which *ld* adds automatically), and without the suffix (.so or .a).
- the library type (static or dynamic)
- debug or release mode. A “No” or “Yes” in this field indicates whether or not the builder matches the debug or release version of the
Converting projects

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 want to use a release version of your binary and link against a debug version of the library, 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 a relocatable notation so other members with different workspace locations can all work successfully without having to make any project adjustments.

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.
Converting projects

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.
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 open the Import wizard:
Importing projects

Choose File→Import..., or right-click in the Navigator or C/C++ Projects view, and then choose Import....

The Import wizard.

The Import wizard can import resources from several different sources:

- Existing Container Project into a Workspace
- Existing Project into a Workspace
- External Features
- External Plug-ins and Fragments
Importing projects

- File System
- GCC Coverage Data from a Project
- QNX Board Support Package
- QNX `mkifs` Buildfile
- QNX Source Package
- Team Project Set
- Archive file

**Importing an existing container project into a 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 projects

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 markers, it is possible that wizard just cannot find their location.

3 By default, every project referenced by the container project is also 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.

Importing an existing project into a 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 Projects panel.
Importing projects

2 Enter the full path of an existing project directory in the Select root directory field, or the path to an archive in the Select archive file field, or click the appropriate Browse... button to select a project directory or archive using the file selector.

3 Check the projects you want to import.

4 If you wish, click Copy projects into workspace to leave the original project unchanged.

5 Click the Finish button to import the selected project into your workspace.
Importing external features
Eclipse developers use this for developing IDE plugins and features.

Importing external plugins and fragments
Eclipse developers use this for developing IDE plugins and features.

Importing a 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.
Importing projects

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.
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.

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.

---

**Importing GCC coverage data from a 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.dac` 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 then click the **Next** button.

   The IDE displays the **GCC Coverage Import** panel.
Importing projects

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

Referenced projects and comments.

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.

**Importing a 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.
Importing projects

The IDE displays the **Import QNX BSP** panel.

![Import QNX BSP panel](image)

**Importing a BSP:**

2 Select an installed BSP from the **Known Packages** list.

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.

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.
Importing projects

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.

5 Click Yes to build all of the BSP projects that were just imported. Click No to return to the IDE.

Importing a 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 a mkifs .build file.
Importing projects

2 Click the Browse... button beside Select project to import to select a destination for this import.

3 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.

4 Select one or more projects, and then click OK.
   The IDE imports the selected .build file’s System Builder configuration.

Importing a 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.
Importing projects

2 Select an installed source package from the Known Packages list.
   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.

Importing a 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:
Importing projects

1 In the Import wizard, choose Team Project Set and click the Next button.

The IDE displays the Import a Team Project Set panel.

![Import a Team Project Set](image)

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.

Importing an archive file

To copy files and folders from a ZIP archive into an existing project in your workspace:

1 In the Import wizard, choose Archive File and click the Next button.

The IDE displays the Archive File panel.
Importing projects

importing code from a zip archive.

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.
Importing projects

This project or folder must already exist before you open the Import wizard.

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 subfolders, 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? 511
Types of launch configurations 511
Running and debugging the first time 513
Running and debugging subsequent times 517
Setting execution options 518
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., the 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

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.
Running and debugging the first time

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:
Running and debugging the first time

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.

You can also use the Debug As menu item to conveniently select a particular launch configuration:

---

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:

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.)
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:

- 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 use this method, you must have selected Display in favorites when you first created your launch configuration. If you didn’t, edit the Display in favorites menu option under the Common tab. See “Setting execution options” in this chapter.

To debug or run a program using your favorites list:

1. Do one of the following:
Setting execution options

- 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:

![Launch configurations list]

2 Select your launch configuration.

Launching the last-launched program

To relaunch the last program you ran or debugged:

Press F11 or click the dropdown button (▼) beside the Debug or Run icon, then select your launch configuration.

Setting execution options

The Launch Configurations dialog has several tabs:

- Main
- Arguments
- Environment
Setting execution options

- Download
- Debugger
- 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:
Setting execution options

Different fields appear in the Main tab, depending on the type of configuration you're creating. Here are descriptions of all the fields:

**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 Project**.
Setting execution options

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` manager.

- 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.

For serial debugging, make sure that the pseudo-terminal communications manager (`devc-pty`) is running on your target.

Serial Port Options

Here you can specify the serial port (e.g. `COM1` for Windows hosts; `/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.
Setting execution options

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 `myProgram -v -L 7`, type `-v -L 7` in this field. You can put `-v` and `-L 7` 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.
Setting execution options

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_2 when you run your program, use this tab. Click New to add an environment variable.

![Environment Tab Image](image)

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.
Setting execution options

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.
Setting execution options

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.

The **Use unique name** option is set on by default. This means the IDE makes your executable’s filename unique (e.g. by appending 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.

---

526 Chapter 14 • Launch Configurations Reference
Setting execution options

Stop on startup at

This option is set on by default and the default location is `main()`(). 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:

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.

If you choose to track all the variables or registers, your program’s performance may suffer.

**Debugger Options**

**GDB command file**

This field lets you specify a file for running `gdb` using the `-command` option (see the **Utilities Reference**).
Setting execution options

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.

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.

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...
Setting execution options

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

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 file, you can save it in a location you specify (such as in your project). Saving as a shared file 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
Setting execution options

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 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):
Setting execution options

Application Profiler

Lets you count how many times functions are called, who called which functions, and so on. For more information about this tool, see the Profiling Your Application chapter.

![Application Profiler](image)

Memory Analysis

Lets you track memory errors. For more information about this tool, see the Finding Memory Errors chapter.

![Memory Analysis](image)
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 information about 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... 537
Tutorial 1: Creating a Standard Make C/C++ project 537
Tutorial 2: Creating a QNX C/C++ project 540
Tutorial 3: Importing an existing project into the IDE 541
Tutorial 4: Importing a QNX BSP into the IDE 543
Before you start...

Here are several tutorials to help you get going with the IDE.

**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:
Tutorial 1: Creating a Standard Make C/C++ project

1. To open the New Project wizard, select **File**→**New**→**Project**... from the main menu of the workbench.

2. Expand **C** (or **C++**) folder, and 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:

6. Name your file “**Makefile**” and click **Finish**. The editor should now open, ready for you to create your **Makefile**.
Tutorial 1: Creating a Standard Make C/C++ project

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.

7 When you’re finished editing, save your file (right-click, then select **Save**, or click the Save button in the tool bar).

8 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 chapter 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 appears.](image)

   The New Project wizard appears.

2. Name your project, then select the *type* (e.g. **Application**).

3. Click **Next**.

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).

