# Table of Contents

I. Getting Started ....................................................................................................................... 1

1. Preface ........................................................................................................................... 3

1.1. What is OpenRG? ............................................................................................... 3

1.2. Objective ............................................................................................................. 4

1.3. Who Should Read This Book ............................................................................. 4

1.4. What You Will Learn ......................................................................................... 5

1.5. How This Book is Organized ............................................................................. 5

1.6. Conventions Used in This Book ......................................................................... 6

1.7. Tell Us What You Think .................................................................................... 6

2. System Setup ................................................................................................................. 7

2.1. Software and Hardware Requirements ............................................................... 7

2.2. Improving the Efficiency of Your Development Process ................................... 8

3. The OpenRG SDK ........................................................................................................ 9

3.1. SDK Generated Content ..................................................................................... 9

3.2. Installation ......................................................................................................... 10

3.2.1. Quick Installation ................................................................................... 10

3.2.2. The Full Installation Procedure ............................................................. 10

3.2.3. RGLoader ............................................................................................... 11

3.3. The OpenRG License ....................................................................................... 11

3.3.1. The License File .................................................................................... 11

3.3.2. Evaluation vs. Commercial Modes ........................................................ 12

3.3.3. Troubleshooting ..................................................................................... 13

3.4. Development Tree ............................................................................................. 13

3.5. Toolchain .......................................................................................................... 14

3.6. System Utilities ................................................................................................. 15

4. Technical Support ........................................................................................................ 17

4.1. The Support Team ............................................................................................ 17

4.1.1. Role ........................................................................................................ 17

4.1.2. Structure ................................................................................................. 17

4.1.3. Contacting Jungo’s Support ................................................................... 18

4.2. Service Level Agreement ................................................................................. 18

4.3. Support Account Manager ................................................................................ 18

4.4. Issue Life Cycle ................................................................................................ 18

4.5. Using the Support Center ................................................................................. 20

4.5.1. Home Page ............................................................................................. 21

4.5.2. Considerations Before Opening an Issue ............................................... 21

4.5.3. Reporting an Issue ............................................................................... 22

4.5.4. Using Issue Queries ............................................................................... 23

4.5.5. Automatic Closure of Issues .................................................................. 23

4.6. Remote Debugging ........................................................................................... 23

4.6.1. Remote Computer Requirements ........................................................... 24

4.6.2. Secure Shell (SSH) ................................................................................ 24

4.6.3. Using DNS and a Redirection Service .................................................. 25

4.6.4. Setting Up a Serial Power Relay Device ............................................... 26

4.6.5. Protecting the Remote Computer ........................................................... 27

4.7. Feedback ........................................................................................................... 27
II. Developing with OpenRG .................................................................................................. 28
5. Development Tutorial .................................................................................................. 30
  5.1. Getting Started .................................................................................................. 30
  5.2. The Programming Example .............................................................................. 31
    5.2.1. Wag the Watchdog ............................................................................... 32
    5.2.2. Watchdog Components .......................................................................... 32
    5.2.3. Installation and Build ............................................................................. 33
    5.2.4. Configuring OpenRG – See the Watchdog Run .................................... 33
  5.3. Stage 1 – Adding a New Package .................................................................... 35
    5.3.1. New Package Directory Structure .......................................................... 35
    5.3.2. Working with MakeFiles ....................................................................... 36
    5.3.3. Initializing a New Package .................................................................... 37
    5.3.4. Lesson Summary .................................................................................... 39
  5.4. Stage 2 – The Configuration Database ............................................................. 40
    5.4.1. Adding the Watchdog Management Component ................................... 40
    5.4.2. Extending the Configuration Database .................................................. 41
    5.4.3. Defining the Management API .............................................................. 42
    5.4.4. Implementing CLI commands ............................................................... 44
    5.4.5. Setting Up Default Values ..................................................................... 48
    5.4.6. Testing the Application CLI Commands ............................................... 48
    5.4.7. Lesson Summary .................................................................................... 49
  5.5. Stage 3 – Creating a Management Page ........................................................... 50
    5.5.1. Creating the Watchdog WBM Component ............................................ 50
    5.5.2. Creating a Print Function ....................................................................... 51
    5.5.3. Creating a Scan Function ....................................................................... 53
    5.5.4. Tying the Watchdog WBM Page with the WBM .................................. 55
    5.5.5. Compilation ............................................................................................ 58
    5.5.6. Lesson Summary .................................................................................... 58
  5.6. Stage 4 – Integrating the Application ............................................................... 59
    5.6.1. Creating the Watchdog Process Component .......................................... 59
    5.6.2. A Simple Example ................................................................................. 60
    5.6.3. The Watchdog Process .......................................................................... 61
    5.6.4. Using the Local Management Utilities .................................................. 63
    5.6.5. Lesson Summary .................................................................................... 64
  5.7. Stage 5 – Integrating the Kernel Module ......................................................... 64
    5.7.1. Creating the Watchdog Kernel Module Component .............................. 65
    5.7.2. Creating the Kernel Module .................................................................. 66
    5.7.3. OpenRG's Generic Char Device ............................................................ 66
    5.7.4. Implementing a IOCTL ......................................................................... 67
    5.7.5. Watchdog Logic ..................................................................................... 69
    5.7.6. Compilation and Loading ...................................................................... 70
    5.7.7. Loading the Kernel Module During Boot Sequence .............................. 71
    5.7.8. Lesson Summary .................................................................................... 71
  5.8. Tutorial Appendix ............................................................................................... 71
  6. Board Tailoring ............................................................................................................ 74
    6.1. Overview ....................................................................................................... 74
    6.2. Flash Sections ............................................................................................... 75
      6.2.1. Boot .................................................................................................... 75

© 1998-2012 Jungo Ltd.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1. Links</td>
<td>117</td>
</tr>
<tr>
<td>10.2. On-change Events</td>
<td>118</td>
</tr>
<tr>
<td>10.3. Passing Parameters Between Scan and Print Functions</td>
<td>119</td>
</tr>
<tr>
<td>10.4. Editable Tables</td>
<td>123</td>
</tr>
<tr>
<td>10.5. Look and Feel</td>
<td>126</td>
</tr>
<tr>
<td>10.5.1. The Look and Feel Concept</td>
<td>127</td>
</tr>
<tr>
<td>10.5.2. Creating a New Theme</td>
<td>127</td>
</tr>
<tr>
<td>10.5.3. Changing the Look and Feel of the New Theme</td>
<td>128</td>
</tr>
<tr>
<td>10.6. Adding Static HTML Pages to the WBM</td>
<td>136</td>
</tr>
<tr>
<td>11. Tailoring the Role-based Permissions Engine</td>
<td>137</td>
</tr>
<tr>
<td>11.1. Overview</td>
<td>137</td>
</tr>
<tr>
<td>11.2. Defining Roles and Permission Sets in OpenRG's Code</td>
<td>138</td>
</tr>
<tr>
<td>11.3. Managing a Role's or a User's Set of Permissions at Runtime</td>
<td>139</td>
</tr>
<tr>
<td>11.4. OpenRG's Permission Validation Mechanism</td>
<td>140</td>
</tr>
<tr>
<td>11.5. Associating Permissions with the GUI Elements</td>
<td>141</td>
</tr>
<tr>
<td>12. Integrating Applications</td>
<td>144</td>
</tr>
<tr>
<td>12.1. Adding an Application to the Development Tree</td>
<td>144</td>
</tr>
<tr>
<td>12.2. Adding an Application to the Image</td>
<td>145</td>
</tr>
<tr>
<td>12.3. Testing Your Application</td>
<td>145</td>
</tr>
<tr>
<td>12.4. Inter Process Communication</td>
<td>145</td>
</tr>
<tr>
<td>12.4.1. Why IPC?</td>
<td>146</td>
</tr>
<tr>
<td>12.4.2. High-level Design</td>
<td>146</td>
</tr>
<tr>
<td>12.4.3. IPC APIs</td>
<td>147</td>
</tr>
<tr>
<td>12.4.4. Low-level Design</td>
<td>149</td>
</tr>
<tr>
<td>12.4.5. Tutorial</td>
<td>149</td>
</tr>
<tr>
<td>13. Integrating Kernel Modules</td>
<td>152</td>
</tr>
<tr>
<td>13.1. Loading Modules During Boot Sequence</td>
<td>152</td>
</tr>
<tr>
<td>13.2. Storing Module Files In the Ramdisk</td>
<td>154</td>
</tr>
<tr>
<td>14. Integrating Device Drivers</td>
<td>155</td>
</tr>
<tr>
<td>14.1. Adding Device Drivers</td>
<td>155</td>
</tr>
<tr>
<td>14.1.1. Adding Device Drivers to the Development Tree</td>
<td>155</td>
</tr>
<tr>
<td>14.1.2. Adding Device Drivers to the Image</td>
<td>156</td>
</tr>
<tr>
<td>14.2. Defining Network Devices and Network Bridges</td>
<td>156</td>
</tr>
<tr>
<td>14.2.1. Defining a Hardware Network Interface</td>
<td>156</td>
</tr>
<tr>
<td>14.2.2. Defining a Network Bridge</td>
<td>157</td>
</tr>
<tr>
<td>14.2.3. Allowing Absent Devices</td>
<td>157</td>
</tr>
<tr>
<td>14.3. Adding Network Device Types</td>
<td>157</td>
</tr>
<tr>
<td>14.3.1. When Would You Add a New Network Device Type?</td>
<td>158</td>
</tr>
<tr>
<td>14.3.2. The dev_if_ops_t Structure</td>
<td>158</td>
</tr>
<tr>
<td>14.3.3. Code Compilation for Localhost</td>
<td>160</td>
</tr>
<tr>
<td>14.4. Adding New IOCTLs</td>
<td>160</td>
</tr>
<tr>
<td>14.4.1. Defining the IOCTL</td>
<td>161</td>
</tr>
<tr>
<td>14.4.2. Implementing The IOCTL</td>
<td>162</td>
</tr>
<tr>
<td>III. Advanced Development Concepts</td>
<td>163</td>
</tr>
<tr>
<td>15. System Architecture</td>
<td>165</td>
</tr>
<tr>
<td>15.1. Architectural Design Guidelines</td>
<td>165</td>
</tr>
<tr>
<td>15.2. Source Code to Image Mapping</td>
<td>166</td>
</tr>
<tr>
<td>15.3. Component Partitioning</td>
<td>166</td>
</tr>
</tbody>
</table>
15.3.1. Coordination Logic ................................................................. 167
15.3.2. Configuration Database ....................................................... 169
15.3.3. Devices .............................................................................. 169
15.3.4. System Services ................................................................. 169
15.3.5. Management Tasks .............................................................. 170
15.3.6. OpenRG Components ......................................................... 171
15.3.7. Kernel .................................................................................. 171
15.3.8. KOS/KNET ......................................................................... 171
15.4. Network Devices ..................................................................... 171
15.4.1. Device States ..................................................................... 171
15.4.2. Types of Devices ............................................................... 172
15.4.3. Network Devices in OpenRG .............................................. 173
15.4.4. Debugging Devices ............................................................. 173
15.5. System Reconfiguration .......................................................... 173
15.6. Packet Flow Through OpenRG .............................................. 174
15.6.1. Knet Hooks – Introduction ................................................ 174
15.6.2. Rx Flow ............................................................................. 175
15.6.3. Tx Flow ............................................................................. 176
16. Main Task .................................................................................. 178
16.1. Compiling the Main Task Watchdog Tree .................................. 178
16.2. Understanding Asynchronous Programming and Main Task Architecture .................................................. 179
16.2.1. The Asynchronous Nature of Main Task .............................. 179
16.2.2. The Task ............................................................................ 179
16.3. The Entity ............................................................................... 180
16.4. Creating a New Directory and Modifying the Makefile ........... 180
16.5. Implementing the Watchdog Entity ......................................... 181
16.6. Registering the Entity in Main Task ........................................ 185
16.7. Implementing the Watchdog Task .......................................... 185
16.8. Writing the Makefile .............................................................. 188
17. Adding Applications to the Main Task ...................................... 189
17.1. Generic Task Design .............................................................. 189
17.1.1. Rules of Thumb ................................................................. 189
17.1.2. Context and <name>_t ......................................................... 189
17.2. Generic Task API ................................................................. 190
17.2.1. Task API by Example (PPP) .............................................. 190
17.3. Writing <name>_close() ........................................................ 191
17.3.1. The Problem ..................................................................... 191
17.3.2. What Callbacks Should be Protected? ............................... 192
17.3.3. The Solution ................................................................. 192
17.3.4. A Good Example ............................................................. 192
17.3.5. A Bad Example .............................................................. 193
17.4. Porting Processes to Tasks ................................................... 193
17.5. Naming Conventions ......................................................... 194
17.6. Asynchronous Programming ............................................... 194
17.6.1. Purpose ............................................................................ 194
17.6.2. Writing Asynchronous Code .............................................. 194
17.6.3. Examples ........................................................................ 194
17.6.4. Convention Notes ................................................................. 198
17.6.5. Synchronous Vs Asynchronous API .............................................. 198

18. Configuration Database ................................................................. 199
18.1. System Configuration Structure ..................................................... 199
  18.1.1. The Configuration File (rg_conf) ............................................. 199
  18.1.2. The RAM Configuration File (rg_conf_ram) ......................... 200
18.2. The Configuration Entries ........................................................... 200
18.3. Obscuring Data ........................................................................... 200
18.4. Booting OpenRG ........................................................................ 200
18.5. Restoring Defaults ..................................................................... 201
18.6. Reconfiguring OpenRG ............................................................... 201
18.7. Modifying the Configuration File .................................................. 201
18.8. Modifying RGLoader's Configuration ............................................. 202
18.9. Troubleshooting the Configuration File ........................................... 203

19. The Build Mechanism ..................................................................... 205
19.1. The Build Process ........................................................................ 205
  19.1.1. The Tree Configuration Stage ................................................... 205
  19.1.2. The Make Stage ...................................................................... 207
19.2. The Multi-distribution Build Concept ............................................ 208
  19.2.1. A Simple Make ........................................................................ 208
  19.2.2. Compiling Two Different Distributions .................................... 208
  19.2.3. Switching Between Distributions ............................................ 208
  19.2.4. Different Configurations of a Single Distribution .................... 209
  19.2.5. Defining the Build Directory Outside the Tree ....................... 209
  19.2.6. Parallel Builds ........................................................................ 209
19.3. Makefile Examples ....................................................................... 210
  19.3.1. A Simple Makefile ................................................................... 210
  19.3.2. Compiling Multiple Targets ..................................................... 210
  19.3.3. Compilation Flags ................................................................... 211
  19.3.4. Compilation Targets ............................................................... 212
19.4. Configuration Files ....................................................................... 215
19.5. Debugging Makefiles ................................................................. 217
19.6. Makefile Variables ....................................................................... 218
19.7. *.mak Files .................................................................................. 221
  19.8. Using C++ .................................................................................. 222
    19.8.1. Setting the Development Environment ................................ 222
    19.8.2. Linking ................................................................................ 223

20. Debugging ..................................................................................... 224
20.1. Debugging Methods ................................................................. 224
20.2. GNU Debugger ........................................................................... 225
  20.2.1. Basic GDB ............................................................................. 226
  20.2.2. Remote Debugging ................................................................. 228
20.3. Debugging a User-Mode Program ................................................. 229
  20.3.1. Remote Connection Setup ....................................................... 229
  20.3.2. Accessing the WBM via a PPPoS Connection ....................... 230
  20.3.3. GDB Server Setup ................................................................. 231
  20.3.4. Start GDB Server ................................................................. 231
  20.3.5. Start GDB on Host ............................................................... 232
20.3.6. Crash Dump ................................................................. 232
20.4. Kernel Debugging .......................................................... 234
   20.4.1. General Strategy: Swim Upstream .............................. 234
   20.4.2. KGDB Server Setup ................................................ 234
   20.4.3. Running Two Serial Channels on a Single Port .......... 235
   20.4.4. "Oops" .................................................................... 235
   20.4.5. System Map ........................................................... 236
   20.4.6. Object Dump .......................................................... 236
   20.4.7. KSYMS .............................................................. 237
   20.4.8. SHOW_TRACE ...................................................... 237
   20.4.9. Logging Console Output .......................................... 237
   20.4.10. Finding an Address in a Loaded Module ................. 237
20.5. Debugging Kernel Modules ........................................... 238
20.6. Finding an Address ....................................................... 238
20.7. Configuring GDB .......................................................... 240
   20.7.1. .gdbinit .............................................................. 240
   20.7.2. Dynamic Link ....................................................... 240
20.8. Cross-GDB Compilation ............................................... 240
20.9. Crash Analyze Tool ...................................................... 241
   20.9.1. Tool Description .................................................... 241
   20.9.2. Crash Example ..................................................... 242
20.10. User Mode Memory Debugging .................................... 243
20.11. Kernel Mode Memory Debugging ................................. 245
   20.11.1. Setup ............................................................... 245
   20.11.2. The CONFIG_RG_DEBUG_KMALLOC Method ........... 245
   20.11.3. Finding a Memory Leak ....................................... 246
20.12. Debugging Tips and Tricks .......................................... 247
   20.12.1. Running Debugging Applications with NFS ............ 247
   20.12.2. Debugging the WBM with Firebug ......................... 247
20.13. Using the Crash Logger ............................................... 247
IV. Feature Knowledge Base .................................................. 249
21. Version Management ...................................................... 251
   21.1. Backward Compatibility .......................................... 251
   21.2. An Example .......................................................... 253
22. BusyBox ......................................................................... 255
   22.1. Adding Applets ....................................................... 255
23. KOS/KNET ..................................................................... 257
   23.1. KOS .................................................................. 257
   23.2. KNET .................................................................. 258
   23.2.1. Knet Hooks ......................................................... 258
24. Remote Upgrade ............................................................ 263
   24.1. Overview .............................................................. 263
   24.2. Remote Upgrade Behavior ....................................... 263
   24.3. The Remote Upgrade File Utility ................................. 264
       24.3.1. Creating a New Upgrade File ......................... 264
       24.3.2. Adding a Product Header ...................... 264
       24.3.3. Listing Products in the Upgrade File .............. 264
   24.4. Backwards Compatibility ....................................... 264
27.2.7. cwmp ................................................................. 305
27.2.8. dev ................................................................. 305
27.2.9. docsis ............................................................. 306
27.2.10. factory ........................................................... 306
27.2.11. fastpath ......................................................... 307
27.2.12. firewall ......................................................... 307
27.2.13. firmware_update ........................................... 308
27.2.14. flash ............................................................. 308
27.2.15. inet_connection ............................................ 310
27.2.16. jnet .............................................................. 310
27.2.17. kernel ............................................................ 310
27.2.18. log ................................................................. 311
27.2.19. misc .............................................................. 311
27.2.20. net ................................................................. 312
27.2.21. psb6973 ........................................................ 313
27.2.22. pvc ................................................................. 314
27.2.23. qos ................................................................. 314
27.2.24. scr ................................................................. 314
27.2.25. service ........................................................... 314
27.2.26. switch ............................................................ 314
27.2.27. system ........................................................... 315
27.2.28. upnp ............................................................. 316
27.2.29. wbm .............................................................. 317
27.2.30. wireless ........................................................ 317
27.3. Adding a New Command Line Function ......................... 318
  27.3.1. Writing Functions ............................................ 318
  27.3.2. Defining a Command Array ................................. 319
  27.3.3. Defining a Category .......................................... 319
  27.3.4. Registering Commands ...................................... 319
  27.3.5. Removing Commands ........................................ 320
  27.3.6. Example – Adding Commands ......................... 320
  27.3.7. Notes ............................................................. 321
28. Web-based Management .................................................. 322
  28.1. Overview ............................................................ 322
    28.1.1. High Usability .............................................. 322
    28.1.2. Customization .............................................. 323
    28.1.3. Flexibility .................................................... 324
  28.2. Directory Structure .............................................. 324
  28.3. General Structure and Flow .................................... 325
    28.3.1. The HTTP Connection Life Cycle ...................... 325
    28.3.2. Main Function .............................................. 327
    28.3.3. Print Page Function ........................................ 328
    28.3.4. Scan Page Function ....................................... 329
    28.3.5. Input Validation ............................................ 331
    28.3.6. Initialization ................................................ 333
  28.4. The Configuration File .......................................... 334
  28.5. Debugging Techniques .......................................... 335
    28.5.1. Common Problem 1 ........................................ 335
28.5.2. Common Problem 2 ........................................................................... 336
28.6. Multilanguage Support ........................................................................... 338
  28.6.1. Language Files .............................................................................. 339
  28.6.2. Adding Language Strings .............................................................. 339
  28.6.3. Adding a New Language ................................................................. 340
29. OpenRG Over UML ........................................................................................ 343
  29.1. Overview ............................................................................................ 343
  29.2. Installing UML ..................................................................................... 343
  29.3. The Virtual Flash Layout .................................................................... 343
  29.4. Running OpenRG over UML ............................................................... 344
  29.5. Virtual Networking for a Virtual Machine ........................................... 346
  29.6. Virtual Removable Storage ................................................................. 346
  29.7. Debugging .......................................................................................... 347
    29.7.1. User-Mode Debugging ................................................................. 347
    29.7.2. Kernel-Mode Debugging ............................................................... 347
30. Firewall ............................................................................................................. 348
  30.1. General ............................................................................................... 348
    30.1.1. Overview ..................................................................................... 348
    30.1.2. Terminology ................................................................................ 349
    30.1.3. Flow Description ........................................................................ 352
    30.1.4. Directory Structure .................................................................... 353
    30.1.5. Rules and Rule Structure ............................................................. 354
    30.1.6. rg_conf Structure ...................................................................... 355
    30.1.7. The Reconf Mechanism ............................................................... 356
  30.2. Usermode Implementation ................................................................... 356
    30.2.1. Opcodes ...................................................................................... 356
    30.2.2. Services ....................................................................................... 357
    30.2.3. Chains ......................................................................................... 359
    30.2.4. Ruleset ....................................................................................... 359
    30.2.5. VM_CONN Mechanism ............................................................... 364
  30.3. Kernel Implementation ......................................................................... 364
    30.3.1. Packet Flow ................................................................................ 365
    30.3.2. Connections ................................................................................ 365
    30.3.3. Application Level Gateway (ALG) ............................................. 370
    30.3.4. Network Address Port Translation (NAPT) ................................. 378
    30.3.5. Opcode Handling ....................................................................... 379
    30.3.6. NAT/Connection Mapping ......................................................... 379
    30.3.7. Firewall Fastpath ......................................................................... 380
    30.3.8. Fragments Handling ................................................................. 380
    30.3.9. IP Statistics .............................................................................. 381
  30.4. Debugging Methods ............................................................................ 381
    30.4.1. CLI Commands .......................................................................... 381
    30.4.2. Analyzing the Firewall Trace Output ........................................ 383
31. Quality of Service in a Nutshell ..................................................................... 385
  31.1. Scheduling and Shaping the Egress Traffic ....................................... 385
    31.1.1. Using the Queuing Disciplines .................................................. 385
    31.1.2. Packet Classification Mechanisms ............................................. 388
    31.1.3. Packet Priority Mappings ............................................................ 389
31.1.4. The TCP Serialization Control .......................................................... 389
31.2. Managing The Ingress Traffic ................................................................. 389
31.3. The QoS Source Code ............................................................................. 390