5. 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.
Tutorial 3: Importing an existing project into 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 3: Importing an existing project into the IDE**

In this tutorial, you’ll use the IDE’s Import wizard, which lets you import existing projects, files, and even files from ZIP archives into your workspace.

You can use various methods to import source into the IDE. For details, see the Managing Source Code 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.
2 In the Import wizard, choose **Other → 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:
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 generate 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:
Tutorial 4: Importing a QNX BSP into the IDE

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.
Tutorial 4: Importing a QNX BSP into the IDE

- 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.

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 source 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>$HOME/ide4-workspace</td>
</tr>
<tr>
<td>.metadata folder (for personal settings)</td>
<td>$HOME/ide4-workspace/.metadata</td>
</tr>
<tr>
<td>Error log</td>
<td>$HOME/ide4-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
- **flashcmp** — Compress files for a Flash filesystem
- **gcc** — Compile and link a program
- **gdb** — Debugger
- **icc** — Intel C and C++ compiler (x86 only, purchased separately)
- **ld** — Linker
- **make** — Maintain, update, and regenerate groups of programs
- **mkefs** — Build an embedded filesystem
- **mkifs** — Build an OS image filesystem
- **mkimage** — Build a socket image from individual files
- **mkrec** — Convert a binary image into ROM format
- **objcopy** — Copy the contents of one object file to another
- **pdebug** — Process-level debugger
- **qcc** — Compile command
- **qconn** — Provide service support to remote IDE components
- **strip** — Remove unnecessary information from executable files
- **usemsg** — Change the usage message for a command

For more information, see the *Utilities Reference*. 
Appendix D

What’s New

In this appendix. . .

What’s new in IDE 4.0.1? 561
What was new in IDE 4.0? 563
What was new in the IDE in 6.3.0 SP3? 582
What was new in the IDE in 6.3.0 SP2? 582
What was new in the IDE in 6.3.0 SP1? 615
This chapter identifies the changes made to the QNX Momentics IDE.

What’s new in IDE 4.0.1?

IDE 4.0.1 includes the following new features:

- Support for Windows Vista
- IDE supports self-hosted Solaris and Neutrino OSs
- Eclipse updates
- General improvements
- Improvements to upgrading your software
- Download Center updates
- Documentation updates

Support for Windows Vista

Full support for the Windows Vista OS

QNX IDE workbench version 4.0.1 includes support for the Windows Vista operating system. For a complete list of supported host and targets, see the QNX IDE Momentics Release Notes.
What's new in IDE 4.0.1?

Windows Vista patch available for download to support the QNX win32 SDK
The QNX Download Center includes a patch to provide you with Windows Vista support for the QNX win32 SDK. This patch includes an updated version of the core file cygwin1.dll, uname.exe, and bash.exe.

IDE supports self-hosted Neutrino and Solaris
The IDE Workbench now supports self-hosted Neutrino and Solaris configurations.

Eclipse updates
In IDE 4.0.1, you can upgrade to use Eclipse platform version 3.2.2, and the Eclipse CDT to version 3.1.2. For detailed information about the changes made to the Eclipse Platform and CDT, see the Eclipse.org website.

General improvements
IDE 4.0.1 includes various overall improvements to the application. For a detailed list of these improvements, see the release notes.

Improvements to upgrading your software
In addition to updating your IDE features and qconn, you can now use the Software Updates manager to quickly update the IDE software.

If you have an existing installation of IDE 4.0 on your machine, you can use the IDE Software Updates manager to update the IDE to version 4.0.1 (see the topic “Upgrading using the IDE Software Update manager” in the installation notes for Windows and Linux hosts; for Solaris and Neutrino, you will need to follow the typical installation instructions for a new installation.
What was new in IDE 4.0?

Download Center updates
For this release of the IDE, the Download center has an update to qconn and to the malloc library. For instructions about using the Software Update Manager, see the installation notes for Windows or Linux, or see the QNX Momentics IDE Online Help.

Documentation updates
The documentation includes various improvements and enhancements. For a list of some of these improvements, see the items under the heading, “List of fixes” in the release notes. In addition, a printed version of the QNX Momentics IDE User’s Guide is also available.

What was new in IDE 4.0?
IDE 4.0 included the following new features:

- Application Profiler perspective
- System Profiler perspective
- Memory Analysis perspective
- System Information perspective
- System Builder perspective
- Managed Make for QNX projects
- C/C++ development
- Momentics workbench
- Debugging

Application Profiler perspective
Full profiling support for shared libraries and DLLs (requires the libc patch)

Call information is included by instrumented shared libraries
What was new in IDE 4.0?

and DLLs. Callout-type calls from noninstrumented to instrumented code are now shown.

Background symbol loading

Symbol loading is now performed in a background thread, resulting in enhanced user experiences, because you no longer have to wait for the profiler to initialize itself loading symbols during the launch startup process.

System Profiler perspective

Configuration settings

You can now save and load (as well as import and export) System Profile configuration settings. System Profiler sessions are now controlled through the standard Launch Configuration mechanism.
What was new in IDE 4.0?

Adaptive Partition Summary view
Show adaptive-partition usage and summary information, including:

- reporting the bankruptcy/critical information in the summary
- displaying the critical areas in the timeline

CPU usage
The CPU Usage pane displays a thread’s partition and priority and is now a tabbed view that presents the display of CPU usage sliced by CPU, partition, and priority. You can easily sort and export this information. In addition, the functionality of the Process Activity editor pane and the CPU Usage editor pane have been merged together.

Timeline navigation and usability improvements
The System Profiler Timeline display provides insight into the execution of the system, broken down by individual process, thread, and interrupt activity.
What was new in IDE 4.0?

Improving the ability to navigate through the system events lets you quickly zoom into problem areas or follow the flow of activity, matching what you see visually in the timeline with what you expect to occur in your systems.

Specific improvements include:

- Direct event and event owner position highlighting. Simply double-click on an event in the event log display, and the timeline immediately jumps to that event.
- Event navigation by selected event owners. Navigate along an execution timeline for a particular thread, forward and backward, quickly and easily with a click of a button.
- The Jump To Event dialog lets you collaborate more easily with your colleagues by providing direct event navigation by event index, event cycle, or natural time.
- The Timeline State Legend lets you quickly see and adjust the color-coding of the various system states in the timeline.

Enhanced data searching, filtering and reduction capabilities

The instrumented kernel log files can easily generate tens, if not hundreds, of megabytes of information in a short period of time. This amount of information provides a good context for examining problems, but is often more data than is required to highlight a problem to another developer.

This version of the System Profiler lets you save and create new log files that contain only a relevant subset of the data required for other developers to understand problems. New log files can be created focusing on only a particular range of time, and filtering out unnecessary events or event owners, such as server processes.

A new event data extraction and interpretation framework has been introduced with this release that allows event data conversions, particularly of User Events, to be more powerfully applied. Users can now describe their own events using C-style data structures. Accompanying this new framework is a new Event Data Properties view that more clearly displays, in a
What was new in IDE 4.0?

In a tabular manner, the key/value data associated with individual events.

New data analysis tools

This release of the System Profiler provides you with some additional analysis tools to help you understand the execution status of their system and to give you more insight into why certain sequences of events are occurring.

- **Why Running view** — One of the most difficult tasks in a highly modularized software system is tracing the flow of execution as it flows from one component to another. The Why Running view works in conjunction with the System Profiler editor’s Timeline pane to provide them with a single-click answer to the question “Why is this thread running?” for whatever thread is actively executing at the current cursor position.

  By repeating this action (or generating the entire running backtrace), you receive a clearer view about the sequence of activities leading up to their original execution position. Not to be confused with an execution backtrace, this “running backtrace” highlights the cause and effect relationship leading up to the initial execution position.

- **Thread State Snapshot view** — In addition to asking why a particular process’s thread may be running, developers are often faced with the task of understanding what the rest of the system is doing at a particular point in time. This question can easily be answered using the new Thread State Snapshot view. Like the Why Running view, this view is based on the current cursor position in the System Profiler editor’s Timeline window. For a given time and position, it determines the state of all threads in the system, and it presents the information so that you can better determine if this system state is what you had originally anticipated.

- **Client/Server CPU Usage Statistics** — An additional statistics panel has been added into the System Profiler to compliment the graphical Usage and Active System Profiler...
What was new in IDE 4.0?

editor panes. This statistics table generates a map of both direct and imposed CPU usage (based on the thread RUNNING state).

In a message-passing system, such as Neutrino, where servers are often working on behalf of clients, determining the true execution cost can be challenging. This new view helps with that endeavor. For each executing thread, this view contains the direct CPU time the thread executed, and if that thread communicated to another server, the amount of time that the server executed on its behalf is accumulated as imposed CPU usage to the client. You can then easily export this data can then be easily exported for use with other analysis and charting tools.

General usability improvements

There have been a large number of general usability and performance improvements made to the System Profiling tools. Some of the more interesting improvements include:

- Cut and paste support in tables (Trace Event Log, Why Running, Search, Statistics views) allowing you to more easily exchange small snippets of information with other developers by simply cutting and pasting the information from the System Profiler tools.
- There have been some significant changes to the Trace Event Search user interfaces to align with the new QNX Momentics workbench search frameworks. Search results can now be overlaid, one on top of the other, without having to generate compound search conditions. The results display in a more tabular format and can be grouped into clusters in addition, you can use the system clipboard to export the results.

Memory Analysis perspective

This perspective includes a completely new Memory Analysis Tooling that’s reliable, fast, and robust, and with greatly improved performance and stability.
What was new in IDE 4.0?

Its features include:

- multiple views, most of them including both alphanumeric and graphical representations
- 2D and 3D bar and differentiator charts that graphically present the memory events and traces, allowing you to use multiple ways to sort and filter out the collected information, allowing a generation of user statistics and allocation statistics over a time base
- displays of Bands, Bins, and Usage of the memory that let you replay the collected data
- a Trace Details view that lets you replay the selected bins, bands, or usage, synchronized with the allocations
- the ability to import data generated by the debug malloc library
What was new in IDE 4.0?

- maintaining thread/process, shared library ownership and ordering/timing information of generated events if present
- detecting memory leaks — snapshot in time, detecting specific string and memory functions errors
- support for Standard Make C Project memory events and C++ memory events, including backtrace information and traceability of events back to the source code if present

You can now:

- change the Memory Analysis settings for a running process
- import debug malloc events from logged trace events to a file on the target system
- perform memory analysis on a running program by launching with debug malloc or attaching to a running process
- reconnect to existing Memory Analysis sessions

**System Information perspective**

Adaptive partitioning scheduler view

A new APS view lets you view information about adaptive partitions on a running target system.
What was new in IDE 4.0?

CPU affinity and process priority/scheduling controls
Threads and processes now have context menus that let you set the CPU affinity and change the process priority and scheduling algorithm. You can also change a thread’s name on targets running QNX Neutrino 6.3.2 or later.

Thread name support
Thread names are now shown in System Information views, if available.

Highlighting data changes
Changing data is now highlighted in System Information views, and a delta is displayed for CPU/data usage.
What was new in IDE 4.0?

System Builder perspective

TFTP Server improvements

The TFTP Server view includes several improvements:

- Network selection capability
- Improved user file search
- Search options and preferences available quickly from within the view

Managed Make for QNX projects

You can now create and use QNX Managed Make projects to handle building code dynamically for any and all QNX targets.
What was new in IDE 4.0?

These are similar to QNX Projects except they provide a richer environment for setting compiler and linker options. They also let you use custom build steps and set environment variables and build macros for `make`. Managed Make projects can be created in the New Project Wizard under C or C++. Select QNX Neutrino Executable, Shared Library, or Static Library Project Type, and then select which Configurations to build for. There are configurations for each processor type, compiler version, and debug/release build options.

**C/C++ development**

- C/C++ general improvements
- C/C++ user interface
- Indexers
- Debugger

**C/C++ general improvements**

DOM-based language backend

The C/C++ perspective is now served by a much more powerful language model in the form of a document object model (DOM) that helps make everything from the C/C++ outliner, to searching, to refactoring, to class browsing faster and more accessible to plugin developers.

Kernighan & Ritchie (K&R) C language support

In addition to the new DOM serving the backend source code model, the native parser is also much more tolerant of K&R C language dialects.

Faster searches

You will also notice significant improvements to the performance and speed of the C/C++ parser and the efficiency of the indexer. As a result, searches (both general and specific, such as searches for references and declarations) are faster and more accurate than in previous releases.
What was new in IDE 4.0?

C/C++ user interface

Build environment control

You can now fully configure the environment variables set when starting external build commands, through the Standard Make and Managed Build C/C++ project properties, or directly when creating new projects from the New C/C++ Project wizard.

PathEntry variables

The PathEntry project properties are important for the correct operation of the parser. The new **Window→Preferences→C/C++→PathEntry Variables** configuration lets developers configure a common set of variables that allow the sharing of projects more easily between different developer configurations. PathEntry variables provide you with a capability to open files and to load objects that are in one’s CLASSPATH.
Binary parser options

The binary parsers are more generic with the capability to redefine what external commands they use for data interpretations. Configure the settings in the project properties, or at New Project creation time.
What was new in IDE 4.0?

New discovery capabilities

Further work has been done to facilitate the discovery of compiler and project settings, including the ability to read from a build output file.
What was new in IDE 4.0?