32. Dynamic DNS .......................................................................................... 391
32.1. Overview ............................................................................................... 391
32.2. Adding A New Provider ......................................................................... 392
    32.2.1. Main Task API ............................................................................. 392
    32.2.2. Provider-Specific WBM Fields .................................................... 395

33. Virtual Private Network (VPN) ................................................................. 397
33.1. Public Key Infrastructure ....................................................................... 397
    33.1.1. Digital Certificates ...................................................................... 398
    33.1.2. X.509 Certificate Format ............................................................. 398
    33.1.3. Replacing OpenRG's Certificate .................................................. 399
33.2. Relevant rg_conf Entries ...................................................................... 399

34. Adding a Proxy .......................................................................................... 400

35. Voice over IP (VoIP) ............................................................................... 402
35.1. Compiling VoIP .................................................................................... 403
    35.1.1. Compiling an ATA Image ............................................................ 403
    35.1.2. Compiling a PBX Image ............................................................... 404
35.2. The Signaling and Voice Stages .............................................................. 404
35.3. Reference Hardware Platforms .............................................................. 404
35.4. Architecture .......................................................................................... 405
    35.4.1. Asterisk Architecture ................................................................. 405
    35.4.2. Digital Signal Processor ............................................................. 409
    35.4.3. RTP/RTCP .................................................................................. 412
    35.4.4. SIP ALG ..................................................................................... 412
    35.4.5. Quality of Service ................................................................. 413
35.5. Modifying the Voice Module ................................................................. 414
    35.5.1. Integrating Additional DSPs ......................................................... 414
    35.5.2. Adding New Features to Asterisk ............................................... 415
    35.5.3. Web-based Interface Tailoring ..................................................... 418
    35.5.4. Configuration File Entries ............................................................ 418

36. Secure Socket Layer (SSL) VPN .............................................................. 419
36.1. Adding a New SSL VPN Application .................................................... 419
36.2. How to Modify jar Files Location ........................................................ 422

37. TR-069 and Extensions .......................................................................... 423
37.1. Overview .............................................................................................. 423
37.2. Architecture Implementation ............................................................... 424
    37.2.1. File Structure ............................................................................. 424
    37.2.2. Parameters ................................................................................. 424
    37.2.3. Adding a New Parameter ............................................................ 426
37.3. Jungo Extensions .................................................................................. 427
37.4. JACS Implementation .......................................................................... 428
37.5. TR-069 Parameter List .......................................................................... 430
37.6. TR-098 Parameter List .......................................................................... 449
37.7. TR-104 Parameter List .......................................................................... 458
37.8. TR-111 Parameter List .......................................................................... 462

38. TR-064 .................................................................................................... 463
38.1. Overview ....................................................................................................... 463
38.2. Architecture Implementation ........................................................................ 463
  38.2.1. Engine Structure ................................................................................ 464
  38.2.2. Implementation .................................................................................. 465
  38.2.3. Adding New Objects .......................................................................... 467
38.3. Parameter List ............................................................................................... 467
39. Network Acceleration .............................................................................................. 483
  39.1. Fastpath Hardware Acceleration ................................................................... 483
  39.2. Fastpath Software Acceleration .................................................................... 484
    39.2.1. Architecture ........................................................................................ 485
    39.2.2. An In-depth Look at PSE .................................................................. 487
39.1. Testing OpenRG's Networking Performance ...................................................... 490
  40. Asynchronous JavaScript and XML (AJAX) ....................................................... 490
    40.1. OpenRG AJAX Architecture ........................................................................ 490
    40.2. AJAX Server Functionality .......................................................................... 493
41. Testing OpenRG's Networking Performance ...................................................... 497
  41.1. Hardware Requirements ................................................................................ 497
  41.2. Testing OpenRG with SmartBits .................................................................... 497
    41.2.1. Physical Connection and Configuration Reference ............................ 498
    41.2.2. Preparing OpenRG For Testing ......................................................... 499
    41.2.3. Setting an ARP Table on OpenRG .................................................... 499
  41.3. Testing OpenRG in Routing Mode ............................................................... 500
    41.3.1. OpenRG Configuration ...................................................................... 500
    41.3.2. SmartBits Configuration .................................................................... 502
  41.4. Testing OpenRG in NAPT Mode .................................................................... 506
  41.5. Testing OpenRG in Bridge Mode .................................................................... 508
    41.5.1. OpenRG Configuration ...................................................................... 508
    41.5.2. SmartBits Configuration .................................................................... 510
  41.6. Testing OpenRG's IPSec Connection ........................................................... 514
    41.6.1. OpenRG Configuration ...................................................................... 514
    41.6.2. SmartBits Configuration .................................................................... 517
42. Supporting Separate Networks with Routing Groups ............................................. 521
  42.1. Overview ....................................................................................................... 521
  42.2. Routing Groups ............................................................................................. 521
  42.3. Routing Tables .............................................................................................. 522
  42.4. Adding Routes to Tables .............................................................................. 522
  42.5. Routing Decision Rules ................................................................................ 523
V. Appendix .................................................................................................................. 525
43. Working with the Concurrent Versions System ................................................... 527
  43.1. What is CVS? ............................................................................................... 527
  43.2. The Advantages ............................................................................................ 527
  43.3. Setting up CVS on Your Computer .............................................................. 528
  43.4. Initial Import of OpenRG into CVS ............................................................. 529
    43.4.1. Downloading JCVS ........................................................................... 530
    43.4.2. Downloading the Source Code Packages ........................................... 530
    43.4.3. Importing the Source Tree into CVS ................................................. 530
    43.4.4. Porting the Source Tree into the Main Trunk .................................... 531
  43.5. Importing Newer Versions ........................................................................... 531
  43.6. Generating Distribution Upgrade Patch Files .............................................. 532
Part I  Getting Started
# Table of Contents

1. Preface ................................................................................................................................... 3  
2. System Setup ......................................................................................................................... 7  
3. The OpenRG SDK ................................................................................................................ 9  
4. Technical Support ................................................................................................................ 17
Preface

1.1 What is OpenRG?

OpenRG™ is a leading software solution for broadband service delivery. Its unique hardware-independent design enables service providers to concentrate on enhancing service to their subscribers, offering full flexibility to select the best suited CPE, and eliminating the complexity and costs that are typically associated with using multiple CPE models and deploying new ones.

OpenRG is a best-of-breed middleware product for residential gateways, which resides in the CPE. The middleware includes drivers, OS, protocols and advanced applications to enable all of the broadband applications and services.

OpenRG empowers various network devices in the digital home, including triple play residential gateways, home/SOHO routers, home gateways, wireless access points, cable/DSL routers, voice gateways and more.
Broadband operators worldwide have selected and deployed OpenRG-based home gateways to drive revenue generating business models and services.

Jungo also offers OpenSMB—gateway middleware for the small and medium-sized business market. References to OpenRG in this document also apply to OpenSMB.

You can view OpenRG’s specification as well as additional documentation at http://www.jungo.com/openrg/documentation.html, version 5.5.

1.2 Objective

This book is intended to serve as a resource for the OpenRG community. It describes methods for setting up your host computer, adding applications and drivers, tailoring the Web-based management and other ways of extending OpenRG.

The Programmer’s Guide is complemented by the Configuration Guide, which provides an explanation and description for every entry in OpenRG’s configuration file, rg_conf, responsible for controlling every OpenRG feature. In addition, as OpenRG is POSIX compliant, documentation for standard Linux and POSIX modules is provided in Jungo’s website, at http://www.jungo.com/openrg/linux_doc.html.

1.3 Who Should Read This Book

The intended audience for this book is a computer programmer with a working knowledge of the C programming language, Linux/RTOS operating system, embedded systems and networking protocols and applications.
1.4 What You Will Learn

Here are some of the more important topics dealt with in this book:

• Adding applications and device drivers
• Tailoring OpenRG's Web-based management
• Integrating existing management interfaces with OpenRG
• Planning for mass-production

1.5 How This Book is Organized

The Programmer's Guide is divided into five parts, each exploring a part of OpenRG. It provides detailed explanations and tutorials describing the OpenRG environment and how to use it.

Part I: Preface The first part introduces you to OpenRG, and provides guidelines for reading this book.

Part II: Getting Started The second part takes you up to the point where you have OpenRG running on your host. This consists of setting up your host computer and installing the necessary software to get you up and running. In addition, this part introduces Jungo's Technical Support, aimed at granting you better understanding and utilization of OpenRG.

Part III: Start Developing with OpenRG The third part enables you to start working with OpenRG. The part begins with a detailed Development Tutorial, providing you with a hands-on tool that deals with all major aspects of the development process. This part also provides detailed instructions on how to modify and extend OpenRG to run on your platform. You will learn how to tailor OpenRG to your board, how to integrate applications and device drivers with the OpenRG environment, how to modifying the Web-based management interface to your specific needs, and more. Detailed step-by-step instructions, guides and expert tips will accompany you throughout the various programming tasks.

Part IV: Advanced Development Concepts This part provides extensive explanations of the OpenRG architecture, configuration database and the Build mechanism. In addition, an entire chapter is dedicated to general debugging tactics, with special emphasis laid on networking and low-level debugging methods.

Part V: Features Knowledge Base This part provides an in-depth exploration of OpenRG's features, covering its ready to use modules, such as VoIP, Web-based management, etc.
1.6 Conventions Used in This Book

The following typographic conventions are used throughout this book:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Italic</em></td>
<td>This typeface is used when introducing a new term, and for highlighting words with special contexts.</td>
</tr>
<tr>
<td><strong>Bold</strong></td>
<td>This typeface is used for file and path names, and for highlighting words with special contexts.</td>
</tr>
<tr>
<td><strong>Typewriter</strong></td>
<td>This typeface is used for function and command names.</td>
</tr>
<tr>
<td><strong>URLs</strong></td>
<td>URLs appear in blue font, and are clickable.</td>
</tr>
</tbody>
</table>

Code is quoted in boxes with a highlighted background.

Notes hold important, emphasized information that is directly relevant to the surrounding text.

Procedures described in this book are performed using the Linux shell prompt in two user modes:

- Regular user – denoted by the `$` symbol
- Root user (or superuser) – denoted by the `#` symbol

1.7 Tell Us What You Think

Information in this book has been tested and verified to the best of our ability. Please let us know if you find any errors, as well as if you would like to express your suggestions for future editions, by writing to **rg_support@jungo.com**

For more information about Jungo Ltd., visit our Web site at: **http://www.jungo.com**
2
System Setup

2.1 Software and Hardware Requirements

Developing with OpenRG requires the following setup:

- A Residential Gateway or development platform

- A host PC running one of the following versions of the Linux operating system:
  - Debian 3.1 or higher
  - Mandrake 8.1 or higher
  - Redhat 7.3 or higher
  - Ubuntu

- A text editor that supports tags, such as VI or Emacs.

- The following minimum hardware requirements for the host PC:
  - 1.6GHz CPU
  - 256MB RAM
  - 1.5GB free disk space
  - 2 network cards
Internet access, unless the installation is performed using a CD

2.2 Improving the Efficiency of Your Development Process

- Use a text editor (e.g. VI or Emacs), which provides a tagging mechanism that helps you navigate through the code. A sample `.ctags` file, which already contains OpenRG-specific navigation commands, can be downloaded from http://www.jungo.com/openrg/download/ctags_file. In order to install this file, execute the following:

  1. Download the `ctags` file.

  2. Rename the file and save it in your home directory:
     ```
     $ cp ctags_file $HOME/.ctags
     ```

- Create a ctags database, which will enable you to quickly locate functions and variables in OpenRG's vast code. To do so, go to the root directory of your RG tree, and run following command:

  ```
  $ ctags -R --links=no
  ```

Creation of the ctags database is a time-consuming process. If you would like to shorten the database creation time, exclude irrelevant directories from the indexing process, as shown in the following example:

  ```
  $ ctags -R --links=no --exclude=*/pkg/kos/koslet.c --exclude=*stubs* --exclude=build/*/ --exclude=pkg/doc/* --exclude=pkg/gdb/* --exclude=pkg/kaffe/* --exclude=pkg/gcc/* --exclude=pkg/autotest/* --exclude=tags --exclude=cscope.out --exclude=*.oa --exclude=*.txt --exclude=*.htm* --exclude=*.xml* --exclude=*.diff --exclude=*.tmp --exclude=*.swp
  ```

After the database file is created, test it by searching for any function/variable used in OpenRG's source files. For example, to search for function `tc_get_stats`, run the following command:

  ```
  $ gvim -t tc_get_stats
  ```

Gvim opens `pkg/qos/main/tc.c`, pointing to the line where the `tc_get_stats` function is defined.

- Install the `ccache` application.

- Use a version control mechanism during your development cycle. Jungo uses the Concurrent Version System (CVS) to manage software project versions. Information on how to work with CVS is provided in Chapter 43.
3

The OpenRG SDK

The OpenRG Software Development Kit (SDK) is a downloadable toolkit that enables you to build and customize OpenRG, in order to create a production ready image.

3.1 SDK Generated Content

Contact your sales representative (at http://www.jungo.com/openrg/contact.html) for the OpenRG evaluation SDK. Moreover, a commercial (licensed) version can be provided by Jungo. A licensed SDK includes a license file, which contains both a list of enabled features and a list of source and binary packages. This license is extracted with the JPKG application, which loads the OpenRG development tree to your host. Once your tree is loaded and compiled, you can obtain the following:

- Images – the following product images are created during OpenRG’s build process:

  openrg.img  An OpenRG product image, loadable via the Command Line Interface (CLI).

  openrg.rmt  An OpenRG product image, loadable via the 'OpenRG Firmware Upgrade' screen of the WBM (refer to Chapter 24). For more information, refer to the 'Firmware Upgrade' section of the OpenRG Administrator Manual.

  flash.img   A full image of the Flash contents including Bootloader, factory settings and OpenRG image (refer to Section 8.1.1).

  rgloader.img A bootloader application that loads the operating system into the Flash during the boot sequence (refer to Section 3.2.3).

- Environments – the following environments are available:
The OpenRG SDK

**Tools**  A toolchain consisting of utilities required to compile OpenRG. The install script copies the toolchain to the `/usr/local/openrg/` directory.

**System Utilities**  Useful utilities for working with various OpenRG components (refer to Section 3.6).

**USB RNDIS**  The USB drivers (INF files) required for Windows platforms. This directory is provided only in distributions that include this feature. You can download OpenRG USB RNDIS from [http://www.jungo.com/openrg/download/openrg_usb_rndis.tgz](http://www.jungo.com/openrg/download/openrg_usb_rndis.tgz).


### 3.2 Installation

#### 3.2.1 Quick Installation

Execute the following commands in order to install OpenRG from the SDK package:

```
$ tar xvzf rg_sdk_install.tgz
$ cd rg_sdk_install
$ ./install.sh
```

Refer to `compilation_readme.txt` for details explaining how to build your distribution. This file will appear in your chosen root directory, specified during the installation process.

#### 3.2.2 The Full Installation Procedure

For a detailed description of the installation and compilation procedures, download the Installation Guide for your distribution from [http://www.jungo.com/openrg/install.html](http://www.jungo.com/openrg/install.html). The following installation guides are available:

- Software Development Kit (SDK)
- User-Mode Linux (UML)
- Intel IXP425-Based Platforms
- Broadcom Platforms
- Freescale (PPC) Platforms
- Mindspeed Platform
- Ikanos Platform
- Infineon Platforms
• Cavium Platform

• Centillium Platform

3.2.3 RGLoader

The compilation of your distribution produces the OpenRG Bootloader, `rgloader.img`. The appropriate Installation Guide for each distribution that requires use of RGLoader, explains the burning procedure. RGLoader is platform-specific and may not appear in all distributions. If your distribution does not include an RGLoader, refer to the board's Installation Guide for information on how to use the manufacturer's bootloader.

3.2.3.1 Using RGLoader

You can use RGLoader to:

1. Burn an image. You can use RGLoader to burn a new image in two variations:
   • Burning a full Flash image
   • Burning the image and Factory Settings
2. Update to a new RGLoader version

3.2.3.2 Compiling RGLoader

1. Change directory to where you installed the OpenRG development tree, for example:
   `$ cd ~/rg-5.5.0/rg`
2. Execute the following command:
   `$ make config DIST=RGLOADER_<platform> && make`

   The compilation process will produce an image ready for burning, located under `build.RGLOADER_<platform-specific distribution>/os/rgloader.img`.

3.3 The OpenRG License

3.3.1 The License File

The OpenRG license file, provided by Jungo, activates OpenRG as an evaluation or commercial product. This file is usually named `jpkg_<arch>.lic`, and contains the following components:

• License string – a string that encrypts the entire license file information.
• **Version** – the version number.

  VERSION: 5.5.0

• **Features** – an abbreviation list of all the product features included in the license.

  features:  
  toolchain: Development Environment  
  rg_foundation: Foundation Package (Mandatory)  
  usb_rndis: LAN Interface: USB 1.1 Slave, RNDIS 5.0  
  usb_host: LAN Interface: USB 1.1/2.0 Host  
  80211: LAN Interface: Wireless LAN, 802.11a/b/g

• **src/bin packages** – an abbreviation list of the included source and binary packages.

  jpkg src: base  
  atm  
  build  
  busybox  
  e2fsprogs  
  freeswan  
  .  
  jpkg bin: 802.1x  
  acl  
  atm-rg  
  attr  
  auto-conf  
  backup  
  .

• **Limitations** – if in evaluation mode, the limitations of a non-commercial image are listed.

  uptime: 24  
  evaluation: yes  
  days from installation: 30

• **Hardware and software specifications.**

  target OSs: linux  
  msg type: non-dist  
  jpkg arch: armv5b

• **Package list** – a list of all the packages along with the passwords required to extract them.