Indexer selection

The C/C++ source indexers are now plugable components, each providing different feature support, letting you select from:

- No indexing  No support for searches or cross referencing.
- C/C++ parser-based indexing
  
  DOM model based, requires proper project configuration (defines and includes). Full searching, cross referencing and refactoring are supported.
What was new in IDE 4.0?

Drag-and-drop support

The C/C++ Project and the C/C++ Outline views have improved support for dragging and dropping of code elements and binaries.

Make Target filters

You can reduce the clutter in the Make Targets view by enabling the Hide Empty Folders filter.

Jump to definition

Complementing the already existing Open Declaration, also known as F3, is the powerful Open Definition (Ctrl-F3) that takes you directly to the definition of the class or type you’re looking for.
What was new in IDE 4.0?

New editing commands

Do more with less effort; additional C/C++ Editor key bindings are available for more flexible annotation and marker navigation.

C/C++ indexers

The C/C++ indexers have been substantially improved for greater performance and scalability.

The IDE lets you select the indexers to use to gather symbol information used by code search features. You can choose to use indexers in the project properties for each project under the C/C++ Indexer heading. It is also available in the Preferences dialog, which sets the default indexer for new projects and can also be used to apply that indexer selection to all projects in the workspace. There are three choices for indexing: no indexer, fast indexer, and full indexer. No indexer implies that no indexing takes place, and as a result, the search features will not function. The full indexer does a full parse of the workspace and provides the most accurate information, and it is similar to what was provided in version 6.3.0 SP2. The fast indexer uses heuristics to optimize the parse and is the recommended selection for large projects.

We recommend that you use the fast indexer option. The fast indexer takes a fraction (approximately 10-25%) of the time to index a large project than in IDE version 6.3.0 SP2. It also successfully indexes extremely large projects.

For example, the fast indexer can index the Firefox browser source in approximately 14 minutes. There are approximately 5600 indexable files (source and header files) with over 300,000 symbols.

Debugger

A new Modules view

All of the binary components, or modules, are now displayed in the debugger’s Modules view. This includes DLLs, shared objects and application binaries themselves. Additionally, each module
What was new in IDE 4.0?

can be expanded, letting you put breakpoints
directly on binary components without opening the
C/C++ Project view.

Memory view

The debugger now uses the standard platform
Memory view, allowing for a number of format
translations and multiple memory location
inspection.

Register groups

The Registers view now supports the concept of
register groups, so you can define the collection of
registers relevant to your particular debugging
environment.
What was new in IDE 4.0?

QNX Momentics workbench

Significant performance enhancements

For detailed information about the performance enhancements throughout the workbench for Eclipse, see the “What’s New” topic in the online help for the Workbench Users Guide.

Debugging

Console EOF In the Console view, you can signal end-of-file to a program waiting for input; for Windows, press Ctrl-Z, and for Linux, Neutrino, and Solaris, press Ctrl-D.

Console encoding

You can configure the console to display output using a character encoding different from the default encoding. To set the console encoding for an application, use the Console Encoding settings on the Common tab of a launch configuration.

Capturing program output

You can now configure a program’s output in a file in addition to writing it to the console. The behavior is controlled by settings found on the Common tab of launch configurations.
What was new in the IDE in 6.3.0 SP3?

Multiple console views
If you need to view multiple consoles at the same time, you can open additional Console views using the New Console View command located on the Open Console drop-down menu in the Console view.

What was new in the IDE in 6.3.0 SP3?
Service Pack 3 includes all of the bug fixes and updates found in the previous service packs. The development environment is the same as the one introduced with Service Pack 2.

What was new in the IDE in 6.3.0 SP2?
The following sections describe some of the more interesting or significant changes made to the IDE in QNX Momentics 6.3.0 SP2:

- 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
What was new in the IDE in 6.3.0 SP2?

- Background workspace auto-refresh
- Regular expressions in Find/Replace dialog
- 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:
  - You can dock it on the top right (default), top left, or left.
  - Perspective buttons include text for quickly identifying the current perspective.
- The Fast View bar can be on the bottom (default), left, or right.
- The size of the Fast View bar is reduced when there are no fast views.
- Title bars and tabs were merged to conserve space.
- Drag-and-drop has been improved.
- 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.
What was new in the IDE in 6.3.0 SP2?

- Different tab folder styles and uses of color have been employed to help indicate active and selected views and editors more clearly.
- 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

![Progress View](image)

*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 click it to immediately return to your work.
What was new in the IDE in 6.3.0 SP2?

This dialog also shows you the details of other currently running operations in the workspace and informs you when one operation is blocked and 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.
What was new in the IDE in 6.3.0 SP2?

- 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 — drop-down list of editors supports type-ahead.

Themes

The QNX Momentics IDE contains basic support for themes to allow for the customization of colors and fonts used in the workbench.

Background workspace auto-refresh

Changes made in the local filesystem can now be automatically refreshed in the workspace. This automatic refresh 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 deactivated from the Workbench preference page.
What was new in the IDE in 6.3.0 SP2?

Regular expressions in Find/Replace dialog

The Find/Replace dialog for text editors 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 light bulb appears at the upper-left corner of the field.

New text editor functions

You can 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 was new in the IDE in 6.3.0 SP2?

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 was new in the IDE in 6.3.0 SP2?

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 you 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 was new in the IDE in 6.3.0 SP2?

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 was new in the IDE in 6.3.0 SP2?

Code folding

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

Makefile editor

The Makefile editor has a new Preferences dialog and supports code folding.
What was new in the IDE in 6.3.0 SP2?

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
What was new in the IDE in 6.3.0 SP2?

- Breakpoint filtering
- Workspace variable support
- Mixed source/assembly
- Global variables
- Debug console
- Automatic refresh options
- Detailed 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’s active for.

![Properties dialog box](https://via.placeholder.com/150)

**Breakpoint filtering**

The Breakpoints view now lets you filter out all of the irrelevant breakpoints based on the specific process that you’re debugging.
What was new in the IDE in 6.3.0 SP2?

**Workspace variable support**

C/C++ launch configurations now include support for workspace variables in the Environment, Argument, and Working Directory tabs.

**Mixed source/assembly**

You no longer have to toggle the C/C++ editor to show the assembly of a program. Instead, use the Disassembly view to see both assembly code and source mixed:
What was new in the IDE in 6.3.0 SP2?

Global variables

You can add global variables to the Variables view instead of having to add them as separate expressions.

![Variables view]

Debug console

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

![Debug Console]

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.
What was new in the IDE in 6.3.0 SP2?

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 was new in the IDE in 6.3.0 SP2?

Detailed 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 was new in the IDE in 6.3.0 SP2?

**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.

This is for projects being built with one of the platform-specific *nto*-gcc drivers and a custom *Makefile*. 
What was new in the IDE in 6.3.0 SP2?