  JPKG: rg-voip-ossip-gpl_4.8.3_src.ejpkg a56d25ab14ce2da  
  JPKG: rg-vendor-jstream-gpl_4.8.3_src.ejpkg 15f4eb2b76eaf6b  
  JPKG: rg-802.1x_4.8.3_armv5b.ejpkg 0a25ac29fb710d2  
  JPKG: rg-acl_4.8.3_armv5b.ejpkg e2a79c079784ca60

The **jpkg** application, which is fetched and installed on your host during the installation process, extracts the license and loads the OpenRG development tree to your host.

### 3.3.2 Evaluation vs. Commercial Modes

OpenRG is available in evaluation and commercial modes. Experiencing OpenRG in evaluation mode enables prospective customers to appraise the full features of OpenRG, and assess integration feasibility, performance, security and mass-production issues before purchase. Evaluation mode imposes certain limitations on the behavior of the product. These limitations do not detract from the evaluation potential in any way, but are put in place to differentiate between the evaluation and commercial products. The limitations placed on the evaluation version of OpenRG are:
• A license agreement on the main WBM login page.

• An 'Evaluation' notice at the top of WBM pages.

• A 24 hour limitation period on image functionality.

• A 30 day limitation period on development environment functionality.

The evaluation version of the SDK, as well as pre-compiled images, can be provided by your sales representative, whom can be contacted at http://www.jungo.com/openrg/contact.html. The license file is required to lift these limitations and activate OpenRG as a commercial product.

3.3.3 Troubleshooting

3.3.3.1 OpenRG halts at start, notifying that there is no suitable license.

OpenRG is configured as commercial, and the image cannot access a suitable license. Possible reasons:

• The current installed license does not allow usage of the modules configured into OpenRG. In this case, an extended license is required, contact openrg@jungo.com.

3.3.3.2 Is there a way to "activate" an existing OpenRG evaluation image?

No, OpenRG licenses are compiled into the image, there is no way to activate an evaluation image or replace a license of an existing image. In order to create the same feature set commercial image:

• The distribution tree used to compile the evaluation image should be configured to produce commercial images. Refer to the "Installing License" section for more information.

• A satisfying license should exist on the compiling host.

• The distribution tree should be rebuilt.

3.4 Development Tree

The OpenRG development tree contains all of the files needed to compile an OpenRG image. The tree contains three main directories: os, pkg and vendor.
Figure 3.1 OpenRG Development Tree

**os** Contains the OpenRG Linux-based kernel. The directory structure within **os** is the standard Linux directory structure. If your distribution is based on Linux 2.4, the directory name and location is `/os/linux-2.4`.

**pkg** Includes all of the OpenRG usermode feature implementation and specific kernel modules.

**vendor** Contains vendor-specific files needed to support OpenRG on a given platform. The files may be either vendor-supplied, written by Jungo or both. These files may come from the chip vendor as well as from the board builder, who use the chip to build their own boards.

This directory structure is modular; developers can add directories with new code wherever they desire, and integrate them with the OpenRG system.

### 3.5 Toolchain

The toolchain consists of the following tools that are used during the compilation:

**gcc** is a C compiler.

**ld** combines a number of object and archive files, relocates their data and ties up symbol references.

**as** is primarily intended to assemble the output of the C compiler for use by the linker.

**nm** lists object files symbols.
**objcopy** copies the contents of an object file to another.

**objdump** displays information about one or more object files.

**readelf** displays information about one or more ELF format object files.

**size** lists the section sizes for each of the object or archive files in its argument list.

**strip** discards all symbols from object files objfile.

**gdb** a debugger that is used to find errors. It allows you to stop a program at any point and examine and change the values of variables.

During the OpenRG installation, the toolchain is placed in `usr/local/openrg`. For your convenience, each of these tools (except for **size** and **readelf**) has a corresponding shortcut (`j<tool_name>`). Without leaving the OpenRG development tree, this shortcut enables you to call its respective tool from the toolchain that corresponds to the currently used OpenRG distribution. These shortcuts are bash scripts located in `usr/local/bin`. The following example demonstrates calling the **objdump** tool by its shortcut from a configured RG tree:

```
$ cd /home/<user>/rg/
$ make config DIST=<distribution_name>
$ cd pkg/dns /* or any other subdirectory of the RG tree, source or build */
$ jobjdump /* this will run /usr/local/openrg/1386-jungo-linux-gnu/bin
    * /1386-jungo-linux-gnu-objdump */
```

### 3.6 System Utilities

The following are system utilities, which you may find useful while working with various OpenRG components.

<table>
<thead>
<tr>
<th>Utility</th>
<th>Usage</th>
<th>Location</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>build_flash</td>
<td>Builds OpenRG's Flash image</td>
<td>rg/build/pkg/perm_storage</td>
<td>Section 8.1.2</td>
</tr>
<tr>
<td>rg_conf_inflate</td>
<td>Converts a binary configuration dump file to an rg_conf text file</td>
<td>rg/build/pkg/tools</td>
<td>Section 18.9</td>
</tr>
<tr>
<td>crash_analyze.pl</td>
<td>Analyzes OpenRG's crash information via the serial console output</td>
<td>rg/pkg/tools</td>
<td>Section 20.9</td>
</tr>
<tr>
<td>gen_maint_patch.sh</td>
<td>Upgrades an OpenRG distribution (version 4.2 and later)</td>
<td>rg/pkg/tools</td>
<td>Section 43.6</td>
</tr>
<tr>
<td>upd_file_util</td>
<td>Creates a remote upgrade (.rmt) file, or adds a product header to an existing one</td>
<td>rg/build/pkg/rmt-update</td>
<td>Section 24.3</td>
</tr>
<tr>
<td>Tool</td>
<td>Description</td>
<td>Location</td>
<td>Section</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>jgcc</td>
<td>Calls the <code>gcc</code> compiler from a toolchain, which corresponds to the currently compiled OpenRG distribution</td>
<td>Installed in <code>usr/local/bin</code></td>
<td>3.5</td>
</tr>
<tr>
<td>jnm</td>
<td>Calls the <code>nm</code> tool from a toolchain, which corresponds to the currently compiled OpenRG distribution</td>
<td>Installed in <code>usr/local/bin</code></td>
<td>3.5</td>
</tr>
<tr>
<td>jobjcopy</td>
<td>Calls the <code>objcopy</code> tool from a toolchain, which corresponds to the currently compiled OpenRG distribution</td>
<td>Installed in <code>usr/local/bin</code></td>
<td>3.5</td>
</tr>
<tr>
<td>jobjdump</td>
<td>Calls the <code>objdump</code> tool from a toolchain, which corresponds to the currently compiled OpenRG distribution</td>
<td>Installed in <code>usr/local/bin</code></td>
<td>3.5</td>
</tr>
<tr>
<td>jstrip</td>
<td>Calls the <code>strip</code> tool from a toolchain, which corresponds to the currently compiled OpenRG distribution</td>
<td>Installed in <code>usr/local/bin</code></td>
<td>3.5</td>
</tr>
<tr>
<td>jgdb</td>
<td>Calls the <code>gdb</code> tool from a toolchain, which corresponds to the currently compiled OpenRG distribution</td>
<td>Installed in <code>usr/local/bin</code></td>
<td>3.5</td>
</tr>
</tbody>
</table>

**Table 3.1** System Utilities
4

Technical Support

4.1 The Support Team

Jungo's support team is consisted of carefully selected, highly trained professionals, who specialize in all aspects of OpenRG.

4.1.1 Role

The support team's role is to provide you with a focal point within Jungo, responsible for:

• Answering general and configuration questions
• Reproducing issues that you report
• Providing code Hotfixes
• Acting as a feedback channel for the development of new features
• Distributing maintenance and upgrade version releases

4.1.2 Structure

The support group is structured of three tiers:

1st Tier  Responsible for answering technical questions and resolving configuration problems, as well as reproducing reported problems.

2nd Tier  Responsible for answering technical questions and providing Hotfixes in order to achieve quick problem resolutions (between 1–3 days).
3rd Tier  Responsible for conducting thorough investigations for time consuming problems, and for providing Hotfixes for complex issues (over 3 days).

4.1.3 Contacting Jungo's Support

Contacting the support group is mostly recommended through the Support Center, which is a secure Web-based interface for reporting problems and questions. The support center advantages include:

- A secure password protected channel
- An e-mail notification mechanism to your pre-defined list, on any information or update submitted. Note that this mailing list also contains the e-mail addresses of the relevant marketing and R&D contacts
- A focal point for an issue, eliminating loss of information sent through other channels
- Provides a database for tracking and querying previously reported issues
- Presents account information details such as SLA package, sales representative, Hotfix counter and more

4.2 Service Level Agreement

The Service Level Agreement (SLA) packages are defined according to the availability of upgrade and maintenance releases, Hotfix coverage and response time. The different available packages allow you to select an SLA package that best suits your technical requirements. A detailed description of the available SLAs can be found at: http://www.jungo.com/openrg/download/sla.pdf.

4.3 Support Account Manager

Jungo assigns a Support Account Manager within the support team, to customers who license Gold or a higher SLA package, as well as to customers who had hired Professional Services. The Support Account Manager's role is to be your focal point within the support team, for the actual deployment period of OpenRG. The account manager monitors the issues that you report and assures that requests, answers and fixes are provided in a timely manner by the support team.

4.4 Issue Life Cycle

The following diagram depicts the different possible states in an issue's life cycle.
Figure 4.1  Issue Life Cycle

NEW
When a support issue is opened, its initial state is NEW. It is left in this state until Jungo's support engineer first handles it.

ASSIGNED
This is the state of the issue from the moment a support engineer first handles it, until it is resolved. During this time, the issue has a 'PENDING' field. The 'PENDING' field states what event should happen in order to progress towards a resolution. The possible pending events are:

- **Jungo's Response** A support engineer is working on the issue, and will provide an update when more information is available.

- **Customer's Response** Jungo is waiting for additional information from the customer—i.e., responses to Jungo's clarification requests, configuration files, etc.—in order to better investigate the issue.

When waiting for 'Customer's Response', a combo box marked 'Pending' appears at the bottom of the page. This combo box allows customers to decide whether to return the issue to Jungo (in case all the required information has been provided), or leave the issue pending (if they have to provide additional information).

- **Jungo's Implementation** Jungo has started implementation according to the issue. This may be fixing a bug, implementing a new feature, editing a document, etc.

RESOLVED
Once the issue is resolved, it is changed to the RESOLVED state. Resolving an issue means it is considered closed. However, sometimes there may be more procedural processes involved. When an issue is resolved, it has a 'RESOLUTION' field. This field may be one of the following:

- **FIXED** The issue is a bug or an enhancement and the required changes have been done.

- **Answered** The issue is a question, and it has been answered in full.
• **INVALID** The issue has been found to be invalid. For example, when the customer has been guided to a resolution without involving any code modifications or configuration changes.

• **WORKSFORME** Both Jungo engineers and the customer could not reproduce the bug. It seems that the problem has only occurred once for an unknown reason and does not recur.

• **WONTFIX** This state is extremely rare, however, in certain cases, Jungo decides not to handle the problem or implement the feature.

• **DUPLICATE** The issue is a duplicate of a previously opened issue. The customer has already opened an issue for the very same problem or request. In this case, one of the two (usually the latter), will be resolved as 'DUPLICATE' and the process will continue from the former issue.

The customer can change the state to RESOLVED with all resolution fields, except for DUPLICATE. However, this is not possible when the issue is pending Jungo's implementation. In most cases, Jungo is the one that changes the issue to RESOLVED. In cases that involve implementation, the issue is changed to RESOLVED when the implementation is completed. In this case, the issue is pending new delivery. Once the fix/implementation is delivered to the customer, it is pending the customer's verification. In this case, the customer may change the state to either VERIFIED or REOPENED. Issues that are in the RESOLVED state for more than 8 weeks, will be closed.

**REOPENED**
This state is similar to NEW, except it is given to an issue that has already been resolved. If the customer does not accept the resolution, the issue is changed to REOPENED. During the time Jungo handles the issue, its state is changed to ASSIGNED, and the issue will move through the cycle detailed above.

**VERIFIED**
When the customer changes an issue to this state, it means the resolution has been accepted, tested and works. From this state, Jungo will move the issue to the CLOSED state.

**CLOSED**
When the issue is closed, it means that the problem has been completely resolved. If, in the future, the customer encounters the problem again, the issue should be changed to REOPENED.

### 4.5 Using the Support Center

The Support Center enables you to report software problems and to raise issues that require a support team response. After logging in to the Support Center, you will be able to:

• View the list of issues sorted by their state–NEW, ASSIGNED, VERIFIED, etc.

• Query previously reported issues

• View the details of a specific issue
• View an up-to-date account report specifying your SLA, provided deliveries, Hotfixes tracking and more

4.5.1 Home Page

Once logged in to the Support Center with the user name and password provided by Jungo, the home page is displayed. This page presents:

• Any reported issues that are not yet in CLOSED state

• Allows performing queries on previously posted issues

• Contains an 'Issue Writing Guidelines' link, which assists with reporting an issue (refer to Section 4.5.3)

• Contains an 'OpenRG Documents' link, which leads you to OpenRG's documentation, sorted by version, at Jungo's Web site

• Contains an 'Account Information' link, which presents the following information:
  • SLA package details
  • A 'Show Contacts Information' link, which presents the list of contacts on file in Jungo, as well as your focal point contact
  • A 'Hotfix Report' section, which presents how many Hotfixes were provided and their status within Jungo (Adopted, Open, Scheduled, Rejected)
  • A 'Distributions Tracking' section, which presents the OpenRG versions provided and the issue number with which the distribution can be found

4.5.2 Considerations Before Opening an Issue

The following should be considered before submitting an issue:

• Does the issue occur on Jungo's original OpenRG image?

• Can the issue be reproduced on Jungo's reference platform?

• An elaborated scenario by which the issue can be reproduced

• Capture any information that might assist in investigating and debugging

• Search Jungo's User Manual for instructions on how to reach a desired configuration

• Search Jungo's Programmer's Guide for instructions on how to add and manipulate functions
### 4.5.3 Reporting an Issue

An issue can be categorized into the following types:

- 'Howto' questions, requesting guidance for configuration and setups
- Clarification questions regarding OpenRG's functionality
- Reporting of issues, such as a crash observation, performance, behavioral anomaly, specification and RFC compliance, etc.
- Programming and source code related questions
- Enhancement proposal or request

To report and submit an issue to the support team, perform the following:

1. Click the 'New' link in your Support Center home page. Every field link in the 'Enter Issue' page, such as 'Version', 'Distribution', etc, leads to an explanation on that field.

2. In the 'Summary' field, enter a short title for the issue. Keep this title concise for easy tracking when queries are performed on the issues' database.

3. In the 'Version' drop-down menu, select OpenRG's version on which the issue was found. To find out the version for an issue, perform one of the following:
   - In OpenRG's Web-based Management, click the 'System' tab. The 'Overview' screen displays the software version
   - Connect to the gateway using Telnet or a serial cable. Type "ver" (in versions prior to 4.3) or "system ver" (in versions 4.3 and up)

4. Select your distribution (platform) in the 'Distribution' drop-down menu.

   Note: If your version, distribution, or any other field is not available in the drop-down menu, add a remark in the 'Description' field in order to notify the support team, which will add it for future use.

5. Select the priority for the issue in the 'Priority' drop-down menu. This field refers to the importance and order in which the issue should be handled by Jungo.

6. Select the severity for the issue in the 'Severity' drop-down menu. This field refers to the impact of the issue on OpenRG's functionality. For example:

   **LOW Severity P1 Priority Scenario** You could not find documentation for a specific setup. The severity could be set to LOW, but the priority should be set to P1, as this is preventing you from achieving the desired setup.
**HIGH Severity P5 Priority Scenario**  While running general QA testing on OpenRG, you encountered a crash in a component that will not be used in your project. The crash should be logged as HIGH severity, while the priority on your behalf would be P5—the least important.

7. In the 'Description' field, describe the issue elaborately. In order to reply to an issue promptly and accurately, we urge you to use the template guidelines, found within the 'Description' field. Following these guidelines will ensure that we receive all the details required to understand, reproduce and analyze the reported issue with minimum back and forth correspondence, thus preventing any delay in resolution.

### 4.5.4 Using Issue Queries

The 'Query' link in your Support Center home page enables you to search through the issues' database. You can follow up on an issue's status, or investigate whether an issue has already been reported. You can run simple or complex queries to track and locate previously reported issues. For your convenience, we have prepared several queries that may assist you in querying the database. These are: 'All issues', 'All unresolved issues', 'All closed issues', and 'All unclosed issues'.

### 4.5.5 Automatic Closure of Issues

The Jungo Support Center will automatically close issues, to which no response had been received, after about three months. Once an issue is resolved, or if more information is required for further investigation, the support team will send three reminders (over the duration of three months). If no reply is received, whether it is for verification or for more information, the support team will assume no further handling is required and close the issue automatically.

### 4.6 Remote Debugging

This section describes the setup and process of remotely connecting your Linux computer, referred to as the "remote computer", to an "assistant computer" in Jungo's lab, in order to either detect or resolve a problem that could not be identified in Jungo. There are many techniques that you can use to establish a remote connection between one computer and another, some of which are described herein. Select the one that best fits your current needs. All techniques are based on SSH. Note that some require non-trivial software that must be downloaded from the Internet. The following figure depicts the remote connection's topology.
4.6.1 Remote Computer Requirements

In order to efficiently assist in resolving a problem, the remote computer's minimum requirements are:

- Debian Linux
- A 2 GHz processor
- 512 KB of RAM
- Two network interfaces–one for the LAN and another for the WAN
- A serial cable for direct communication with OpenRG

Also required is a serial power relay device, enabling the remote assistant to perform a 'hard' reboot when needed. Such hardware can be evaluated and obtained using this link: http://www.pencomdesign.com/2ch_relay.htm. For more information, refer to Section 4.6.4.

You must also verify that the SSH server is configured to allow incoming requests, and that a TFTP server is running in order to allow image modifications on the board. This can be done by typing:

```
netstat -l | grep SSH
```

You can type the same for a TFTP server. Depending on your Linux distribution, you can either use 'chkconfig', or manually update your `/etc/init.d/` directory for activate-during-startup tweaks.

4.6.2 Secure Shell (SSH)

SSH is one of the best remote login client-server utilities, and it is supplied with all Linux distributions. SSH can be used for a terminal console session just like Telnet, and can be configured to transfer X-Windows applications as well. You can remotely connect to a
computer and watch an application running on the machine, but not the entire desktop. The remote assistant will use VNC (Virtual Network Computing) tunneling over SSH, to ensure that remote access to your computer is secured.

Install 'x11vnc server' (http://www.karlrunge.com/x11vnc/) to allow remote access to your computer. In order to allow tunneling through SSH, you must add a new user to your computer that will have the permissions you want for a remote user. The remote assistant will connect to your computer using 'vncviewer' through SSH. This connection is safe for several reasons:

- Your computer is password protected for privileged users only through SSH.
- All access to your desktop is done using VNC tunneling through SSH. This means that Jungo is only accessing port 22 on your computer, and not the unprotected port 5900 that VNC uses. It is recommended that you block all incoming traffic on port 22.
- Using x11vnc, it is possible to allow connections only from localhost, and setup passwords for VNC connections. Refer to 'man x11vnc' for more details.

In order to limit the remote assistant to port 22 on your machine, refer to Section 4.6.5, which provides a script that blocks all traffic unless its designated port is 22. To summarize, it is required that you have x11vnc installed on your computer, and have an SSH server running. It is recommended to block all incoming traffic, excluding traffic designated to port 22.

### 4.6.3 Using DNS and a Redirection Service

This section describes how to map a static/dynamic IP address or a long URL to an easy-to-remember subdomain, in order to have your host remotely accessed without giving away its IP address. The following example describes the service provided by http://www.no-ip.com, but any other service provider can be used. Perform the following:

2. Go to "Sign Up" and complete the registration form, and activate your account by clicking the URL that was sent to you by email.
3. Go to the "Downloads" section. Download and install the DUC client.
4. Install and run your client.
5. At this point you must add a host name, follow the software link to do so.
6. Click refresh and check your new host.