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. You can improve performance by limiting the directories and files of large projects.
What was new in the IDE in 6.3.0 SP2?

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 was new in the IDE in 6.3.0 SP2?

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 was new in the IDE in 6.3.0 SP2?

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
- Index error markers

C/C++ content assist

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

- classes and structure members
- local and global variables
- functions
- preprocessor defines
- preprocessor commands
What was new in the IDE in 6.3.0 SP2?

```c
struct mystruct {
    struct mystruct *next;
    int first;
    long second;
};

int main(int argc, char **argv) {
    struct mystruct s;
    s.
}
```

Configure completion options in the global C/C++ Editor Preferences.
What was new in the IDE in 6.3.0 SP2?

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.
What was new in the IDE in 6.3.0 SP2?

Open type

Use **Navigate→Open type** (Ctrl-Shift-T) to open the declaration of C/C++ classes, structures, unions, typedefs, enumerations, and namespaces.

![Open Type Window]

**C/C++ Browsing perspective**

Use the C/C++ Browsing perspective to navigate the class and structure members of a particular project.
What was new in the IDE in 6.3.0 SP2?

**Makefile editor**

The Makefile editor now provides syntax highlighting, code completion, and content outlining capabilities.
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, by selecting Window→Preferences→C/C++→C/C++ Editor Preferences. Then you can use Ctrl-Click to move 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 that are missing path-configuration information.

Configure the indexer from the C/C++ Indexer project settings:
What was new in the IDE in 6.3.0 SP2?

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
What was new in the IDE in 6.3.0 SP2?

- deeper, configurable, backtrace information, configured separately for allocation tracing and error detection

- 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 was new in the IDE in 6.3.0 SP2?

System Profiler

The new features added to the System Profiler in 6.3.0 SP2 are:

- 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:

Code Coverage

Improved reporting output and export capabilities.
What was new in the IDE in 6.3.0 SP2?

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 was new in the IDE in 6.3.0 SP2?

The new Signal Information view:
What was new in the IDE in 6.3.0 SP2?

The Memory Information and Malloc Information views (formerly found in the Memory Analysis perspective) are now part of the System Information perspective.
What was new in the IDE in 6.3.0 SP2?

System Builder

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

- You can create projects with multiple IFS images. You can combine an IFS with one or more EFS images while building the project.
- There are more ways to add a filesystem image to the existing project.
- You can build each IFS or EFS component separately.
What was new in the IDE in 6.3.0 SP1?

- You can combine images as a separate step. You can define images to combine at that point, and you can dynamically change the combination parameters for each component.

- The System Builder displays the filesystem layout for each IFS or EFS image.

- The System Optimization component is more flexible.

What was new in the IDE in 6.3.0 SP1?

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

- System Profiler
  - improved scalability and performance
  - improved graphic timeline display for events
  - additional filters: state activity, IPC activity, and CPU usage
What was new in the IDE in 6.3.0 SP1?

- Improved documentation, including more extensive code importing procedures, etc.
- The System Builder perspective now supports projects with more than one buildfile, and the perspective’s icons have been improved.
- 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.
- New Support for Intel’s C compiler (icc).
- The Code Coverage perspective now works with gcc 3.3.1 or later.
Appendix E

Migrating from Earlier Releases

In this appendix...

- Introduction  619
- From 6.3.0 SP1, SP2, or SP3 to IDE 4.0  620
- From 6.3.0 to 6.3.0 Service Pack 2 or Service Pack 3  620
- From 6.2.1 to 6.3.0  627
- From 6.2.0 to 6.3.0  631
You can easily migrate your old workspace and projects to this release.

**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:

```bash
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, 6.2.1, or 6.3), you’ll have to take different steps to convert your projects.
From 6.3.0 SP1, SP2, or SP3 to IDE 4.0

You can upgrade to IDE 4.0 from 6.3.0 SP1, SP2, or SP3 — whether or not you’ve also installed the QNX Neutrino Core OS 6.3.2. The information for migrating from 6.3.0 to SP2 or SP3 still applies; see below.

Note the following:

- The default workspace for IDE 4.0 is ide4-workspace, whereas in SP2 and earlier, the default was workspace, so now there’s less chance of accidentally migrating your old workspace.

- When you import existing projects, you now have the option of making a copy of it in your workspace. This is preferable because it leaves the original untouched as a backup. See “Importing projects” in the Common Wizards Reference chapter.

From 6.3.0 to 6.3.0 Service Pack 2 or Service Pack 3

In addition to the many fixes and enhancements to the QNX plugins, Service Pack 2 (which is also incorporated in Service Pack 3) 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 need to revert to an older version of the IDE, be sure to read the “Reverting to an older IDE” section.

You will receive an error message during this process with the following text:


This message is caused by internal changes to many of the perspectives commonly used for C/C++ development. You can safely ignore this error.

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:
From 6.3.0 to 6.3.0 Service Pack 2 or Service Pack 3

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:

![Properties for HelloWorld]

---

Appendix: E • Migrating from Earlier Releases
4 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:

![Properties for Hello](image)

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**

This section describes the following issues:

- Intel ICC error parser
From 6.3.0 to 6.3.0 Service Pack 2 or Service Pack 3

- 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

**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>Intel C/C++ Compiler Error Parser</td>
</tr>
<tr>
<td>QDE Extra Make Error Parser</td>
</tr>
<tr>
<td>CDT GNU Make Error Parser</td>
</tr>
<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 Visual C 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.
From 6.3.0 to 6.3.0 Service Pack 2 or Service Pack 3

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 receive errors when performing a file search (Search→File...):

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.
From 6.3.0 to 6.3.0 Service Pack 2 or Service Pack 3

This can cause problems if you try to load the project into an older version of the IDE.

If you plan to revert 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:
From 6.2.1 to 6.3.0

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:
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]

   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]

   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…→Update Make Projects (Commands tab).

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:
From 6.2.0 to 6.3.0

1. 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:

   ![Different Workspace Version Dialog]

   This workspace was written with a different version of the product and needs to be updated.
   Workspace location: C:\work\61workspace
   Updating the workspace may make it incompatible with other versions of the product.
   Press OK to update the workspace and open it. Press Cancel to exit with no changes.

   OK  Cancel

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:

   ![Cannot Preserve Layout Dialog]

   The saved user interface layout is in an obsolete format and cannot be preserved.
   Your projects and files will not be affected.
   Press OK to convert to the new format.
   Press Cancel to exit with no changes.

   OK  Cancel

   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:
From 6.2.0 to 6.3.0

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).
   • 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 Momentics 6.2.0 to QNX Momentics 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

! .bsh file (QNX System Builder) 161 .cdtproject file 18, 43, 82, 92, 468, 474 .efs file (QNX System Builder) 169 .ifs file (QNX System Builder) 168 .kev files (System Profiler) 386 .launch file 531 .metadata folder 468, 553 .project 18, 43, 82, 92, 171, 474 .sysbldr_meta file 171 .zip archives 104

A adaptive partitions creating at boot time 165 interactively 381 information about 373, 445 module, adding to OS image 195 parameters, setting 165, 379 programs, running in 165 Add New Target 206 addr2line 557 advanced mode (Properties dialog) 486 using to override regular options 486 All Processes pane (System Summary view) 348 analysis tools specifying for launch 532 Application Processes pane (System Summary view) 348 Application Profiler 221–241 specifying for launch 533 Application Profiler editor 236 colors used in 236 ads module, adding to OS image 195 ads utility 165 APS view 373 Arguments (Process Information view) 355
Index

Arguments tab (launch configurations dialog) 519, 521
assumptions in this guide xxv
autobuild enabling 47

B
back-trace depth 279
binaries seeing usage messages for 159
Binary Inspector view 156, 159
Binary Parser tab (New Project wizard) 469
block coverage 245
defined 246
bookmarks for Help documents 12
Bookmarks view 435
boot order 163
boot scripts 161, 165
branch coverage 245
currently not provided in the IDE 246
defined 246
breakpoints 136–141
removing 141
BSPs
buildfiles ship with 160
filename conventions in 167
importing source into the IDE 104
BSPs perspective 110
buffer overflow errors (MAT) 291
build all projects 49
automatic feature 47
configurations 97
CPULIST 53
enabling the autobuild feature 47
executables with code coverage enabled 248
options 53
order of projects 51
saving resources before 50
selected projects 50
terminology 47
variants 45
Build Command field (New Project wizard) 465
Build Setting field (New Project wizard) 465
Build Variants tab (New Project wizard) 462
buildfile defined 160
importing a 174

C
C project
C/C++ indexer 46
Standard Make, as distinct from a QNX multivariant project 17
C++ library 482
C/C++ Development perspective 39–76
C/C++ editor 39, 59–69
   adding comments to code in 66
color 66
debugging from 129
font 66
line numbers 66
preferences 66
C/C++ indexer
   choosing 46
disabling 46
C/C++ Indexer tab (New Projects wizard) 471
C/C++ Local launch configuration 512
C/C++ Postmortem debugger launch configuration 512
C/C++ Projects view 39, 40
   adding standard includes and defines 77
   compared with Navigator view 41
C/C++ QNX PDebug (Serial) launch configuration 512
C/C++ QNX QConn (IP) launch configuration 512
call counts  See also call pairs
call pairs 240
Calls Made pane (Malloc Information view) 361
case
don’t use to distinguish a project or file 16
channels
   shown in the System Blocking Graph view 363
clean
defined 47
selected projects 50
code coverage 248, 252
code coverage 245–259
   block coverage 245
   branch coverage 245
   combining sessions 255
   defined 245
   enabling 248
   for non-QNX projects 248
   icons 255
IDE tool is a visual font end to gcov 245
   launch, specifying for 534
   line-by-line information 256
   markers 259
   measures quality of tests 245
   scan interval 251
   should be set for only one build variant at a time 248
   summary report 259
   viewing reports in a browser 257, 261
Code Coverage Report view 256, 257, 260
Code Coverage Sessions view 254
coexistence of OS versions 20
PhAB 21
Photon 21
colors
   for signals 363
   in the C/C++ editor 66
timeline editor 447
Combine images (QNX System Builder) 177
combined image 166, 169
comments
Index

adding to code in C/C++
editor 66
Common tab (launch configurations
dialog) 519, 530
communications
IP 27
serial 27
compiler
optimization levels 479
selecting type of 478
specifying command-line
options 479
specifying defines to be passed
to 479