Your computer now has a permanent name as long as the DUC client is activated and your host name is checked. Your computer should be accessible now via that host name. Note that you can configure the DUC client to be a system service or to set it up to run during startup.
4.6.4 Setting Up a Serial Power Relay Device

A serial power relay device that connects to OpenRG and to your computer, enables power management commands (i.e. turn on/off) via the CLI terminal. Several devices can be purchased according to your needs. 1-channel, 2-channels and 8-channels are currently available by Pencom Design (http://www.pencomdesign.com). Each channel can control a single OpenRG board. The relay device should connect to your wall power outlet and to OpenRG in the following topology:

![Power Relay Device Connection Diagram](image)

Figure 4.3 Power Relay Device Connection

The numbered connections in this figure are as follow:

1. This is the power source for the relay board. It uses a regular 9v adapter that comes with your relay board.

2. This is a regular RS232 connection to your computer terminal. It enables the PC to control the power relay, thus enabling a 'hard reset' on your board remotely.

3. There are two outlets connected to the exact same location on the relay board. One of them is connected to a 'live' power source, and the other is a 'dummy' connected to your OpenRG board.

4. The 'dummy' outlet is not connected to a live power source, but to an outlet that is controlled by the relay board.

By issuing a command through the terminal, you close a circuit between the 'live' outlet and the 'dummy' outlet. Note that this connection is not trivial. You must open the relay board box and connect both outlets to the same connection. For more details, refer to the Pencom Design documentation, located at: http://www.pencomdesign.com/files/8ch_relay_manual.pdf.

As previously mentioned, there are several relay boards available. The scenario described above is general, and is not detailed enough for proper activation of your board using the relay
remotely. There are specific jumper configurations that should be taken into consideration when connecting the relay boards. Each board has a different jumper set, depending on its number of channels. Contact your power relay board manufacturer for the full configuration schemes and for more information.

4.6.5 Protecting the Remote Computer

Copy and paste the following script, save it (for example, as `allow_ssh.sh`) and make sure its permissions are set to 'executable'. Then run it.

```bash
#!/bin/sh

# This script is used to enable your computer with incoming traffic through port 22 only, to allow remote SSH sessions.
# This ensures your computer stays safe allowing access only to users that you have approved.

# Arguments:
# $SERVER_IP - your WAN server IP address.

#My system IP/set IP address of server
SERVER_IP="X.X.X.X"

# Flushing all rules
iptables -F
iptables -X

# Setting default filter policy
iptables -P INPUT DROP
iptables -P OUTPUT DROP
iptables -P FORWARD DROP

# Allowing unlimited traffic on loopback
iptables -A INPUT -i lo -j ACCEPT
iptables -A OUTPUT -o lo -j ACCEPT

# Allowing incoming ssh only
iptables -A INPUT -p tcp -s 0/0 -d $SERVER_IP --sport 513:65535 --dport 22 --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -p tcp -s $SERVER_IP -d 0/0 --sport 22 --dport 513:65535 --state ESTABLISHED -j ACCEPT

# Ensuring nothing comes or goes out of this box
iptables -A INPUT -j DROP
iptables -A OUTPUT -j DROP

# Use this to return to a normal state
iptables -X
iptables -P
iptables -P INPUT ACCEPT
iptables -P OUTPUT ACCEPT
```

4.7 Feedback

Jungo’s support team is on a constant lookout for improving the service we provide. Better service equals to customer satisfaction. We urge you to send us your feedback, either by e-mail or phone, in order to convey the points where we can improve our service to you. For our contact information, refer to Chapter 47.
Part II  Developing with OpenRG
Table of Contents

5. Development Tutorial .......................................................................................................... 30
6. Board Tailoring .................................................................................................................. 74
7. Porting OpenRG to a Different Architecture ............................................................... 86
8. Preparing for Production ................................................................................................. 101
9. Software Configuration .................................................................................................. 115
10. Tailoring the Web-based Management ....................................................................... 117
11. Tailoring the Role-based Permissions Engine ............................................................. 137
12. Integrating Applications ............................................................................................... 144
13. Integrating Kernel Modules .......................................................................................... 152
14. Integrating Device Drivers ............................................................................................ 155
5

Development Tutorial

5.1 Getting Started

OpenRG is a comprehensive development environment, addressing the unique requirements of software development for embedded network devices. The OpenRG development platform consists of a GNU-based development environment coupled with a rich set of foundation modules and utilities for the development of embedded networking devices. The OpenRG development platform offers developers powerful tools for building, testing, deploying and maintaining residential gateway applications, dramatically increasing developer efficiency and productivity.

The aim of this tutorial is to provide you, the developer, with a hands-on tool that will hopefully serve as a lucid description of how to get things done with OpenRG. All major aspects of the development process are dealt with, including the extension of the OpenRG configuration database, the construction of Web-based management pages, the integration of applications, and more low-level aspects such as kernel modules integration. The tutorial is comprised of stages, each dealing with a separate topic. It is advised to set aside approximately 6-8 hours to thoroughly go through all of the tutorials stages. The tutorial consists of the following stages:

- Stage 1: Adding a new package to OpenRG
- Stage 2: Extending the OpenRG configuration database
- Stage 3: Creating Web-based management pages
- Stage 4: Integrating the application
- Stage 5: Integrating kernel modules
The tutorial stages include references to corresponding 'in-depth' chapters in the Programmer's Guide, which elaborate on the topics discussed, offering a more comprehensive guide and reference. These chapters are designed to serve the developer when confronted with advanced programming tasks that are not within the scope of the tutorial. When getting to know OpenRG for the first time, it is suggested that you go through the tutorial, step-by-step, and perform all of the programming tasks described within. This will give you a sound foundation of knowledge that will later facilitate the 'real' development process.

5.2 The Programming Example

Working with OpenRG is relatively easy for any programmer with a working knowledge of C. The OpenRG development platform provides sample utilities and module creation wizards that make the development process easy and straightforward. The following diagram illustrates the basic stages required to integrate features within OpenRG.

**Figure 5.1 An Illustration of the Development Process**

It is important to understand the logic that underlies the development process. The following sections will elaborate further on these concepts, but it is beneficial to understand the following concepts:

1. The OpenRG database is extended to support the new application data.
2. The Web-based management interface is created using OpenRG's GUI libraries.
3. The application communicates with OpenRG's coordination logic using Local Management API (inter-process communication).
4. The kernel module is integrated using the development platform's standard interface.
5.2.1 Wag the *Watchdog*

The service that will be integrated during this tutorial is a *Watchdog*. A Watchdog monitors the state of an event and performs a predefined action according to a parameter provided by the user. In our case, the Watchdog will reboot the system if it does not receive a "keep-alive" signal from the application within a specified time frame. A flow chart of the Watchdog’s behavior is depicted in Figure 5.2.

![Figure 5.2 The OpenRG Watchdog](image)

5.2.2 Watchdog Components

Since our Watchdog is similar in design to any service you may wish to integrate with OpenRG, it is comprised of the following components:

**Configuration Database** The Watchdog’s configuration is stored in OpenRG’s configuration database, `rg_conf`. The database is extended to store the Watchdog application entries: Enabled, Keep-alive signal margin and Boot message. Command line instructions are introduced to manage the Watchdog application configuration from OpenRG’s Command Line Interface (CLI) processor.

**Web-based Management** A WBM page is developed to allow the user to configure the Watchdog’s parameters.

**Application** The Watchdog application samples OpenRG’s Main Task periodically (every 'margin/2' seconds), to ensure that OpenRG is still running. In this case it signals the kernel module with a "keep-alive" message. In addition, the Watchdog checks whether its configuration file entries have been changed, and reconfigures itself accordingly.

**Kernel Module** Finally, the Watchdog kernel module uses OpenRG’s general character device to enable communication from the Watchdog application. The module provides IOCTL
functions to configure the margin and boot message, as well as to handle the "keep-alive" signal. The module will perform a reboot in case the "keep-alive" signal fails to arrive within a specified time frame.

5.2.3 Installation and Build

1. In order to start working, you must first install your development environment. To learn how to do so, please refer to Section 3.2.

2. Remove the Watchdog feature by remarking the tutorial CONFIG flag you have added in `rg/pkg/build/dist_config.c`.

3. Reconfigure and compile the entire development tree from the top of the tree.

   ```
   $ make distclean
   $ make config DIST=<distribution name> LIC=<license file path>
   $ make
   ```

4. Load the newly created image using OpenRG's Firmware Upgrade utility from 'Maintenance' menu item of the 'System' tab.

5. After the image is up and running, restore the factory defaults using the respective link in the same location (System -> Maintenance).

5.2.4 Configuring OpenRG – See the Watchdog Run

In this tutorial you are about to add a feature to OpenRG. Before you start working on the tutorial, make sure that you are familiar with the OpenRG system architecture. As described earlier, adding the feature involves several stages, during which quite a lot of material is presented. Since progressing through the tutorial while having a clear vision of the final outcome can better the educational process, we will present you with a sneak preview of what is to come.

The development tree contains the entire tutorial example, including configuration database, Web-based management pages, external process and kernel module source codes. Moreover, it can easily be enabled or disabled. This section will show you how to enable and disable the tutorial example, compile and burn the image on your development board. In the subsequent tutorial stages, you will be required to create the example from scratch.

1. First, take a look at OpenRG's 'Services' management screen. Once your development environment is up and running, point your browser to the following address: 192.168.1.1. For detailed information on using OpenRG's Web-based management, refer to the accompanying User Manual.

2. Click the 'Advanced' menu item. Notice that it contains numerous links.

3. Look for the 'Watchdog' link.
Do not be concerned if you cannot locate the link, it is not visible since the Watchdog feature does not exist yet. The remainder of this section will describe how to add the feature, and make it accessible via the 'Services' management screen.

To add the Watchdog feature to the OpenRG image you are about to compile, perform the following:

1. Go to the following directory: \texttt{rg/pkg/build}.

2. Open the file \texttt{config_opt.c}.

3. Search the file for the string "TUTORIAL".

4. You will find the following line:

   \begin{verbatim}
   { "CONFIG_RG_TUTORIAL", NULL, OPT_NOSTRIP | OPT_HC }
   \end{verbatim}

   This line appears in the \texttt{rg_config_options} array, among other feature definitions. What should interest you, is the configuration flag name that serves as the first parameter in the entry. The entries following the flag are internal to OpenRG and should not be altered.

5. Add the \texttt{CONFIG_RG_TUTORIAL} flag to the tree configuration command.

   \begin{verbatim}
   $ cd ../..
   $ make distclean
   $ make config DIST=<distribution name> LIC=<license file path> \ CONFIG_RG_TUTORIAL=y
   $ make
   \end{verbatim}

6. A remote upgrade image will be created—\texttt{build/openrg.rmt}. The build results are placed in the \texttt{build.<distribution name>} directory. The \texttt{build} directory is actually a symbolic link, automatically created by the 'make config', which points to the last distribution build directory. Use the Web-based management to upgrade the image to your board, which will reboot after the upgrade. For detailed instructions on performing a firmware upgrade using OpenRG's Web-based management, refer to the 'Upgrading the Gateway's Firmware' section of the OpenRG Administrator Manual.

7. Click the 'Advanced' menu item under the 'Services' tab. You should see that a 'Watchdog Triggering' link has been added.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{watchdog.png}
\caption{Watchdog Triggering}
\end{figure}
8. When the image is up and running, restore the factory defaults from the 'Maintenance' menu item of the 'System' tab.

To test the Watchdog feature, in the 'Watchdog Triggering' screen, uncheck the 'Enabled' check box, set the signal margin to 30 seconds and click 'Apply'. The management console will continue to be active for the next 30 seconds, after which OpenRG will reboot and prompt you to log in again.

If your board is connected to your computer with a serial cable, you can use a serial console application such as Minicom in Linux, to view the progress log. This log includes the message you defined as the Ending Message and the rest of the boot sequence. The configuration option above provides an example of OpenRG’s feature management. For further information regarding feature management and configuration, refer to Chapter 9.

9. Now that you have seen the Watchdog run, reconfigure the tree without the CONFIG_RG_TUTORIAL flag, in order to start working.

```bash
$ cd ../..
$ make distclean
$ make config DIST=<distribution name> LIC=<license file path>
$ make
```

5.3  Stage 1 – Adding a New Package

New packages are usually added in new directories, by creating suitable Makefiles and integrating them to OpenRG’s existing Makefiles. The code to support the new functionality is then developed and integrated with OpenRG.

5.3.1 New Package Directory Structure

The Watchdog tutorial package will be developed in the watchdog directory, under rg/pkg. All Watchdog features will also be implemented under this directory. This includes the configuration database enhancement, Web-based management pages, the external process and the kernel module. For each of these components you will later on create the sub directories mgt, wbm, process and module, respectively, and integrate them with the Watchdog package.

```bash
~$ cd ~/rg/pkg
~/rg/pkg$ mkdir watchdog
```

Note: The tutorial example source code is located in the rg/pkg/samples/tutorial directory. However, files that you create when performing this tutorial must be saved under rg/pkg/watchdog. This directory includes the mgt, module, process and wbm directories, containing code relevant to the configuration database, kernel module, external process and management pages, respectively.

In addition to the code mentioned above, some of the code is already written as part of the development tree and provided in your image, but protected with a suitable ifdef or simple if statements. In order to activate these lines manually, you can simply remove these statements.
5.3.2 Working with MakeFiles

The Make methodology is an integral part of the OpenRG development environment. It includes a utility that automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them. To prepare for using make, you must write a file called Makefile that describes the relationships among files in your program and provides commands for updating each file. The make process of OpenRG is hierarchical according to its directory structure. The root directory of OpenRG contains the top level Makefile which is responsible to activate the Makefile of each of the sub-directories, os, pkg and vendor. For in-depth information on the OpenRG Makefile system, refer to Chapter 19.

In order to include the newly added package in OpenRG's build process, we need to modify the Makefile of the parent directory. In this case, we need to modify the Makefile in rg/pkg to build the new watchdog directory. To add the Watchdog directory, simply add the following to the Makefile before the already existing JMK_SUBDIRS+=\$(if $(CONFIG_RG_MAIN),main):

```
  JMK_SUBDIRS+=watchdog
```

Now that we have linked the new package to the OpenRG make system, we need to create the Makefile of the Watchdog package in the directory rg/pkg/watchdog. The purpose of this Makefile is to export this package's initialization API to OpenRG's initialization, compile the package initialization functions, and integrate all the package components into a single object file that is linked with OpenRG. Following is the initial Makefile of the Watchdog package:

```
rg/pkg/watchdog/Makefile
1   ifndef JMKE_ENV_INCLUDED
2     JMK_ROOT=../../
3     include $(JMK_ROOT)/jmk/env_root.mak
4   endif
5
6   JMK_O_TARGET=watchdog.o
7   JMK_O_OBJS=watchdog_init.o
8
9   JMK_EXPORT_HEADERS_DIR=watchdog
10  JMK_EXPORT_HEADERS+=watchdog_init.h
11
12  $(call JMKE_INCLUDE_RULES)
```

Here is a line-by-line explanation of the above Makefile:

**Line 2** JMK_ROOT is a variable that defines the OpenRG 'top of tree' in a relative way. Since the Watchdog directory is directly under the rg/pkg directory, we set JMK_ROOT to "././".

**Line 3** This line relies on the definition of the 'top of tree' to include the global environment settings file envir.mak. This Makefile configuration file defines global variables that are used in the make process. For example, the following make variables are set: tool chain, default compilation and link flags. This line should be the third line of every Makefile.

**Line 6** The JMK_O_TARGET variable defines the object target created by this Makefile. This object includes all the components of the Watchdog package and is linked with OpenRG's main executable.

**Line 7** The JMK_O_OBJS variable defines the object files that are combined together to create the target object file defined in the previous line. These object files are created from their
respective source file, in this case the `watchdog_init.o` file is created from the `watchdog_init.c` source file.

**Line 9** The JMK_EXPORT_HEADERS_DIR variable defines the package include directory where the package API will be installed. This directory is under the `rg/pkg/include` directory.

**Line 10** The JMK_EXPORT_HEADERS variable defines the list of include files that define the package API. These files are exported to the `rg/pkg/include/watchdog` directory.

**Line 12** Including `jmk/rg_root.mak`, in order to import all the compilation rules that are used by `make`.

Later on we will discuss adding and integrating the Watchdog components feature, and elaborate on how to add additional Makefiles to manage the sub directories of the Watchdog package as well. Once the suitable Makefile files are in place, every time you change a source file the `make` command, executed in the package directory, suffices to perform all necessary recompilations for the package.

```bash
~$ cd ~/rg/pkg/watchdog
~/rg/pkg/watchdog$ make
```

In order to build the OpenRG image with the new feature, we need to run the `make` command in the top of the OpenRG tree.

```bash
~$ cd ~/rg
~/rg$ make
```

If we need to recompile all objects, we could use the `clean` target to remove all the products of the `make`. We could clean a specific package or the whole OpenRG tree based on the directory we are in.

```bash
~$ cd ~/rg
~/rg$ make clean
```

A deeper clean which removes the results of the `config` is done with the `distclean` target, after which you cannot compile the tree unless you run `config` again, by executing `make config DIST=... LIC=...`. This will usually be done at the top of the OpenRG tree:

```bash
~$ cd ~/rg
~/rg$ make distclean
```

### 5.3.3 Initializing a New Package

The package initialization API and implementation are written in the `rg/pkg/watchdog` directory, in the files `watchdog_init.h` and `watchdog_init.c`. The Watchdog initialization is integrated with OpenRG's initialization process in `rg/pkg/main/mt_init.c`. Following is the API for Watchdog initialization:

```c
rg/pkg/watchdog/watchdog_init.h
```

```c
1 #ifdef __WATCHDOG_INIT_H_
2 #define __WATCHDOG_INIT_H_
3 4 void watchdog_init(void);
5 void watchdog_uninit(void);
6```
The following is a line-by-line explanation of the above API:

**Line 1-2,7** This is the standard multiple include protection mechanism. The include file will be processed only on the first time it is included.

**Line 4,5** These lines define the Watchdog initialization and un-initialization functions that will be invoked during OpenRG start-up and shutdown procedures, respectively.

For now, we will create an empty implementation of the Watchdog initialization as described below and in file `rg/pkg/watchdog/watchdog_init.c`. We include `main/mt_main.h` to get the API of OpenRG that will be used later on in the initialization of the package:

### `rg/pkg/watchdog/watchdog_init.c`

```c
#include <main/mt_main.h>

void watchdog_init(void) {
    rg_error(LCONSOLE, "Initializing Watchdog components.");
}

void watchdog_uninit(void) {
    rg_error(LCONSOLE, "Un-initializing Watchdog components.");
}
```

Finally, we need to link the Watchdog object to OpenRG's main process and invoke the initialization of the Watchdog from the OpenRG initialization process. The OpenRG main process is built in `rg/pkg/main`, and the Makefile in this package should be modified to link in the Watchdog object. Here is the line that should be added to this Makefile:

```makefile
JMK_L_OBJS_$(MT_TARGET)+=$(JMKE_BUILDDIR)/pkg/watchdog/watchdog.o
```

The `JMK_L_OBJS_$(MT_TARGET)` variable defines the list of objects that should be linked with OpenRG's main process. This line adds the Watchdog object file created in `rg/pkg/watchdog`. `JMKE_BUILDDIR` is used since this is an object file and all output files are created under the build directory. The OpenRG main process start up and shut down procedures are defined in `rg/pkg/main/mt_init.c`. In order to include the Watchdog `init` and `uninit` functions in this process, the following lines must be added to this file in the relevant sections:

### `rg/pkg/main/mt_init.c`

```c
#include <watchdog/watchdog_init.h>

void mt_init(void) {
    watchdog_init();
    ...
}

void mt_uninit(void) {
    watchdog_uninit();
    ...
}
```

The include file will introduce the Watchdog initialization API to OpenRG. The `mt_init` and `mt_uninit` functions, which start up and shut down OpenRG, will invoke the Watchdog
init and uninit functions respectively. Note that the function calls should be included inside the #ifdef __TARGET__ conditional compilation. If you want your new package to generate separate log messages from the rest of the components in the log file (rather than be logged as "Other"), perform the following. Add your package to the enum in pkg/include/log_entity_id.h:

```c
LOG_ENTITY_WATCHDOG = 400,
```

Then, add your package to the entities array in pkg/mgt/lib/log_entity.c:

```c
{ LOGENTITY_WATCHDOG, "watchdog", T_PTR(Tlog_entity_watchdog), LOG_USER,
  LOG_BUFFER_VARLOG },
```

LOG_ENTITY_WATCHDOG The package identifier.

**watchdog** Specifies the code for the new package. This argument will be used in the CLI to specify your package.

T_PTR(Tlog_entity_watchdog) The string that the user will see in the WBM for this package. This string (the T<name> variable) should be added to the pkg/mgt/lib/log_entity_lang.csv language file.

LOG_USER The facility that will be used for your package when syslog protocol log messages are sent to a remote syslog server. The most common facility is LOG_USER.

LOG_BUFFER_VARLOG The log buffer to which messages from your package are redirected. The common one is the "varlog" buffer (the default system logging buffer). Some entities, however, can be redirected to the security log.

Testing that the Watchdog and OpenRG are integrated is done by recompiling the OpenRG's tree and loading it to the board:

```
$ cd rg
$ make
```

OpenRG now includes the new (and empty) Watchdog package. The rg/build/openrg.rmt file can now be loaded to the target board, using the remote firmware upgrade procedure.

### 5.3.4 Lesson Summary

Going through the steps described above, you have learnt:

- How to create a new package and modify the **rg/pkg/Makefile** to include it with the tutorial feature.
- How to create Makefile for a new package.
- How to write and integrate the package initialization.
- How the build stages work.
- How to build your package.
For a complete reference regarding the files used in this stage, refer to this tutorial's appendix (Section 5.8).

## 5.4 Stage 2 – The Configuration Database

Every application that runs on OpenRG should be able to store and retrieve its own configuration data. OpenRG stores application data in the configuration database: `rg_conf`. The configuration database is consisted of volatile and persistent (non-volatile) sections. The volatile section is `rg_conf_ram`, while the persistent is simply `rg_conf`. This stage describes how to:

- Add a new management component to the Watchdog package, which will manage the data in the configuration database.
- Extend the configuration database with new entries, both volatile and persistent.
- Define and implement an API to store and retrieve the configuration database entries of the Watchdog application.
- Extend the Command Line Interface (CLI) to view and modify the values of the Watchdog configuration.

### 5.4.1 Adding the Watchdog Management Component

The Watchdog application requires adding new entries to the configuration database, and implementing an API to access these entries. We will implement this API in the `mgt` directory. To add this component to the Watchdog package, we need to create a sub-directory `mgt` under the `rg/pkg/watchdog` directory, write a new Makefile in that directory, and have the Watchdog Makefile enter this sub-directory during compilation.

- The first step is to create the directory:
  ```
  $ cd ~/rg/pkg/watchdog
  $ mkdir mgt
  ```

- The next step is to enhance the Watchdog Makefile so it enters the `mgt` sub-directory, and link the object from this directory to the Watchdog object. Following are the lines to be added. The first line adds the `mgt` directory to the JMK_SUBDIRS variable. The second line adds the `mgt` object file to the link of the Watchdog object.
  ```
  JMK_SUBDIRS+=mgt
  JMK_L_OBJS+=mgt/watchdog_mgt.o
  ```

- The last step is to add the Makefile to the `mgt` directory and develop the code.
  ```
  rg/pkg/watchdog/mgt/Makefile
  ```
This Makefile is very similar to the Makefile of the Watchdog package added earlier. The only differences are the names of the files that are created and exported.

**Line 1-4, 12** The standard header and footer of an OpenRG Makefile.

**Lines 6-7** Define that the API includes the files `watchdog_mgt.h` and `watchdog_mgt_init.h`, which will be exported to the `rg/pkg/include/watchdog/mgt` directory.

**Lines 9-10** Define that the object created here is `watchdog_mgt.o`, which is composed of the object files `mgt.o` and `commands.o`. These object files are created from their respective source files `mgt.c` and `commands.c`.

### 5.4.2 Extending the Configuration Database

All the configuration database entry names are defined in `rg/pkg/include/rg_conf_entries.h`. Usually, you would have to add the new entries to this include file. In this case, the Watchdog entries have already been included:

```c
#define Swatchdog "watchdog"
#define Smargin "margin"
#define Sboot_msg "boot_msg"
```

The configuration database is hierarchical, and it is therefore recommended to store all information related to a specific feature under the same root. In this case the Watchdog's information is stored under the "watchdog" entry. The **Swatchdog** entry defines the name of the root for the Watchdog information. The following entries are stored under this root: the application status (enabled or disabled), the margin, and the boot message.

The status of the Watchdog is stored in the volatile configuration database under "watchdog/disabled". There is no need to define the **Sdisabled** entry as it is already defined by other features. The value of "watchdog/disabled" is initially set to zero at boot time. We only use the "watchdog/disabled" to demonstrate what happens when you disable the Watchdog process. After the time margin passes, the Watchdog module will reboot the system.

Other entries are stored in the persistent configuration database. The **Smargin** entry stores the interval that determines when the application will send "keep-alive" signals to the kernel module. The **Sboot_msg** entry stores the message that OpenRG displays on the console screen before booting up.

The entries of the configuration database are defined as constant definitions, instead of using the actual strings. The purpose for this is catching typos at compilation time rather than at run time.
time, and using common entry names in all files making it easier to locate the entries using the `grep` command.

**rg_conf Serialization**

![Diagram of rg_conf Serialization]

Figure 5.4  rg_conf Serialization

### 5.4.3 Defining the Management API

The management API for the Watchdog application provides functions to store and retrieve the Watchdog configuration in the configuration database and activate or deactivate sending the keep-alive signal. This API is defined in the following file:

```c
rg/pkg/watchdog/mgt/watchdog_mgt.h
```

```c
#ifndef _WATCHDOG_MGT_H_
#define _WATCHDOG_MGT_H_

/* watchdog_mgt.h - rg conf API for watchdog package */

#define MAX_BOOT_MSG 512
#define DEFAULT_MARGIN 60

typedef struct {
    char msg[MAX_BOOT_MSG];
} msg_t;

typedef struct {
    int margin;
    msg_t boot_msg;
} watchdog_conf_t;

/* Retrieve the watchdog configuration from rg conf. */
void watchdog_conf_get(watchdog_conf_t *p);

/* Store the watchdog configuration to rg conf. */
void watchdog_conf_set(watchdog_conf_t *p);

/* Enable/disable the watchdog activity */
void watchdog_enable(int is_enabled);

/* Check if the watchdog is active */
int watchdog_is_enabled(void);

/* Check if the margin is valid */
int watchdog_is_margin_valid(int margin);
```

This include file defines several constants used by the Watchdog, such as DEFAULT_MARGIN and MAX_BOOT_MSG, and data structures, such as msg_t and watchdog_conf_t, used by the mgt API. Following, the watchdog_* functions are defined, which provide access to the current configuration and status of the Watchdog. Implementing this API is depicted in the following code:

```c
#include <set.h>
#include <main/openrg.h>
#include <watchdog/mgt/watchdog_mgt.h>

/* mgt.c - rg conf API for watchdog package */

/* Retrieve the watchdog rg conf */
static set_t **watchdog_set_get(void)
{
    return set_get(rg_conf, Swatchdog);
}

/* Retrieve the watchdog status from rg conf ram */
static set_t **watchdog_ram_set_get(void)
{
    return set_set(rg_conf_ram, Swatchdog);
}

/* Retrieve the watchdog configuration from rg conf. */
void watchdog_conf_get(watchdog_conf_t *p)
{
    /* Get watchdog's section in rg_conf */
    set_t **set = watchdog_set_get();

    /* Get the margin and boot message */
    p->margin = set_get_path_int(set, Smargin);
    strncpy(p->boot_msg.msg, set_get_path_strz(set, Sboot_msg), MAX_BOOT_MSG);
}

/* Store the watchdog configuration to rg conf. */
void watchdog_conf_set(watchdog_conf_t *p)
{
    /* Get watchdog's section in rg_conf */
    set_t **set = watchdog_set_get();

    /* Store the margin and boot message */
    set_set_path_int(set, Smargin, p->margin);
    set_set_path_str(set, Sboot_msg, p->boot_msg.msg);
}

/* Enable or disable the watchdog status in rg conf ram */
void watchdog_enable(int is_enabled)
{
    /* Get watchdog's status from rg conf ram */
    set_t **setRam = watchdog_ram_set_get();

    set_set_path_flag(setRam, Sdisabled, !is_enabled);
}

/* Check if the watchdog is active */
int watchdog_is_enabled(void)
{
    /* Get watchdog's status from rg conf ram */
    set_t **setRam = watchdog_ram_set_get();

    return !set_get_path_flag(setRam, Sdisabled);
}

/* Check if the margin is valid - in the range of 30 to 60 */
```
int watchdog_is_margin_valid(int margin)
{
    return (margin >= DEFAULT_MARGIN / 2 && margin <= DEFAULT_MARGIN);
}

The `rg_conf` and `rg_conf_ram` global variables are used to access the memory structures storing the configuration database. The `set.h` API allows reading and writing values to the configuration database. The configuration database stores only character strings, but provides an API to read and write these values as numbers, strings and boolean flags.

The static functions `watchdog_set_get` and `watchdog_ram_set_get` are used to retrieve the root of the Watchdog application configuration section in the database, for the persistent and volatile parts respectively.

The rest are API functions: `watchdog_conf_get` and `watchdog_conf_set` will retrieve and store the Watchdog configuration in the configuration database, including the margin and the boot message. `watchdog_enable` will turn on and off the keep-alive signal. `watchdog_is_enabled` is a boolean function used to check whether or not the Watchdog is sending keep-alive signals. Last is the validity check function, `watchdog_is_margin_valid`, which determines if the values of the margin are valid.

The API is now ready to be used by a client. In this case we are going to implement CLI commands to manage the Watchdog configuration.

### 5.4.4 Implementing CLI commands

The CLI commands that will manage and control the Watchdog application include the following commands, under the `watchdog` category:

- **conf** Configure the Watchdog margin and end message.
- **start** Hidden command to start the keep-alive signal.
- **status** Show the current configuration of the Watchdog.
- **stop** Hidden command to stop the keep-alive signal.

First, we need to define the CLI commands and register them with OpenRG’s CLI command processor. We register the commands during the initialization of the `mgt` component using `multilevel_cli_add_commands` available from `rg/pkg/cli/cli.h`. The registration function requires that we fill two data structures. The first describes the category of the commands registered and the second defines the commands and their arguments.
The `watchdog_cmd_category` defines the command category that will be displayed when the help command is used with no arguments. The `watchdog_commands` array defines the commands that will be displayed when using the help command on the Watchdog category.

For each command we define the command name, a short and long description, the function that will handle the command, and the parameter types that are required by this command. You
may also specify that a command is hidden by setting the flags to CMD_ITEM_HIDDEN. The parameter types allow both fixed and variable number of parameters. Indicating the pair CMD_PARAM_ARGC and CMD_PARAM_ARGV allows a variable number of arguments. Otherwise the number of arguments is fixed.

The command processor performs basic validity checks on the parameter values including strings (CMD_PARAM_STR) and numbers (CMD_PARAM_INT). A special type defines an output stream (CMD_PARAM_ESTREAM) to which the function can write the output, using the estream_printf function. The parameter list must end with the value CMD_PARAM_END.

The registration of commands should be hooked to the initialization of the Watchdog module. We will have to call the mgt initialization function from the Watchdog initialization function. This is done by creating an include file with the mgt initialization function, and using it in the Watchdog initialization function. Following is the mgt initialization API that must be written in:

```
#ifndef _WATCHDOG_MGT_INIT_H_
#define _WATCHDOG_MGT_INIT_H_

/* Register and unregister the watchdog CLI commands */
void watchdog_mgt_init(void);
void watchdog_mgt_uninit(void);

#endif
```

Following are the lines that need to be added to the Watchdog initialization function in `rg/pkg/watchdog/watchdog_init.c`:

```
... #include <watchdog/mgt/watchdog_mgt_init.h> ...
void watchdog_init(void) {
    rg_error(LCONSOLE, "Initializing Watchdog components.");
    watchdog_mgt_init();
}
void watchdog_uninit(void) {
    rg_error(LCONSOLE, "Un-initializing Watchdog components.");
    watchdog_mgt_uninit();
}
```

Finally, all we need to do is implement the four CLI commands defined for the Watchdog, using the Watchdog mgt API, as follows:

```
rg/pkg/watchdog/mgt/commands.c
...
#include <watchdog/mgt/watchdog_mgt.h>
static int watchdog_conf_cmd(estream_t *estream, int margin, char *boot_msg) {
    watchdog_conf_t conf;
    if (!watchdog_is_margin_valid(margin)) {
...
At this point, we could configure and compile the entire tree, which takes a lot of time. Instead, compilation of the package could be done incrementally. The 'make config' phase is responsible for exporting the headers. Since we have added a new component which requires exporting a header, we need to run 'make config' in the new mgt directory, and then run 'make':

```
$ cd rg/pkg/watchdog/mgt
$ make config
$ make
```

Once there are no errors in the mgt directory, we can go up to the Watchdog directory and run 'make', to link mgt with the rest of the Watchdog package.

```
$ cd ..
$ make
```

Now that the Watchdog package object is built, we can build the OpenRG main process by building rg/pkg/main.

```
$ cd ../main
$ make
```

The result should be in the rg/build/pkg/main/openrg program, which includes the Watchdog CLI commands. You can load it to the board and test the behavior of the commands. But before you do so, let's learn how to set up the default values for the persistent rg_conf entries.
5.4.5 Setting Up Default Values

The default values of the configuration database entries are generated by the file `rg/pkg/main/gen_rg_def.c` that runs on the host. You may set default values to the entries by adding the following line to this program:

```c
#include <watchdog/mgt/watchdog_mgt.h>
...
set_set_path_int(rg_conf, Swatchdog "/" Smargin, DEFAULT_MARGIN);
```

This line explicitly sets the default value of the Watchdog margin to the value defined in `watchdog_mgt.h`. In general, the default value of a missing entry in the database is NULL. In case this entry is numeric, it is the same as explicitly setting it to zero. In this case, the `Sboot_msg` entry was not assigned with a default value, and it is automatically assigned the value NULL.

5.4.6 Testing the Application CLI Commands

The functionality of the Watchdog commands can be tested through the serial console:

1. Connect your development board to the PC’s serial connector using a flat serial cable.

2. Open a terminal program. The OpenRG login prompt appears.

3. Enter your user name and password. Unless you made changes to the default settings, both of these values should be ".

   Username: admin
   Password: *****
   OpenRG>

   Use the Watchdog configuration command 'watchdog conf' to configure the Watchdog:

   ```bash
   OpenRG> watchdog conf 50 "End message"
   Returned 0
   OpenRG>
   ```

   Display the Watchdog configuration using the 'watchdog status' command:

   ```bash
   OpenRG> watchdog status
   Watchdog status: Started
   Keep alive signal margin: 50
   Boot message: End message
   returned 0
   OpenRG>
   ```

   You can verify the Watchdog commands’ functionality by using OpenRG’s CLI commands, which examine the configuration database. Use the 'conf print' command to print the entire configuration database.

   ```bash
   OpenRG> conf print
   Wrong number of arguments
   print   Print OpenRG configuration
   Usage:
   print <path>
   or print / to print the whole configuration.
   Returned 1
   OpenRG>
   ```
This command will print the entire contents of the configuration database to the console, which usually provides too much information. To view configuration database information that is relevant to the Watchdog only, use the following command:

```
OpenRG> conf print watchdog
(watchdog
  (margin(50))
  (boot_msg(End message))
)
```

The output displays all of the Watchdog entries in the configuration database. You can see that the margin interval is 50 seconds and the boot message is "End Message". To check whether the Watchdog is enabled, check the volatile configuration database. You can use the CLI command 'conf ram_print' which is similar to the 'conf print' command, but works on the volatile configuration database. Type the following command:

```
OpenRG> conf ram_print watchdog
Returned 0
```

Since only 'Returned 0' appears, it is safe to assume that the Watchdog is enabled, as if it had already been disabled once, a more explicit output would appear, with 0 or 1 denoting the 'disabled' status.

```
OpenRG> conf ram_print watchdog
(watchdog
  (disabled(0))
)
```

Usually, the CLI commands are not used by end-users but rather by developers and technicians who need low level access to the system. For a comprehensive reference on the CLI, refer to Chapter 27.

### 5.4.7 Lesson Summary

Going through the steps described above, you have learnt:

- How to extend the configuration database with user defined entries.
- How to define application data elements to the configuration database entries.
- How to set default values to these entries.
- How to extend the CLI with application specific commands.
- How to use the CLI to view and set configuration database entries.

For a complete reference regarding the files used in this stage, refer to this tutorial's appendix (Section 5.8). The next lesson will show you how to extend the Web-based management GUI, which is used by end-users to monitor and manage the behavior of the device.
5.5 Stage 3 – Creating a Management Page

The Web-based management library (engine) is stored in the `rg/pkg/web_mng` directory. Under this directory you will find the `lib`, `cgi` and `images` directories. The `lib` directory contains the management library. The `cgi` directory contains some entity-less pages. The `images` directory contains all of the images that are used in the Web-based management. There are many other WBM subdirectories across the development tree, one per feature. In these WBM subdirectories you will find the WBM pages of specific entities. It is recommended that new packages added to OpenRG provide their WBM pages as part of the package rather than making changes to the WBM engine. This is the approach we will take with the Watchdog WBM pages. This stage describes how to:

- Create the Watchdog WBM component in the Watchdog package. The WBM component displays the Watchdog configuration and allows the user to modify it.

- Create the WBM page print function that displays the current configuration of the Watchdog component as expressed by `mgt`.

- Create the WBM page scan function that analyzes the user input and stores it back to the configuration database using `mgt`.

- Tie the Watchdog WBM page to the rest of the WBM during the initialization process.

5.5.1 Creating the Watchdog WBM Component

The addition of the Watchdog WBM component is quite similar to the addition of the `mgt` component presented earlier. You need to create a `wbm` directory under the `watchdog` package, have the Watchdog Makefile enter this directory, and write the Makefile and the initialization code for the WBM component.

- The first step is to create the directory:
  ```
  $ cd ~/tutorial/rg/pkg/watchdog
  $ mkdir wbm
  ```

- The next step is to enhance the Watchdog Makefile so it enters the `wbm` subdirectory, and link the object from this directory to the Watchdog object. Following are the lines to be added. The first line adds the `wbm` directory to the JMK_SUBDIRS variable. The second, adds the `wbm` object file to the link of the Watchdog object.
  ```
  JMK_SUBDIRS+=wbm
  JMK_L_OBJS+=wbm/watchdog_wbm.o
  ```

- The last step is to add the Makefile to the `wbm` directory and develop the code.

```
rg/pkg/watchdog/wbm/Makefile

1     ifndef JMKE_ENV_INCLUDED
2       JMK_ROOT=../../../
3       include $(JMK_ROOT)/jmk/env_root.mak
```

© 1998-2012 Jungo Ltd.
The Makefile structure is the same as the Makefile of the mgt component. They only differ by the name of the object files created and the include files exported.

**Lines 1-4,16** The standard header and footer of an OpenRG Makefile.

**Lines 6-7** Define that the WBM component API includes only one file, `watchdog_wbm_init.h`, which will be exported to the `rg/build/pkg/include/watchdog/wbm` directory.

**Line 9** Declare an internal header file used by one of the targets.

**Lines 11-12** Define that the object created here is `watchdog_wbm.o`, which is composed of the object files `watchdog_wbm_init.o` and `watchdog_page.o`. These object files are created from the respective source files `watchdog_wbm_init.c` and `watchdog_page.c`.

**Line 14** Copy the icon to the RAMDISK into the image location.

**Line 15** Include the new image in the configuration process.

The `page_dog.png` image file should be copied into the `watchdog/wbm` directory.

A new directive in the WBM Makefile is the JMK_RAMDISK_IMG_FILES, which is used to define graphic files used on the Web pages of this component. The graphic files should be in either GIF or PNG format. They are automatically copied to the OpenRG image into the `/home/httpd/html/images` directory.

The Watchdog WBM component includes the icon of the Watchdog, used to navigate to the Watchdog page and visible on the page’s header. Each WBM page is printed by one function and scanned by another. The next step will be to create the print and scan functions, and integrate them with the WBM engine.

### 5.5.2 Creating a Print Function

We will start by building the Watchdog management page that shows the current configuration and allows the user to modify it. The page contains a check box and two text box fields. The check box will enable/disable the Watchdog feature, and the text boxes will store the "Keep Alive Signal Margin" and the "Ending Message". Create the following files:

**rg/pkg/watchdog/wbm/watchdog_page.c** A file containing two functions: `p_scr_<page-name>`—a function that prints the page to the screen.
s_scr_<page-name>—a function that extracts data from the page.

rg/pkg/watchdog/wbm/watchdog_page.h A header file containing the print and scan function prototypes.