warning levels 479
Compiler tab (Properties
dialog) 477
Condition Statistics view 387
Connection Information view 341,
344, 366
Console view 47
in the Debugging
perspective 150
containers
build configurations
creating 100
editing 101
build configurations for 100
building 103
creating 98
defined 97
editing configurations for 101
seeing tree view of 101
Content Assist 60
conventions
for filenames in QNX
BSPs 167

for filenames in the QNX
System Builder 168
typographical  xxvi
Core Requests pane (Malloc
Information view) 361
CPU 201
CPUDIR 201
CPULIST 53
Create image (QNX System
Builder) 176
Ctrl-F3 (jump to function
implementation) 76
CVS
default source-management
system in the IDE 81
CVS Repositories view 84
CVS Repository Exploring
perspective 82

D

DDK
importing source into the
IDE 105
Debug perspective
views in 132
Debug view 124
selections in drive other
views 124
Debug/Release mode
for adding extra libraries (Linker
tab in Properties dialog)
484
debugger 119–152
Index

options in launch configuration 527
specifying source locations for 529
Debugger tab (launch configurations dialog) 519, 525
debugging
agent 30
assembly-language functions 131
building an executable for 120
controls for 126
session
controlling a several programs simultaneously 119
via C/C++ editor 129
via hotkeys 128
without the Debug view 128
dev-c-pty 521
Dietician (QNX System Builder) 202
Dinkum C++ library 482
disassembly mode (Debugger) 131
Discovery Options tab (New Project wizard) 470
Distribution pane (Malloc Information view) 361
Download tab (launch configurations dialog) 519, 523
drag-and-drop 84, 110

e
Eclipse
consortium 3
documentation on using CVS in the IDE 82
Platform 3
editors alternate inside the IDE 15, 68
not integrated with the IDE 15
outside the IDE 15, 68
C/C++ 59, 77
Content Assist feature 60
defined 14
enhanced source navigation 77
jumping to function implementation (Ctrl-F3) 76
jumping to function prototype (F3) 76
opening headers (Ctrl-Shift-o) 76
QNX System Builder 157
System Profiler 386
Element Statistics view 387
Environment tab (launch configurations dialog) 519, 523
Environment tab (New Projects wizard) 471
environment variables
CPU 201
CPUDIR 201
CPULIST 53
HOME 553
LD_PRELOAD 305, 306
MAKEFLAGS 23
MALLOC_CTHREAD 306
MALLOCP_TRACE 305
PATH 123
PHOTON 218
PLATFORM 201
PROFDIR 231
PROJECT 201
QCONN_PROFILER 231
QNXP.Configuration 20, 21, 23
QNXP.Host 23
QNXP_TARGET 23, 200, 201, 480
QNXP_TARGET_CPU 202
setting in launch configurations
dialog 523
SOCK 31
TMPDIR 23
VARIANT 202
WORKSPACE 200, 202
Environment Variables pane (Process
Information view) 354
error log file 553
Error Parsers tab (New Project
wizard) 466
errors
markers used to indicate 70
executables
building for debugging 120
sending fresh copy of whenever
you run or debug 524
stripping debug information
before downloading 525
unique name of for each
download session 525
Export wizard 111
expressions
evaluating/examining in the
debugger 142
Expressions view 142
extra libraries
adding to launch
configuration 525

F
F3 (jump to function prototype) 76
Fast index 46
FDs
side channels shown for 367
file name conventions
in QNX BSPs 167
in the QNX System
Builder 168
File System Navigator view 205
files
associations with editors 69
bringing into a project
folder 455
created outside the IDE 455
creating from scratch 69
exporting 110
filtering 42
host-specific 23
IDE removes from target after
downloading 525
importing 84
locations in workspace 553
moving between host and target
machines 205
opening 41
opening headers 76
target-specific 23
Filesystem pane (QNX System Builder) 157
filtering 400
flash filesystem
  appending to an OS image 179
flashcmp 557
font
  in the C/C++ editor 66
Full indexer 46
functions
  finishing names of in the editor 60
  jumping to source for 76
heap usage 358
hello world 44
help
  creating bookmarks in 12
  hover for function synopsis 66
  HTML server within the IDE 9
  navigating 11
  tips and tricks 12
History pane (Malloc Information view) 362
HOME 553
hosts
  host-specific files, location of 23
  hovering (in System Profiler editor) 435

G

gcc 23, 245, 246, 248, 251, 252, 478, 557
gcov 245
gdb 120, 151, 512, 527, 557
GDB
  using directly from the IDE 152
General Resources display (System Resources view) 369
General Statistics view 387
gmon.out file 224
  importing a 232
gprof 223

H

header files, opening 76
Index

Identification Details pane (Process Information view) 354
image
  adding files to 187
  booting 163
  combined 166
  combines 169
downloading a 180
final format of 179
flash filesystem 166, 169
OS 166, 168
properties of 190
types of in QNX System Builder 166
Images directory 171
Images pane (QNX System Builder) 157
Import wizard 84
importing
  PhAB projects 216
include paths
  adding standard includes and defines 77
  specifying 480
Indexer See C/C++ indexer
instrumented kernel 387
Intel C/C++ compiler 478, 624
IOFlags column (Connection Information view) 366
IP communications 28
IPL 163, 164
  adding to an image 178
K
kernel
  instrumented 387
  performance 387
keyboard shortcuts 19
L
launch configurations 56, 511–534
  creating for debugging 514
  creating for running 515
  dialog 518
  tabs in 518
for a debuggable executable 121
launch in background 532
list of favorites 517, 532
not automatically deleted when you delete associated projects 59
old, removing 235
types of 511
launcher 19
  running a program without 516
launching executable 57
ld 557
LD_PRELOAD 305, 306
libraries
  buttons for adding (Linker tab in Properties dialog) 484
  how to build 458
  optimizing in QNX System Builder 202
  shared 459
Index

shared without export 460
specifying locations of for the linker 483
static 459
static for shared objects 459
line numbers
  how to show them in the C/C++ editor 66
link map
  generating a 481
linked resources 84
linker
  command-line options 483
Linker tab (Properties dialog) 480
log files (System Profiler) 386

M

Main tab (launch configurations dialog) 519
make 557
targets
  adding 53
  removing 54
Make Builder tab (New Project wizard) 464
Make Project Migration wizard 629
Makefile
  recursive hierarchy 44
MAKEFLAGS 23
MALLOC_CTHREAD 306
MALLOC_TRACE 305
Malloc Information view 341, 344, 358, 359
Managed Make project
  C/C++ indexer 46
  markers
    for errors 70
memory
  changing a process’s 146
  error types 282
  errors 265–358
  illegal deallocation of 282
  leaks 274
  management 266
  resource leaks 282
  snapshots 280
Memory Analysis 273
  Buffer overflow 291
  disabling 306
  enabling 283
  enabling error detection 287
  environment variables 302
  error messages 300
  error types 282
  illegal deallocation of memory 286
  interpreting errors 282
  memory errors 278, 280
  memory leaks 297
  memory tracing 279
  NULL pointer dereference 288
  resource leaks 282
  runtime errors 282
  trace events 283
Memory Analysis perspective 273
Memory Analysis tool
  launching a program to use 276
  specifying for launch 533
Memory Information view 341, 344, 355, 358
Index

Memory Resources display (System Resources view) 371
migrating
  involves two steps 619
to the current version of the IDE 619–632
mkefs 156, 557
mkifs 156, 557
mkimage 557
mkrec 557
mksbp 156
multivariant project
  as distinct from a Standard Make C project 17

N
natures 17
Navigator view 39
  compared with C/C++ Projects view 41
New Project wizard 44, 456
tabs in 461
New QNX Target System Project wizard 472

O
objcopy 557
Options tab (New Project wizard) 467
OS image
  components of 163
OS versions
  coexistence of 20
    PhAB 21
    Photon 21
  specifying for build 22
Outline view 43
Overrides directory 171
  must be first in the list of search paths 201

P
padding (QNX System Builder) 170
PATH 123
pathname delimiter in QNX
  Momentics documentation xxvii
pdebug 30, 123, 512, 557
perspectives
  C/C++ Development 39–76
  default views 13
  defined 13
  specifying which to switch to during launch 534
PhAB 211
  applications, launching 512
  editing code in 214
  how to close 214
  projects, importing 216
  reopening 214
Phindows 35
PHOTON 218
PLATFORM 201
platforms
  all are enabled by default 462
Index

how to specify which to build
for 462
position-independent code
(PIC) 459
post-build actions (Linker tab in
Properties dialog) 485
preferences 20
C/C++ editor 66
Problems view 47, 70
Process Information view 341, 344,
348
Processes pane (System Summary
view) 348
procnto 163, 165, 194
   naming convention 165
   variants 165
procnto-instr 387
PROFDIR 231
profiling
   a running process 228
   building a program for 224
   instrumented 223
   non-QNX projects 226
   per-function 237
   per-line 235
   per-thread 238
   postmortem 223, 231
   running and profiling a
   process 226
   running as root 228
   sessions
   how to control 233
   statistical 222
   types of 222
programs
   creating 40
debugging for the first
time 514
relaunching a 518
running 40, 56
running for the first time 516
running/debugging once you’ve
created a launch
configuration 517
PROJECT 201
Project Name Prefix (BSP import
wizard) 107
project.bld file (QNX System
Builder) 161, 171
projects
   .cdtproject file 18, 43, 82,
   92, 468, 474
   .project file 18, 43, 82, 92,
   171, 474
   building 175
closing 41
container 97
converting to QNX type 474
creating 44
creating in the QNX System
Builder 173
defined 15
deleting 59
does not also delete launch
configuration 59
different types of 456
exporting 110
flash filesystem 175
how to create 457
importing 84
migrating from 6.2.0 to
   6.3.0 631
migrating from 6.2.1 to 6.3.0 629
names
don’t use case to distinguish 16
no spaces 16
nesting
steps to support 85
non-QNX
how to create 460
opening 41
properties of
setting 486
QNX System Builder 171
configuring 186, 198
target system 472
Projects tab (New Project wizard) 463
properties
image (QNX System Builder) 193
item (QNX System Builder) 197
Properties dialog
advanced mode 486
used when converting a project 476
Properties view
in QNX System Builder 186
qconfig 20, 21, 23
qconn 557
buffers, number of 391, 393
code coverage 246, 247
IP communications 28, 473
launch configuration 512
memory analysis 304
over Qnet 31
priority 397
processes, profiling 226
system, profiling 390, 391
target agent 18, 27
updating 31
QCONN_PROFILER 231
qde command 7
Qnet, running qconn over 31
QNX
recursive Makefile
hierarchy 44
QNX_CONFIGURATION 20, 21, 23
QNX_HOST 23
QNX_TARGET 23, 200, 201, 480
QNX_TARGET_CPU 202
QNX Application Profiler
perspective
configuring the IDE to
automatically change to
227
QNX BSP Perspective 110
QNX C or QNX C++ Project
relies on QNX recursive
Makefile hierarchy 44
QNX C/C++ project
as distinct from a Standard Make
C/C++ project 17
C/C++ indexer 46
Index

QNX GDB Console view
   enabling 151
QNX Memory Analysis
   perspective 265
   switching to automatically 282
QNX Momentics
   version of on host must be the
   same or newer than version
   on target 28
QNX Neutrino
   memory management in 266
   robust architecture of 266
QNX System Builder
   editor 157
   Filesystem pane 157
   Images pane 157
   toolbar 158
QNX System Builder
   perspective 155
QNX System Builder project
   creating a 173
QNX Target System Project
   creating a 56
QNX tools
   overview 4
QWinCfg 21

R

Reductions directory 171, 205
   must be second in the list of
   search paths 201
Registers view 143
regular mode (Properties
   dialog) 486
resource leaks 282
resources
   defined 19
   linked 84
ROM monitor 155, 180, 185
root
   all programs launched from the
   IDE run as 355
Run→Run... menu item 513

S

scan interval (code coverage) 251
sched_aps 165
scrolling (in System Profiler
   editor) 434
search paths (QNX System
   Builder) 199
Seek Offset column (Connection
   Information view) 367
selection
   types of in the System Profiler
   editor 433
sendto 155, 183
serial communications 30
serial terminal 180
   choosing a device 181
   communicating with your
   target 181
   communication
   parameters 181
   supported control
   sequences 180
   transferring files 182
Index

Server Processes pane (System Summary view) 348
Set QNX Build Environment wizard 77
shared libraries
  deleting reduced versions of 205
Shared Libraries view
  (Debugger) 147
shortcuts, keyboard 19
side channels 367
Signal Information view 341, 344, 363
signals
  color-coding for 363
  sending to a running process 345
  sending to a suspended program 149
Signals view (Debugger) 147
SOCK 31
source code
  exporting
    to a .zip file 110
    to the filesystem 110
  importing into the IDE 84
  importing QNX source code 104
  specifying locations of 479
Source tab (launch configurations dialog) 519, 528
spaces
  don’t use when naming a project or file 16
stack errors 358
startup
  naming convention 164
  variants 164
  startup code 163, 164
strip 557
stripped name (Linker tab in Properties dialog) 483
symbols
  loading 147
  stripping from a binary 481
System Blocking Graph view 341, 363
System Builder Console view 175
System Builder Projects view 161
System Information
  perspective 333
  CPU processing load and 346
  key terms used in 335
  updating views in 345
  views in 341
System Memory pane (System Summary view) 348
System Resources view 341, 367
  selecting which display to see 367
System Specifications pane (System Summary view) 348
System Summary view 341, 346
  All Processes pane 348
  Application Processes pane 348
  System Processes pane 348
  System Uptime display (System Resources view) 367
Index

T

target (machine) 27
  CPULIST 53
target agent  See qconn
Target File System Navigator view
  Add New Target option 206
Target Navigator view 341
  customizing 343
  sending a signal using 345
  using to control Information views 344
target system project
  creating a 472
target-specific files, location of 23
targets (make) 53
  adding 53
  removing 54
terminal emulation 521
TFTP server 180, 183
  TFTP Server view 155
Thread Details pane
  configuring 350
Thread Details pane (Process Information view) 350
Thread Information view 341
timeline editor
  colors for 447
Timeline State Colors pane 447
tips and tricks (item in Help menu) 12
TMPDIR 23
toolbar
  QNX System Builder 158
  Tools tab (launch configurations dialog) 519, 532
Total Heap pane (Malloc Information view) 361
Trace Event Log view 387
Trace Search 400
  tracing events 283
typographical conventions xxvi

U

update interval (QNX Application Profiler) 227, 229
Update Old Make Project icon 629
usage message
  displayed for each item in QNX System Builder 156
  for binaries in QNX System Builder 159
usemsg 557
utilities
  used by the QNX System Builder 176

V

variables
  preventing the debugger from reading 135
Variables view 135
VARIANT 202
views
  Application Profiler 234
  APS (Adaptive Partition Scheduling) 373
Index

Binary Inspector  156, 159
Bookmarks  435
C/C++ editor  39
C/C++ Projects  39, 40
Code Coverage Report  256, 257, 260
Code Coverage Sessions  254
Condition Statistics  387
Console  47
Debug  124
defined  14
Element Statistics  387
General Statistics  387
Navigator  39
Outline  43
Problems  47
System Builder Console  175
System Builder Projects  161
Target File System
    Navigator  205
TFTP Server  155
Trace Event Log  387

Make Project Migration  629
New Project  44, 461
New QNX Target System
    Project  472
Set QNX Build
    Environment  77
simple  455
workbench  8
    menus  9
Workbench Build Behavior field
    (New Project wizard)  466
Workbench User Guide  39
    references to CVS in  82
working directory
    on target machine  522
working set  100
Working Set Name (BSP import
    wizard)  107
WORKSPACE  200, 202
workspace
    .metadata folder  468, 553
    defined  5, 16
    migrating from 6.2.0 to
        6.3.0  631
    migrating from 6.2.1 to
        6.3.0  627
    refreshing  15
    specifying where the IDE should
        look for a  16

W

watchpoints  136–141
    removing  141
wizards  453–477
    are categorized according to
        natures  453
    conversion  629
    creating “nature-free” files,
        folders, or projects  455
    creating a new project  456
    how to access  453

X

XIP  194
Z

zooming (in System Profiler editor) 434