A specific print function is called for every Web-based management page that is printed. The function gets a global HTML handle and prints the page structure and details. The param argument can be used to send data to this function, usually from the scan function (in the Watchdog example, this value is NULL). To construct the Watchdog page printing function, enter the following code into:

rg/pkg/watchdog/wbm/watchdog_page.c

```c
#include <web_mng/lib/html_wbm.h>
#include <web_mng/lib/wbm.h>
#include <web_mng/lib/wbm_util.h>
#include <web_mng/cgi/wbm_lang.h>
#include <web_mng/lib/wbm_mgt.h>
#include <mgt/lib/mgt_lang.h>
#include <util/str.h>
#include <main/openrg.h>
#include <watchdog/mgt/watchdog_mgt.h>

#define SYM_ENABLED "enabled"
#define SYM_SIG_MARGIN "sig_margin"
#define SYM_MSG "msg"

void p_scr_watchdog(html_t **handle)
{
    html_t **frame, **table, **cell;

    watchdog_conf_t watchdog_conf;

    frame = p_page_header(handle, page_watchdog, NULL, ICON_DOG);

    table = p_table(frame, MAKE_PERCENT(60), TABLE_STANDARD, 1);

    cell = p_row_std_data(table);
    p_enabled_checkbox(cell, SYM_ENABLED, FIELD_ACTIVE, watchdog_is_enabled(), 0);

    watchdog_conf_get(&watchdog_conf);

    cell = p_row_std(table, Tkeep_alive_signal_margin, 0);
    p_edit_box(cell, SYM_SIG_MARGIN, itoa(watchdog_conf.margin), 2, 2);
    p_text(cell, "%s", Tseconds);

    cell = p_row_std(table, Tending_message, 0);
    p_edit_box(cell, SYM_MSG, watchdog_conf.boot_msg.msg, 30, MAX_BOOT_MSG-1);

    p_buttons(NULL, NO_BTN_DATA, btn_submit, btn_cancel, 0);
}
```

What does this code do?

Line 9 This is the mgt API for the rg_conf database Watchdog entries.

Lines 12-14 These are the names of the fields on this page. The fields are named so that the scan function can retrieve them by name.

Line 22 Print a page header including icon and title. The variable frame is the page's handle.

Line 23 Print a non-transparent table that will occupy 60% of the frame.

Lines 25-26 Allocate a new row, and print a check box inside it. The check box caption is "Enabled". The value is defined by the mgt function watchdog_is_enabled.
**Line 28** Retrieve the Watchdog configuration from the `rg_conf` database using the `mgt` function `watchdog_conf_get`.

**Lines 30-32** Allocate a row with two columns. The left column contains the title "Keep Alive Signal Margin", and the right column contains an edit box with a maximal length of 2 characters. The value for the edit box is converted to ASCII from the Watchdog configuration variable `watchdog_config_margin`. Right after the edit box we print the word "seconds" to indicate the units.

**Lines 34-35** Allocate a row with two columns. The left column contains the title "Ending Message", and the right column contains an edit box with a display length of 30 characters, and a value maximum length that will not exceed `MAX_BOOT_MSG-1`. The value for the edit box is taken from the Watchdog configuration variable `watchdog_conf.boot_msg.msg`.

**Line 37** Print the following push buttons at the bottom of the screen:

- 'OK' Button (btn_submit) - applies changes and returns to the previous page.
- 'Apply' Button (printed automatically with btn_submit) - applies changes and stays at the same page.
- 'Cancel' Button (btn_cancel) - discards changes and returns to the previous page.

You may have noticed that all the constant text strings printed to the screen are not typed in explicitly, but use the `Tsome_text` notification. For example, look at `Tkeep_alive_signal_margin` on line 30. In order to provide a multilanguage GUI, all of these strings, together with all of the Web-based management text, are declared in special language files. These files have the suffix `_lang.csv` and are compiled by the language compiler into C files and header files.

The WBM engine contains such a file, `rg/pkg/web_mng/cgi/wbm_lang.csv`, which includes many of the constants used in the WBM GUI. To add constant strings, you may choose to either create a new language file for the package, or extend the WBM language file. In this tutorial we chose to add all the strings used by the WBM component to the `rg/pkg/web_mng/cgi/wbm_lang.csv` file. You may leave empty fields for the comment and all languages except English. The following lines should be added to the WBM language file:

```
Tkeep_alive_signal_margin,,,,"Keep Alive Signal Margin",,,,,,,,,
Tending_message,,,,"Ending Message",,,,,,,,,
```

The entire tree must now be compiled for these changes to take effect.

**Note:** `seconds` is not defined here since it is already defined in `web_mng/lib/libwbm_lang.csv` for a different WBM page.

### 5.5.3 Creating a Scan Function

Every time the user presses a button, the specific scan function of the current page is called (unless it's a generic button such as Cancel, which is handled in the WBM low-level). The function reads the information that the user entered, validates it and stores it in the system
configuration database. The function then calls the `openrg_reconf()` function in order to signal to main task that the configuration has been changed, and selects the next page to be printed. Every scan function handles some or all of the following topics:

1. Check the button that the user pressed and act accordingly. In many cases the only button is the 'OK' button (btn_submit).

2. Extract the data entered by the user and validate the values. Use the extract structure to define the field types and variables that receive the extracted values. For more information, refer to Section 28.3.5.

3. Check if there are any errors or warnings that should be displayed to the user.

4. Save the values entered by the user to the configuration database.

5. Navigate the user to the next page. In most cases, the user goes back to the previous page. This is accomplished by the page navigation stack mechanism.

Write the code for the scan function `s_scr_watchdog`, which handles all of the above mentioned topics as required for the Watchdog page.

```
rg/pkg/watchdog/wbm/watchdog_page.c

void s_scr_watchdog(void)
{
    int is_enabled;
    char *msg = NULL;
    watchdog_conf_t watchdog_conf;

    extract_data_t extract[] = {
        {SYM_ENABLED, {check:&is_enabled}, TYPE_CHECKBOX, Tempty},
        {SYM_SIG_MARGIN, {num:&watchdog_conf.margin}, TYPE_WATCHDOG_MARGIN,
            Tkeep_alive_signal_margin},
        {SYM_MSG, {str:&msg}, TYPE_STR_WITH_SPACES|TYPE_OPTIONAL,
            Tending_message},
        {NULL}
    };

    if (button_num()!=btn_submit)
        goto Exit;

    extract_data(extract);
    if (g_request->errors)
        goto Exit;

    strncpyz(watchdog_conf.boot_msg.msg, msg, MAX_BOOT_MSG-1);
    watchdog_conf_set(&watchdog_conf);

    watchdog_enable(is_enabled);

    openrg_reconf(FLASH_DELAYED_TOP_PRIORITY);

    active_page_set(navigator_pop());
    Exit:
    p_page(NULL);
}
```

Which lines handle the different topics?

1. Check the button that the user pressed and act accordingly.

   **Lines 16-17** Ignore buttons other than the 'OK' (btn_submit).
2. Extract the data entered by the user and validate the values.

Lines 7-14 Define a table with the following information per form field: field name (relevant SYM_*), reference to the relevant target variable, extraction type, and error message title. The reference is set to the appropriate union member. The extract_data_t and the extraction type are defined in \texttt{rg/pkg/web_mng/lib/wbm_util.h}. The table must terminate with a NULL symbol entry. The information stored in this table provides OpenRG with the necessary information in order to validate user input. For example, line 8 defines a form field of check box type (TYPE_CHECKBOX), named SYM_ENABLED, which is bound to the \texttt{is\_enabled} variable. The implementation of the different types of extraction is located in \texttt{rg/pkg/web_mng/lib/wbm_util.c}.

Line 19 Extract data into the target variables according to the structure of the table.

3. Check if there are any errors or warnings that should be displayed to the user.

Lines 20-21 Exit if an error was encountered during data extraction. The p\_page function will display the appropriate error messages at the top of the page.

4. Save the values entered by the user to the configuration database.

Lines 23-24 Update the Watchdog's margin and boot message using the Watchdog mgt function \texttt{watchdog\_conf\_set}.

Line 26 Set the Watchdog's status using the Watchdog mgt function \texttt{watchdog\_enable}.

Line 28 Reconfigure the system and flush the permanent configuration database entries to the permanent storage.

5. Navigate the user to the next page.

Lines 30-32 Return to the previous page by popping the current page from the top of the page navigator stack.

The following code must be added to \texttt{rg/pkg/web_mng/lib/wbm_util.c}:

```
case TYPE_WATCHDOG_MARGIN:
    local_error = extract_numerical(list->name, list->u.num, 
                                      30, 60, 1, 10, list->error_title, scan_type, &list->had_errors, 
                                      0);
    break;
```

### 5.5.4 Tying the Watchdog WBM Page with the WBM

The Watchdog WBM page should be tied in with the rest of the WBM during the initialization of the Watchdog package. For that purpose we need to create a Watchdog WBM initialization file that will be invoked from the package initialization, and introduce the Watchdog page to the WBM engine. The Watchdog WBM page API includes the print and scan functions. The
Watchdog WBM initialization should provide `init` and `uninit` functions. In addition we need to introduce the Watchdog icon to the WBM engine. We are going to create or modify the following files:

**rg/pkg/watchdog/wbm/watchdog_page.h**  The API of the Watchdog WBM page that includes the print and scan functions.

**rg/pkg/watchdog/wbm/watchdog_wbm_init.h**  The API of the Watchdog page initialization that includes the `init` and `uninit` function prototypes for the Watchdog WBM page.

**rg/pkg/watchdog/wbm/watchdog_wbm_init.c**  A file that implements the functions defined in the API above: `watchdog_wbm_init` - registers the Watchdog WBM page. `watchdog_wbm_uninit` - unregisters the Watchdog WBM page.

**rg/pkg/web_mng/lib/wbm_types.h**  Add the Watchdog icon to the enumeration list of all icons known to the WBM engine.

1. Following is the code for the Watchdog WBM page API. Enter the following code into `rg/pkg/watchdog/wbm/watchdog_page.h`:

```c
#ifndef _WATCHDOG_PAGE_H_
#define _WATCHDOG_PAGE_H_
#include <util/html.h>
void s_scr_watchdog(void);
void p_scr_watchdog(html_t **handle);
#endif
```

2. Defining a `page_id` in `rg/pkg/web_mng/cgi/page_id_str.h`

```c
PAGE_ID(page_watchdog)
```

Note: This line already exists.

3. Integrating the new page into the Web-based management requires that we provide `init` and `uninit` functions as defined in the API below. Enter the following code into `rg/pkg/watchdog/wbm/watchdog_wbm_init.h`:

```c
#ifndef _WATCHDOG_WBM_INIT_H_
#define _WATCHDOG_WBM_INIT_H_
void watchdog_wbm_init(void);
void watchdog_wbm_uninit(void);
#endif
```

4. Following is the implementation of the `init` and `uninit` functions.

```c
#include "watchdog_page.h"
#include <web_mng/lib/wbm.h>
#include <web_mng/cgi/wbm_lang.h>
#include <web_mng/lib/libwbm_lang.h>
#include <web_mng/cgi/page_id_str.h>
```
What does this code do?

**Lines 9-10** Register the Watchdog page with the WBM engine by calling `page_register()`. The arguments to `page_register()` include the page properties such as the title of the page (`Twatchdog_triggering`), the icon to display at the top of the page (`ICON_DOG`), the print function, and the scan function.

**Lines 15** Unregister the Watchdog from the WBM engine.

5. Now, we need to invoke the Watchdog WBM `init` and `uninit` functions from the Watchdog package `init` and `uninit` functions. Add the following lines into the `rg/pkg/watchdog/watchdog_init.c` file in the appropriate places (next to `mgt init` and `uninit`):

```c
#include <watchdog/wbm/watchdog_wbm_init.h>
...
watchdog_wbm_init();
...
watchdog_wbm_uninit();
```

6. The Watchdog icon is introduced to the WBM engine by adding it in two files, the enumeration list of all icons and the table that associates every icon with a graphic file. The `icon_t` enumeration in `rg/pkg/web_mng/lib/wbm_types.h` should be extended with the Watchdog icon:

```
ICON_DOG=<INTEGER>
```

Note: Make sure that the INTEGER specified in the enumeration is unique.

Binding the Watchdog enumeration to the Watchdog icon graphic file is done in `rg/pkg/web_mng/lib/wbm_db.c` under the `icons[]` array:

```
{ICON_DOG, "page_dog.png"}
```

Note: These lines already exists, so all you need to do is activate them.

7. Now, we need to register this page as a new service in OpenRG, which can be reached via its own link under the 'Advanced' menu item of the 'Services' tab (see Figure 5.3). In order to do so, we need to modify `rg/pkg/web_mng/cgi/rg_struct.h` and add a new menu.

To add a new menu ID, add the following:

```
TOPBAR_SECTION_ID_SVC_ADVANCED_WATCHDOG = 866,
```

To add a new advanced service entry, add the following:

```c
topbar_register(&tabs2->submenu, TOPBAR_SECTION_ID_SVC_ADVANCED_WATCHDOG, NULL, page_watchdog, RG_TOPBAR_PREFIX "svc_advanced_watchdog", 1, T_PTR(Twatchdog_triggering));
```
Note that we need to add this entry to the correct submenu. Therefore, locate the correct place in this file where your menu is being created, and add the new lines. To make it optional, add the following:

```
topbar_set_optional(tabs2->submenu, TOPBAR_SECTION_ID_SVC_ADVANCED_WATCHDOG);
```

### 5.5.5 Compilation

1. After configuring the tree (with 'make config'), make sure you are in the `rg/pkg/watchdog/wbm` directory, and compile. This way, you will find any compilation errors quickly without having to wait for the whole tree to compile.

```
$ cd ~/tutorial/rg/pkg/watchdog/wbm
$ make
```

2. Correct any compilation errors you may have. Since no major changes were made to your working distribution, any errors you may encounter now are probably the result of typing errors, and will therefore be easy to correct.

3. Compile the entire development tree, this time from the top of the tree.

```
$ cd ../../..
$ make
```

4. To verify that your compilation was successful, make sure that the Watchdog icon (page_dog.png) was copied to OpenRG's image. You should be able to locate it in the directory `rg/build/pkg/build/disk_image/cramfs_dir/home/httpd/html/images`.

5. A remote upgrade image is created—`build/openrg.rmt`. Use the Web-based management to transfer the image to your board. For detailed instructions on performing a remote firmware upgrade using OpenRG's Web-based management, refer to the 'Upgrading the Gateway's Firmware' section of the OpenRG Administrator Manual.

6. Click the 'Advanced' menu item under the 'Services' tab. You should see that a 'Watchdog Triggering' link has been added to the links bar.

7. Click this link. The 'Watchdog' screen appears (see Figure 5.3).

### 5.5.6 Lesson Summary

Going through the steps described above, you have learnt:

- How to build a management page using the print and scan functions.
- How to integrate new WBM pages into OpenRG's WBM engine and link them to the 'Advanced' page.
- How to compile an image.

For a complete reference regarding the files used in this stage, refer to this tutorial's appendix (Section 5.8).
5.6  Stage 4 – Integrating the Application

This stage presents a way to compile, load and execute an additional Linux process that communicates with OpenRG. This stage consists of two phases. The first phase uses a sample process to demonstrate the way processes are handled. The second phase presents the full implementation of the Watchdog handling process.

5.6.1  Creating the Watchdog Process Component

The addition of the Watchdog process component is relatively simple. You must create a process directory under the watchdog package, have the Watchdog Makefile enter this directory, and write the Makefile for the process component.

- The first step is to create the directory:
  
  ```
  $ cd ~/tutorial/rg/pkg/watchdog
  $ mkdir process
  ```

- The next step is to enhance the Watchdog Makefile so it enters the process subdirectory. The only line that must be added, adds the process directory to the JMK_SUBDIRS variable. Since the process Makefile creates a stand-alone application, it is not linked with OpenRG.

  ```
  JMK_SUBDIRS+=process
  ```

- The last step is to add the Makefile to the process directory and develop the code.

  `rg/pkg/watchdog/process/Makefile`

  ```
  1  ifndef JMKE_ENV_INCLUDED
  2    JMK_ROOT=../../../
  3    include $(JMK_ROOT)/jmk/env_root.mak
  4  endif
  5
  6  JMK_INTERNAL_HEADERS+=mgt_utils.h
  7
  8  JMK_TARGET=watchdog
  9  JMK_RAMDISK_BIN_FILES+=$(JMK_TARGET)
 10  JMK_O_OBJS=watchdog_proc.o
 11
 12  JMK_LIBS+=$(JMKE_BUILDDIR)/pkg/util/libmgt_client.a $(OPENRG_LIBS)
 13
 14  $(call JMKE_INCLUDE_RULES)
  ```

The Makefile structure is a simple RG Makefile and contains only JMK_TARGET and JMK_O_OBJS directives.

**Lines 1-4, 14** The standard header and footer of an RG Makefile.

**Line 6** Defines headers that are used by the current library only.

**Line 8** Defines that the Watchdog program is created in this directory and will be copied to the target image.
5.6.2 A Simple Example

To become familiar with running external application with OpenRG, you will start with a simple example.

1. Create the file `watchdog_proc.c` under the `process` directory, and type in the following:

```c
#include <unistd.h>
#include <util/rg_print.h>

int main(void)
{
    console_printf("Hello World!\n");
    while (1)
    {
        sleep(10);
        console_printf("I'm still alive..\n");
    }
    /* Never reaching point */
    return 0;
}
```

This simple program does not do much, however compiling, burning and running it will make you familiar with the development process. The program simply prints a "Hello World!" message, waits for 10 seconds and prints "I'm still alive.." repeatedly, every 10 seconds.

2. Compile the program and the image. This is done in the exact same method described in Stage 1 of the tutorial. Use 'make' from the root directory to compile, and transfer the image to the board using the Remote Update feature from the Web-based management.

3. Type the following at the CLI prompt:

```sh
OpenRG> system shell
BusyBox v1.01 (2005.09.07-07:38+0000) Built-in shell (lash)
Enter 'help' for a list of built-in commands.
/
```

You have now entered the `BusyBox` shell, granting access to the system's file system (for more information, refer to Chapter 22). You can find all available applications under the `bin` directory:

```sh
/ # cd bin
/bin # ls
ln pptp_callmgr spi
eroute ls pptpcntl spigrp
grep lsmod pptpd sync
gunzip openrg ps tail
gzip mkdir pwd tar
head more ranbits tftp
hotplug mount rm tncfg
ifconfig mv rmdir touch
```

Line 9 Defines that the target will be copied to the `/bin` directory on the ramdisk.

Line 10 Indicates that the Watchdog program is created from the object file `watchdog_proc.o` (which is compiled from `watchdog_proc.c`).
4. The simple application at the top of this section is called 'watchdog', according to the application file name. Activate the program by typing the application name at the prompt:

```
/bin # watchdog
Hello world!
I'm still alive..
```

5.6.3 The Watchdog Process

After becoming familiar with these surroundings, you are ready to continue with the Watchdog example. The Watchdog is an external application, that communicates with Main Task using OpenRG's local management API. During several short local management sessions, the application samples the Watchdog configuration file entries. The application continues its operation, using the Watchdog configuration file values as parameters. The Watchdog checks for a new margin time and ending message and sends a keep-alive signal accordingly.

5.6.3.1 Integration with OpenRG

Replace the contents of `watchdog_proc.c` with the following code:

```c
#include <unistd.h>
#include <syscalls.h>
#include <util/mgt_client.h>
#include <util/rg_chardev.h>
#include <util/rg_print.h>
#include <rg_conf_entries.h>
#include <watchdog/module/watchdog_mod.h>
#include <misc_funcs.h>
#include "mgt_utils.h"

int main(void)
{
    set_t *rg_conf_set, *rg_conf_ram_set;
    int is_disabled, cur_margin = DEFAULT_MARGIN, mt_margin;
    msg_t cur_msg, mt_msg;
    console_printf("Watchdog process control started...\n");
    MZERO(cur_msg);
    while (1)
    {
        /* Read Watchdog sections */
        rg_conf_set = mgt_rg_conf_get(Swatchdog, 0);
        rg_conf_ram_set = mgt_rg_conf_get(Swatchdog, 1);

        in the volatile configuration database (rg_conf_ram), check whether the Watchdog process should be active or not, and copy the retrieved information into a local variable.
        is_disabled = set_get_path_int(&rg_conf_ram_set, Sdisabled);
```
Copy information retrieved from the persistent database (using local management) into the corresponding local variables.

```c
mt_margin = set_get_path_int(&rg_conf_set, Smargin);
strncpy(mt_msg.msg, set_get_path_strz(&rg_conf_set, Sboot_msg), MAX_BOOT_MSG);
```

Check to see if there was a change in the 'keep-alive' signal margin. In case a change occurred, communicate with the kernel module using an IOCTL that sets the updated margin in the module accordingly.

```c
/* Check new margin */
if (mt_margin && cur_margin != mt_margin)
{
    console_printf("Got new margin: [%d]\n", mt_margin);
    cur_margin = mt_margin;
    /* Send IOCTL that changes the margin in the kernel module */
    openrg_module_ctrl(KOS_CDT_WATCHDOG, WATCHDOG_IOCTL_SIG_MARGIN, &mt_margin);
}
```

Check to see if there was a change in the 'Ending Message'. In case a change occurred, communicate with the kernel module using an IOCTL that sets the updated message in the module accordingly.

```c
/* Check new boot message */
if (strncmp(cur_msg.msg, mt_msg.msg, MAX_BOOT_MSG))
{
    console_printf("Got new boot message: [%s]\n", mt_msg.msg);
    strncpy(cur_msg.msg, mt_msg.msg, MAX_BOOT_MSG);
    /* Send IOCTL that changes the boot message in the kernel module */
    openrg_module_ctrl(KOS_CDT_WATCHDOG, WATCHDOG_IOCTL_BOOT_MSG, &mt_msg);
}
```

The final portion of the application code sends a 'keep-alive' signal to the kernel module, as long as the `is_disabled` variable is false.

```c
if (!is_disabled)
{
    /* Sending Keep-alive signal */
    openrg_module_ctrl(KOS_CDT_WATCHDOG, WATCHDOG_IOCTL_KEEPALIVE, NULL);
}
```

Free the local allocated set structure, and wait for half of the Watchdog margin.

```c
set_free(&rg_conf_set);
set_free(&rg_conf_ram_set);
sleep(cur_margin/2);
```

This termination code will never be called since it is outside of the `while(1)` infinite loop. It is added to prevent compilation warnings only.

```c
} return 0;
```

### 5.6.3.2 Running the Application with `rg_system`

To load the application during the boot sequence, you must use `rg_system()`, which runs an external process application from within OpenRG code (either in foreground or background). The function resides in `libopenrg (#include <misc_funcs.h>)`, and is compiled by `JMK_LIBS+=$(OPENRG_LIBS)` which is a flag defined during 'make config' that includes all OpenRG libraries. The function's format is:

```c
int rg_system(char *ext_cmd, int flags, pid_t *ret_pid);
```
Its parameters are:

- **ext_cmd** The command to be executed.
- **int flags**
  - **SYSTEM_LOG** Add a record about running the `ext_cmd` command to the system log.
  - **SYSTEM_DAEMON** Run the command in the background (do not wait until it exits).
- **ret_pid** If not NULL, will contain the process ID of the executed command.

Its possible return values are:

- **exit status** If SYSTEM_DAEMON is not set.
  - `-2, -1, 0` If SYSTEM_DAEMON is set.
    - `-2` If no child was spawned due to an error.
    - `-1` If an error occurred after the child was spawned.
    - `0` On success.

To use this function in order to load the Watchdog during the boot sequence, add the following code to `rg/pkg/main/run_ext_proc.c`:

```c
pid_t ret_pid;
rg_system("/bin/watchdog", SYSTEM_LOG | SYSTEM_DAEMON, &ret_pid);
```

### 5.6.4 Using the Local Management Utilities

Accessing the Watchdog configuration from an external process is done by the local management, which uses Inter Process Communication (IPC). The **mgt** utilities should be linked with the Watchdog process.

1. The first step is to copy the local management wrapper files, `mgt_utils.c` and `mgt_utils.h`, from the `rg/pkg/samples/tutorial/process` directory to your working directory `rg/pkg/watchdog/process`. As you can see, two configuration access functions are supported by the wrapper, `mgt_rg_conf_get` and `mgt_rg_conf_set`. In the Watchdog process, only the first one is used since information is only read from the Watchdog configuration database. If you need to update the configuration database in your application, use the set function to do so.

   ```c
   set_t *mgt_rg_conf_get(char *path);
   int mgt_rg_conf_set(char *path, set_t **set);
   ```

- **mgt_rg_conf_get** This function allocates the whole set in the persistent configuration file (`rg_conf`) or in the volatile configuration file (`rg_conf_ram`) located under `path`. Note that the returned set is a duplication of the original set, and therefore must be freed using the `set_free` function.
**mgt_rg_conf_set**  This function replaces the whole set in the persistent configuration file (`rg_conf`) located under 'path' with the supplied set parameter.

2. In order to link the **mgt** utilities with the Watchdog process, you need to update the Makefile that builds the Watchdog external application. It should also compile the **mgt** utilities `mgt_utils.c`, and link with the appropriate libraries. You should replace the `JMK_O_OBJS` directive and add the `JMK_LIBS` directive in the Makefile you have created:

```bash
rg/pkg/watchdog/process/Makefile

1     JMK_O_OBJS=watchdog_proc.o mgt_utils.o
2
3     JMK_LIBS+=${JMKE_BUILDDIR}/pkg/util/libmgt_client.a ${OPENRG_LIBS}
```

Following is a line-by-line explanation of the Watchdog process Makefile:

**Line 1** Adds `mgt_utils.o` to the `JMK_O_OBJS` variable to be compiled and linked with `watchdog_proc.o` to create the JMK_TARGET.

**Line 3** The `JMK_LIBS` variable is used by the linker to define libraries. It is extended here with the `mgt` client and OpenRG libraries that are needed to build the Watchdog application.

### 5.6.5 Lesson Summary

Going through the steps described above, you have learnt:

- How to add a simple application to OpenRG.
- How to communicate between an external application and Main Task using local management.
- How an external application can extract configuration file entries.
- How to load the application during the system's boot sequence.

For a complete reference regarding the files used in this stage, refer to this tutorial's appendix (Section 5.8).

### 5.7 Stage 5 – Integrating the Kernel Module

The Watchdog feature includes a kernel module that monitors the keep-alive signal, and is responsible to reset the system if no signal arrived during the predefined time margin. In this stage, we will learn how to create a new kernel module. We will use OpenRG's generic *char device* and implement the IOCTL function that enables communication with the user mode.
application. We will add the timer logic required to implement the Watchdog itself. Finally, we will see how to automatically load the module during the boot sequence.

5.7.1 Creating the Watchdog Kernel Module Component

The addition of the Watchdog kernel module component is quite similar to the addition of previous components we have already seen. We need to create a module directory under the watchdog package, have the Watchdog Makefile enter this directory, and write the Makefile for the kernel module component.

- The first step is to create the directory:

```bash
$ cd ~/tutorial/rg/pkg/watchdog
$ mkdir module
```

- The next step is to enhance the Watchdog Makefile so it enters the module subdirectory. The only line that must be added, adds the module directory to the JMK_SUBDIRS variable. Since the kernel module Makefile creates a stand-alone kernel module, it is not linked with OpenRG.

```make
JMK_SUBDIRS+=module
```

- The last step is to add the Makefile to the module directory and develop the code.

```make
1     ifndef JMKE_ENV_INCLUDED
2       JMK_ROOT=../../../
3       include $(JMK_ROOT)/jmk/env_root.mak
4     endif
5
6     JMK_EXPORT_HEADERS_DIR=watchdog/module
7     JMK_EXPORT_HEADERS=watchdog_mod.h
8
9     JMK_MOD_TARGET=watchdog_mod.o
10    JMK_RAMDISK_MODULES_FILES=$(JMK_MOD_TARGET)
11    JMK_O_OBJS_watchdog_mod.o=watchdog.o
12
13    # Note that any object that exports symbols (uses EXPORT_SYMBOL macro),
14    # should be in JMK_OX_OBJS_<modulename> variable and not in JMK_O_OBJS_<modulename>.
15
16    $(call JMKE_INCLUDE_RULES)
```

The Makefile structure is a simple RG Makefile and contains only the JMK_MOD_TARGET and the JMK_O_OBJS directives.

Lines 1-4,16 The standard header and footer of an RG Makefile.

Lines 6-7 Export the kernel module API watchdog_mod.h to rg/pkg/include/watchdog/module.

Line 9 Setting the JMK_MOD_TARGET variable defines that the module watchdog_mod.o is created in this directory, and will be copied to the target image /lib/modules directory.
Line 10 Defines that the kernel module will be copied to the /lib/modules directory on the modfs file system.

Line 11 Sets the JMK_O_OBJ watchdog_mod.o variable, indicating that the Watchdog kernel module is created from the object file watchdog.o (which is compiled from watchdog.c). It is similar to the JMK_O_OBJ variable presented earlier. The difference is that we explicitly define the target's (watchdog_mod.o) object files. This syntax allows us to build several modules, or targets, in the same directory. The source code file name is derived from the object file name, as presented earlier. Note that any object that needs to export symbols (i.e. use the EXPORT_SYMBOL macro), should be in the JMK_O_OBJ[<modulename>] variable instead of the JMK_O_OBJ[<modulename>].

5.7.2 Creating the Kernel Module

The Watchdog kernel module is implemented in watchdog.c, in the directory rg/pkg/watchdog/module. Following is an example of a minimalistic kernel module:

rg/pkg/watchdog/module/watchdog.c

```c
#include <linux/module.h>
#include <linux/kernel.h>

int watchdog_init(void)
{
    printk("Initializing Watchdog module\n");
    return 0;
}

void watchdog_uninit(void)
{
    printk("UnInitializing Watchdog module\n");
}

#ifdef MODULE
int init_module(void)
{
    return watchdog_init();
}

void cleanup_module(void)
{
    watchdog_uninit();
}
#endif
```

Note that the value returned by the init_module function must be 0, for a successful 'insmod' command. Any other value will fail this command. Every module can either be linked with the kernel or loaded dynamically, using the 'insmod' command. This is why we have the code under #ifdef MODULE that is only used when this is a dynamic module. When the kernel is linked with the module, it calls the watchdog_init() function directly.

5.7.3 OpenRG's Generic Char Device

OpenRG provides a generic char device that enables the communication between the user mode process and the kernel module. Using this device requires that we register it in the
module `init` function and unregister it in the module `uninit` function. Below are the lines that we need to add in order to make this module use the generic char device. First, we include the generic char device API:

```c
#include <kos/kos_chardev.h>
```

Every generic char device defines a structure with its type and the various function callbacks available. In the Watchdog kernel module, we only use the IOCCTL entry, which provides a pointer to a function being called every time an IOCCTL is generated. The structure looks as follows:

```c
static kos_reg_chardev_t watchdog_dev = {
    type: KOS_CDT_WATCHDOG,
    ioctl: do_ioctl,
};
```

To register the char device we need to add the following lines to the `watchdog_init` function:

```c
if (kos_chardev_register(&watchdog_dev))
    return -1;
```

We also need to unregister the char device in the `watchdog_uninit` function:

```c
kos_chardev_unregister(&watchdog_dev);
```

In the next steps, all we have left to implement are the `do_ioctl` function that provides communication with the user mode process and the kernel module logic.

### 5.7.4 Implementing a IOCCTL

The IOCCTL function of the kernel module provides communication between the user mode applications and the kernel. Therefore, you must create an API for the user mode application. The API is defined in the `watchdog_mod.h`—the same file that was exported in the Makefile of the kernel module directory. The module supports the following IOCCTL commands:

**Keep Alive** A signal that is being transmitted from the user mode application. If this signal is not transmitted within a certain time margin, the module will reboot OpenRG.

**Signal Margin** Sets the time margin between two signals that are transmitted from the user mode application.

**Boot Message** Sets the message that is printed on the OpenRG console before it reboots.

Use the following lines to create the API:

```c
rg/pkg/watchdog/module/watchdog_mod.h
```

```c
#define WATCHDOG_IOCTL_KEEPALIVE _IO(RG_IOCTL_PREFIX_WATCHDOG_TUTORIAL, 1)
#define WATCHDOG_IOCTL_SIG_MARGIN _IOW(RG_IOCTL_PREFIX_WATCHDOG_TUTORIAL, 2, int)
#define WATCHDOG_IOCTL_BOOT_MSG _IOW(RG_IOCTL_PREFIX_WATCHDOG_TUTORIAL, 3, msg_t)
```

Following is a line-by-line explanation of the above API:
Lines 1-2,13 This combination of `#ifndef` and `#define` of the API’s file name is a common technique that allows including this file successfully more than once.

Line 5 Includes the Watchdog `mgt` library that defines the constants and data structures for the Watchdog package.

Lines 7-11 The actual function command codes provided by the IOCTL function are defined here. The IOCTL number should be unique, and it is created from the base and the second argument of the various `_IO` macros. The `_IO` macro indicates an IOCTL command with no arguments. The `_IOR` macro indicates an IOCTL command that reads data from the kernel, while the `_IOW` writes data to the kernel. Please refer to `rg/pkg/include/rg_ioctl.h` for more information regarding how to define the IOCTL numbers.

We will now implement the `do_ioctl` function. According to the command code, `cmd`, we need to perform the required function. The code below shows all the functions except for the keep-alive signal which is described later in the Watchdog logic. We need to add these lines right after the include lines in `watchdog.c`:

```
#include <watchdog/module/watchdog_mod.h>

static int sig_margin = DEFAULT_MARGIN;
static char boot_msg[MAX_BOOT_MSG + 1];

static int get_sig_margin(void)
{
    return sig_margin;
}

static void set_sig_margin(int new_margin)
{
    sig_margin = new_margin;
}

static char *get_boot_msg(void)
{
    return boot_msg;
}

static void set_boot_msg(char *new_msg)
{
    strncpy(boot_msg, new_msg, MAX_BOOT_MSG);
    boot_msg[MAX_BOOT_MSG] = 0;
}

static int do_ioctl(kos_chardev_t *context, unsigned int cmd,
                    unsigned long data)
{
    msg_t bmsg;
    int margin;

    switch (cmd)
    {
    case WATCHDOG_IOCTL_KEEPALIVE:
        printk("Got keep-alive signal\n");
        /* Perform Watchdog logic */
        return 1;
    case WATCHDOG_IOCTL_SIG_MARGIN:
        printk("Set new margin\n");
        copy_from_user(&margin, (void *)data, sizeof(int));
        set_sig_margin(margin);
        /* Perform Watchdog logic */
        return 1;
    case WATCHDOG_IOCTL_BOOT_MSG:
        printk("Set new boot message\n");
        copy_from_user(&bmsg, (void *)data, sizeof(msg_t));
    ```
For both IOCTL command codes, WATCHDOG_IOCTL_SIG_MARGIN and WATCHDOG_IOCTL_BOOT_MSG, the do_ioctl function receives its parameters via the 'unsigned long data' argument. The caller must provide the right data type appropriate to the specific IOCTL. The IOCTL function copies the data from the user space using the copy_from_user function.

Initiate a IOCTL from the user mode as follows:

```c
int openrg_module_ctrl(kos_chardev_type_t type, int cmd, void *data);
```

- **type** Indicates the device driver to which the IOCTL is called. It is defined in `rg/pkg/include/kos_chardev_id.h`
- **cmd** A IOCTL number as defined in the header file of the module
- **data** An address to a static variable within the user-mode application

At this stage, you can configure the Watchdog kernel module parameters. The next stage is implementing the Watchdog logic.

### 5.7.5 Watchdog Logic

Upon module initialization, we install a timer according to the preset margin. Every time we get a keep-alive signal and whenever the keep-alive margin is changed, we reset that timer. If the timer expires, meaning we didn't get a keep-alive signal within the certain time frame defined by the margin, we reboot OpenRG. We must also remember to disable the timer if the module is removed by the 'rmmod' command. To implement this logic, we need to:

1. Define a timer variable.
2. Invoke `watchdog_start` in the `watchdog_init` function to set the timer.
3. Remove the timer in the `watchdog_uninit` function.
4. Update the timer when we get the keep-alive signal.
5. Write `watchdog_cb` to handle the system reset when the timer expires.

The following code is an example how this is done:

```c
#include <linux/reboot.h>
static struct timer_list watchdog_timer;
...
static int do_ioctl(kos_chardev_t *context, unsigned int cmd,
```
unsigned long data)
{
...    case WATCHDOG_IOCTL_KEEPALIVE:
        printk("Got keep-alive signal\n");
        mod_timer(&watchdog_timer, jiffies+(get_sig_margin()*HZ));
        return 1;
    case WATCHDOG_IOCTL_SIG_MARGIN:
       ...
        mod_timer(&watchdog_timer, jiffies+(get_sig_margin()*HZ));
        return 1;
...    }
}
static void watchdog_cb(unsigned long data)
{
    printk(KERN_CRIT "Watchdog: %s\n", get_boot_msg());
    machine_restart(NULL);
}
static void watchdog_start(void)
{
    set_boot_msg("Initiating system reboot.");
    watchdog_timer.function = watchdog_cb;
    watchdog_timer.data = 0;
    watchdog_timer.expires = jiffies + HZ*get_sig_margin();
    init_timer(&watchdog_timer);
    add_timer(&watchdog_timer);
}
static void watchdog_stop(void)
{
    del_timer(&watchdog_timer);
    ...    int watchdog_init(void)
    {
    ...        watchdog_start();
    ...    }
    void watchdog_uninit(void)
    {
    ...        watchdog_stop();
    ...    }

The file linux/reboot.h contains the API of the machine_restart function. The kernel timer’s API is defined in file linux/timer.h that is included by the kos/kos_chardev.h file.

### 5.7.6 Compilation and Loading

To compile the Watchdog kernel module we must first run 'make config' in the module directory to export the Watchdog module API. Then, we can run the 'make' command which compiles and links the kernel module. The results should be in rg/build/pkg/watchdog/module/watchdog_mod.o.

```
$ cd rg/pkg/watchdog/module
$ make config
$ make
```

The module can be transferred to the board and loaded using the 'insmod' command. When compiling from the top of the tree, the module will be copied to the image /lib/modules directory. The next section explains how to load the module automatically with OpenRG’s startup process.
5.7.7 Loading the Kernel Module During Boot Sequence

On normal operation of the OpenRG system, we would like the Watchdog kernel module to be automatically loaded and initialized during the boot sequence. Loading the kernel modules on boot time is handled by the main task in `rg/pkg/main`. As we have mentioned before, we can either configure each kernel module to either load dynamically using 'insmod' or be linked in with the kernel. To support these options, the make of `rg/pkg/main` compiles the host program `gen_load_kernel_modules.c` and runs it (refer to Section 13.1).

This program generates `rg/pkg/main/load_kernel_modules.c` and `rg/pkg/main/load_kernel.c`. In case we use the dynamic load, the module load operation is added to the first generated file. Otherwise, it is added to the second generated file. To load the Watchdog module dynamically during boot sequence, all we need to do is add the following line to the program `gen_load_kernel_modules.c` in the function `create_output`:

```c
load_user_mode("watchdog_mod", 0, !CAN_FAIL, "CONFIG_RG_TUTORIAL", NULL);
```

The !CAN_FAIL indicates that if loading the kernel module had failed, OpenRG's startup process will fail and reboot the system. Instead of using the CONFIG_RG_TUTORIAL, you should use a defined configuration flag such as CONFIG_OPENRG. The `load_user_mode` is used when the module is built as a stand-alone kernel module. It is also possible to build a module that will be linked with the kernel. For this kind of module you should use the JMK_MOD_2_STAT variable in the Makefile and the `load_kernel_module` here.

5.7.8 Lesson Summary

Going through the steps described above, you have learnt:

- How to create a kernel module.
- Use OpenRG's generic char device.
- Implement an IOCTL.
- Declare timer and callback functions.
- How to add a kernel module to the boot sequence.

For a complete reference regarding the files used in this stage, refer to this tutorial's appendix (Section 5.8).

5.8 Tutorial Appendix

The following table depicts the files used in this tutorial, explaining both their general purpose and their role in the Watchdog example.
<table>
<thead>
<tr>
<th>File Name</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>watchdog_page.h,</td>
<td>The Watchdog header and implementation files. These contain the print</td>
</tr>
<tr>
<td>watchdog_page.c</td>
<td>and scan function prototypes.</td>
</tr>
<tr>
<td>wbm_pages.h, wbm_pages.c</td>
<td>The Web-based management files for registering management pages. Used</td>
</tr>
<tr>
<td></td>
<td>when registering the Watchdog page.</td>
</tr>
<tr>
<td>wbm_db.c</td>
<td>The Web-based management main database file. When registering the</td>
</tr>
<tr>
<td></td>
<td>Watchdog page, this file is used to add a binding reference.</td>
</tr>
<tr>
<td>wbm_types.h</td>
<td>The Web-based management database declaration file. Contains the</td>
</tr>
<tr>
<td></td>
<td>structure for icon enumeration, to which the Watchdog's icon is</td>
</tr>
<tr>
<td></td>
<td>added.</td>
</tr>
<tr>
<td>rg_conf_entries.h</td>
<td>Defines the configuration database entries. Watchdog entries are</td>
</tr>
<tr>
<td></td>
<td>already included.</td>
</tr>
<tr>
<td>gen_rg_def.c</td>
<td>Generates the default values of the configuration database entries</td>
</tr>
<tr>
<td></td>
<td>that run on the host. In the Watchdog example, you can set specific</td>
</tr>
<tr>
<td></td>
<td>default values using this file.</td>
</tr>
<tr>
<td>wbm_util.h</td>
<td>Defines the prototypes of the Web-based management functions. In the</td>
</tr>
<tr>
<td></td>
<td>Watchdog example, this file defines the extraction structure that</td>
</tr>
<tr>
<td></td>
<td>holds input data from the user.</td>
</tr>
<tr>
<td>jmk/env_root.mak</td>
<td>Holds OpenRG's environment makefile definition.</td>
</tr>
<tr>
<td>jmk/rg_root.mak</td>
<td>The OpenRG main makefile.</td>
</tr>
<tr>
<td>mgt_utils.h</td>
<td>Declares OpenRG's Web-based management functions. mgt_utils.h and</td>
</tr>
<tr>
<td></td>
<td>mgt_utils.c are local-management wrapper files that you must copy to</td>
</tr>
<tr>
<td></td>
<td>your working directory.</td>
</tr>
<tr>
<td>mgt_utils.c</td>
<td>Holds OpenRG's function implementation. This file, along with</td>
</tr>
<tr>
<td></td>
<td>watchdog_proc.c, are the source files you must compile into object</td>
</tr>
<tr>
<td></td>
<td>files and link to create the Watchdog application.</td>
</tr>
<tr>
<td>run_ext_proc.c</td>
<td>Implements an external process execution hook. In the Watchdog</td>
</tr>
<tr>
<td></td>
<td>example, adding code to this file loads the application during the</td>
</tr>
<tr>
<td></td>
<td>boot sequence.</td>
</tr>
<tr>
<td>gen_load_kernel_modules.c</td>
<td>Generates the file that implements the insmod commands. Used to</td>
</tr>
<tr>
<td></td>
<td>automatically load and initialize the the Watchdog kernel module</td>
</tr>
<tr>
<td></td>
<td>during the boot sequence (refer to Section 13.1) .</td>
</tr>
<tr>
<td>load_kernel_modules.c</td>
<td>Implements all insmod commands. The Watchdog kernel module load</td>
</tr>
<tr>
<td></td>
<td>operation is added to this file when using dynamic load (refer to</td>
</tr>
<tr>
<td></td>
<td>Section 13.1) .</td>
</tr>
<tr>
<td>load_kernel.c</td>
<td>Links the kernel module with the kernel. The Watchdog kernel module</td>
</tr>
<tr>
<td></td>
<td>load operation is added to this file when dynamic load is not used</td>
</tr>
<tr>
<td></td>
<td>(refer to Section 13.1) .</td>
</tr>
<tr>
<td>mt_main.h</td>
<td>Main Task's main declaration file. Used when declaring the Watchdog</td>
</tr>
<tr>
<td></td>
<td>open function.</td>
</tr>
<tr>
<td>File</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mt_main.c</td>
<td>Main Task's main implementation file. Used when adding the Watchdog <code>open</code> function in order for the entity to run when OpenRG loads.</td>
</tr>
</tbody>
</table>

**Table 5.1 Files Used in the Watchdog Example**
6

Board Tailoring

Note: All references to factory settings and to configuration files in this section are valid only for the OpenRG configuration. Since you are using the LSP configuration, you may ignore these references.

6.1 Overview

The Flash memory is segmented logically into sections, as depicted in Figure 6.1. The sequential ordering of the sections displayed here is not mandatory, however it is recommended that the boot loader and factory settings sections be placed at the beginning of the Flash memory area. By doing so you will reduce wasted Flash space, since the sectors towards the beginning of the Flash are smaller than those that come later. Note that all numerical fields in this section are in network byte order. You can also learn from Figure 6.1 when each sector can be burnt or written to, whether during run-time or production.
6.2 Flash Sections

OpenRG defines the following Flash section types, as depicted in `flash_layout.h`:

```c
typedef enum {
    FLASH_SECT_BOOT = 0,
    FLASH_SECT_FACTORY = 1,
    FLASH_SECT_IMAGE = 2,
    FLASH_SECT_CONF = 3,
} flash_section_type_t;
```

These Flash sections and their uses are described in the following sections.

6.2.1 Boot

The bootstrap and the boot loader share the same section (sectors), and must be written together. The boot loader and bootstrap object code should be position independent. However, it is possible to avoid Position Independent Code (PIC) restrictions on some platforms. Refer to your Installation Guide for relevant platform-specific information. The bootstrap and boot loader Flash sections can be written at run-time. Because these sections are not backed up, run-time burning is done only as a last resort and these sections are not normally updated during the remote upgrade procedure.
6.2.2 Bootstrap

The bootstrap is the assembly code responsible for setting up the hardware. For example, it is responsible to set up the DRAM controller, clock settings and MMU settings. Some architectures do not map Flash memory, making the CPU responsible for copying the bootstrap section to the Flash and running it. Therefore your bootstrap section should not occupy more than 4KB. After hardware initialization, the bootstrap loads the boot loader into the memory and jumps to it (or runs it directly if the Flash is memory-mapped). The boot loader code is written in 'C', so the bootstrap section must prepare the required run-time environment (stack pointer, etc.). On single image systems, the bootstrap also acts as a simple boot loader. Instead of running the boot loader, it loads a pre-defined image to the memory and jumps to it (or runs it directly if the Flash is memory-mapped). In this case, the bootstrap can be a part of the image itself.

6.2.3 Boot Loader

The boot loader is responsible for loading the software image. The boot record counter is used to identify the newest (active) image. The section whose counter has the greatest value is the active one. Before checking the counter's value, the checksum must be checked to verify the validity of the boot record and image. The boot loader then copies the image to the RAM (if needed) and jumps to the beginning of the image. The image Flash offset is passed to the kernel in one of the CPU registers. Some platforms do not have the RAM mapped in Flash memory. In this case, a platform-specific Flash read function should be implemented. This function will translate Flash read operations with "virtual" Flash addresses to low level platform-dependent operations.

6.2.4 Factory Settings

The factory setting Flash section contains static vendor-specific information, such as MAC addresses or CableHome PS. Factory settings data is stored in a set tree. For more information on factory settings refer to Chapter 8.

6.2.5 Image #1 ... #N

In multi-image architectures, each image section contains information about an image and the image itself. A new image will automatically be loaded to an IMAGE section that has the least counter value. Note that when burning the new image using the flash load command, the target section can be selected manually, as in the following example:

```
OpenRG> flash load -u tftp://192.168.1.10/openrg.img -s <section's number>
```

After the image is burnt to the target section, its counter value is increased up to a number that is greater than the value of the previously highest counter by one.
6.2.6  
rg_conf #1, #2 ... #N

This section contains the \texttt{rg\_conf} settings. To learn more about the \texttt{rg\_conf} file, refer to Section 18.1.

6.2.7  
Creating a Section for Permanent File Storage Using JFFS2

After installing OpenRG, some platforms have enough space on their Flash memory for file storage. However, it will be difficult to save and manage files on the Flash, as long as it does not contain a file system. You can solve this problem by compiling an OpenRG image with the \textit{Journalling Flash File System version 2 (JFFS2)}. When this OpenRG image is burnt to the Flash, it creates an additional section containing the JFFS2 file system, in which you can permanently store files.

There are two procedures for adding a JFFS2-formatted section to the Flash. The following are guidelines for choosing the most suitable one for your platform:

- Use the \textbf{first} procedure only if your distribution is based on Intel IXP425 (and has its own MTD module).
- Use the \textbf{second} procedure if your distribution does not have its own MTD module (but uses the generic one instead).

\textbf{Procedure 1:}

1. Add the following string to \texttt{rg/pkg/build/dist_config.c}, under your distribution's section:

   \begin{verbatim}
   token_set("CONFIG_JFFS2_FS", "m");
   \end{verbatim}

2. Remove the following string from \texttt{rg/pkg/build/dist_config.c}, under your distribution's section:

   \begin{verbatim}
   token_set("CONFIG_MTD_PHYSMAP", "y");
   \end{verbatim}

3. Add the following statement to \texttt{rg/pkg/build/hw_config.c}, under your distribution's section:

   \begin{verbatim}
   if (token_get("CONFIG_JFFS2_FS"))
     token_set("CONFIG_IXP425_FLASH_USER_PART", "0x00100000");
   /* In this case a 1MB space ("0x00100000") will be defined.*/
   \end{verbatim}

4. Define the new section in OpenRG's Flash layout, by adding the "User space" definition with a correct offset address to \texttt{rg/build/pkg/perm_storage/flash_layout.c}:

   \begin{verbatim}
   { /* User space */
     offset: 0x00F80000 ,
     size: 0x00100000 , /* 1MB */
     type: FLASH_SECT_USER,
     flags: FLASH_SECT_NO_HEADER | FLASH_SECT_WRITE_WARNING,
   },
   \end{verbatim}
5. Format the 'user' partition by creating a file with all bits set to the same size as the 'user' partition. For example, assuming your new partition's size is 640Kb, create the file as follows:
```
cat /dev/zero | tr \0 \377 | dd count=640 bs=1k of=<640Kb_file>
```

6. Load the file to the 'user' partition, using the `load` command. For example:
```
load -u tftp://192.168.1.10/<640Kb_file> -s 7
```

7. Compile an OpenRG image and burn it to the board, as described at the end of this section.

**Procedure 2:**

1. Add the following string to `rg/pkg/build/dist_config.c`, under your distribution's section:
```
token_set("CONFIG_JFFS2_FS", "m");
```

2. Add the following string to `rg/pkg/build/hw_config.c`, under your distribution's section:
```
token_set_y("CONFIG_ZLIB_DEFLATE");
```

3. Define the new section in OpenRG's Flash layout, by adding the "User space" definition with a correct offset address to your platform's Flash layout file.
```
{ /* User space */
  offset: 0x00F20000,
  size:   0x000A0000, /* 640Kb */
  type:   FLASH_SECT_USER,
  flags:  FLASH_SECT_NO_HEADER | FLASH_SECT_WRITE_WARNING,
}
```

The minimum size of this section equals the total size of two physical Flash blocks. For more information about the block size on your Flash, refer to the Flash chip specification document.

**Note:** The new section's space is created at the expense of one of the Flash sections.

4. When using Linux 2.4:
   Add an MTD partition in `os/linux-2.4/drivers/mtd/maps/physmap.c`. The MTD partition must be of the same size and offset as the 'user' partition on the Flash.
```
{ 
  name: "user",
  offset: 0x003A0000,
  size: 0x00060000,
}
```

When using Linux 2.6:
Perform the following change in `pkg/kernel/linux/boot/rg_prom.c`. In function `openrg_commandline_parse()`, specify the size@offset for the partition (which must be the same as in your flash layout):
```
sprintf(cmdline + strlen(cmdline),
    "mtdparts=phys_mapped_flash:0x%lx@0x%p(\d%s),0x%lx@0x%p(\d%s), "
    "0xa0000@0xf20000(JFFS2)", CONFIG_MTD_PHYSMAP_LEN, RG_PROD_STR,
```

**Notes:**
JFFS2 partitions must have a minimum of 5 erase blocks. For example, if the flash sector size is 128KB, the JFFS2 partition size must be at least 128KB*5=640KB.

The new partition must start and end in the erase block boundaries. For example, if the sector size is 128KB, the flash section start and end offset should be a 128KB multiply.

5. Format the 'user' partition by creating a file with all bits set to the same size as the 'user' partition. For example, assuming your new partition's size is 640Kb, create the file as follows:

```
cat /dev/zero | tr '0' '\377' | dd count=640 bs=1k of=<640Kb_file>
```

6. Load the file to the 'user' partition, using the `load` command. For example:

```
load -u tftp://192.168.1.10/<640Kb_file> -s 7
```

To create an OpenRG image with either of the mentioned configurations, proceed as follows:

1. Clean the distribution:

```
$ make distclean
```

2. Reconfigure your development tree:

```
$ make config DIST=<distribution name in the upper case>
```

3. Build the new OpenRG image:

```
$ make
```

After the image is compiled, burn it to the board's Flash. When booting, OpenRG creates a `mnt/jffs2` directory in the new section. This directory is the mount point of JFFS2. After the file system has been mounted, you can save files to the `mnt/jffs2` directory.

### 6.3 Initial Flash Image Builder

The `build_flash` utility creates `flash.img`. This utility is used on the host machine to build the initial Flash layout, including all required (factory written) sections, together. This utility is universal to all platforms and uses the Flash layout definition file that corresponds to the target platform. This utility locates existing files for each section, and combines them into one file that can be burnt to the Flash. The section information blocks are built and written by the utility. The `build_flash` command takes the following parameters:

- `-b <boot>` Name of file that contains bootstrap and boot loader image.

- `-f <factory settings>` Name of file that contains factory settings.

- `-i <image>` Name of file that contains an image. This parameter can be specified multiple times (once for each image file).
6.4 Designing a Correct Flash Layout

6.4.1 Flash Layout Example

Under the vendor/<vendor_name> subdirectory there are flash_layout.c files according to size:

- flash_layout.c – 8 MB
- flash_layout_16mb.c – 16 MB
- flash_layout_32mb.c – 32 MB

Each of these files includes a Flash layout array of type flash_section_layout_t:

```c
typedef struct {
    u32 offset;
    u32 size;
    flash_section_type_t type;
    u32 flags;
} flash_section_layout_t;
```

Each sector has the following values:

- **offset**  The desired byte offset into the array in which the sector’s data will be written.
- **size**  The size in bytes of the sector in hexadecimal representation.
- **type**  The sector type, which can be any one of the following values:

  ```c
typedef enum {
    FLASH_SECT_UNDEF = -1,  
    FLASH_SECT_BOOT = 0,   
    FLASH_SECT_FACTORY = 1, 
    FLASH_SECT_IMAGE = 2,  
    FLASH_SECT_CONF = 3,   
    FLASH_SECT_BOOTCONF = 4, 
    FLASH_SECT_USER = 5, 
    FLASH_SECT_FLASH_LAYOUT = 6, 
} flash_section_type_t;
```

- **flags**  The sector flags (optional). The sector can have none, one or more of the following flags:

  - FLASH_SECT_WRITE_WARNING
  - FLASH_SECT_NO_HEADER
  - FLASH_SECT_BACK_HEADER
  - FLASH_SECT_VENDOR_HEADER

The following is an example of code for the first sector:

```c
{"/ * Boot section bootstrap + bootloader */
```
Note that the next sector will start at offset 0x000A0000, which is the sum of the previous offset and the previous size.

**Notes:**

1. The size of the Flash sector should be multiplications of the Flash writing block.
2. It is preferable not to place the boot sector directly before or after the image sector, in case of a writing error.

### 6.4.2 Dynamic Flash Layout

The dynamic Flash layout feature enables you to modify the Flash layout throughout the production. We recommend using the dynamic Flash layout ONLY if the Flash size will be changed after deployment. There is usually no need for using this feature. The Flash has a designated sector to which you can burn the Flash layout. RGLoader and the OpenRG image search for the Flash layout section at boot time. If it exists, the Flash layout section is loaded and used as the layout. If it does not exist, the RGLoader and the OpenRG image will use the layout with which they were compiled. OpenRG searches for the Flash layout section in the following places:

\[
(M/2) - K, \\
1 * M - K, \\
2 * M - K, \\
4 * M - K, \\
7 * M - K, \\
8 * M - K, \\
16 * M - K, \\
32 * M - K, \\
64 * M - K, \\
128 * M - K, \\
\]

If you wish to use the dynamic Flash layout, compile your image with the following command:

```bash
make config DIST=UML LIC=/rg/jpkg_uml.lic CONFIG_RG_DYN_FLASH_LAYOUT=y && make
```

The `token_set_y("CONFIG_RG_DYN_FLASH_LAYOUT")` flag creates the `LAYOUT.sec` file, which can be used to load to the board using:

```bash
OpenRG> flash load -u tftp://<IP>/<LAYOUT> -s <section_number>
```

### 6.5 Hardware Button Customization

Hardware buttons can provide an additional, software-independent way in which to implement useful OpenRG features. For example, a user can return to the default factory settings using the Web-based management. By implementing the "restore defaults and reset" button, we provide the user with a way to perform this same function, even if he cannot access the WBM.
6.5.1 Overview

A hardware button is a hardware unit that generates interrupts when pressed, and sometimes when released. Hence, the code that handles the button signals is in the kernel. The code that actually performs the desired action, however, is in the user mode—in Main Task. The standard driver API and the `select()` system call are used for communicating between these two blocks of code.

![Figure 6.2 Hardware Button Logic](image)

Following is a run-through of the interaction between the kernel and user modes when a hardware button is pressed:

1. **Kernel mode** The `hw_button` driver consists of two parts—architecture-independent code (`module_init()`, `module_close()`, `dev_open()`, `dev_poll()`, `dev_read()`) and architecture-dependent code (the button handler). The architecture-dependent code implements the defined API to work with the independent code.

2. **Kernel mode** The driver registers its file operation within `rg_major` device (minor `HW_BUTTON`, node name "/dev/openrg.hw_button").

3. **User mode** To access button information, the user mode code opens the `hw_button` device and calls `select()` for the received file descriptor on read operations.

4. **Kernel mode** The `hw_button` driver's `poll()` function calls `poll_wait()`, to wait for information from any existing button handler.

5. **Kernel mode** On button interrupt, the button handler calls the `btn_action_notify()` function (which calls `wake_up_interruptible`). `btn_action_notify()` is
implemented for multi-OS compatibility. \texttt{btn\_action\_notify()} parameters are the button ID and state (BTN\_PRESSED, BTN\_RELEASED).

6. **Kernel mode** The \texttt{btn\_action\_notify()} function adds the event to the \texttt{hw\_button} 's driver events queue. Along with the button ID and state, the OS timer value (jiffies) is saved.

7. **Kernel mode** After \texttt{poll\_wait()} has returned, the \texttt{poll()} function sets the \texttt{file\_operations} structure mask field to POLLIN, to inform user mode that the read operation is permitted.

8. **User mode** The user mode code reads a \texttt{btn\_action\_t} variable from the driver's file descriptor, to get the button status. It acts according to the role of the button (reset, restore defaults, etc.).

9. **Kernel mode** The driver's \texttt{read()} function removes the read event from the driver's events queue.

The kernel mode is implemented in the \texttt{hw\_buttons.c} and \texttt{hw\_buttons.h} files, while the user mode is implemented in \texttt{uievents}.

### 6.5.2 VxWorks Considerations

In VxWorks, a callback for the \texttt{select()} call is implemented in a quite similar way. The same actions are taken—a driver's function waits on a queue, and an ISR informs about the event by releasing the queue lock. Instead of \texttt{poll()} callback, VxWorks implements \texttt{select()} code in \texttt{iocl()} callback. The logic for releasing the lock is hidden from the ISR by the \texttt{btn\_action\_notify()} function.

### 6.5.3 Restore Default Settings Design

The idea behind the 'Restore Defaults' feature is to provide a user with a software-independent way to return to the factory settings values. It is used, for example, when a user has forgotten his password to the system. Restoring the default settings gives him access to OpenRG again. To restore defaults, the user presses the 'Restore Defaults' button. The button is defined for each platform independently. Some platforms may not have this feature. The BTN\_ID\_1 button events, defined as BTN\_ROLE\_RESET, are used for this feature. According to a state of the button, OpenRG implements the following actions:

- If the button is pressed, the \texttt{sys\_time} value is saved.
- If the button is released, the saved value is compared to the current time. If the time interval exceeds the predefined value, the default settings are restored. In any case, a restart command is issued.

To restore the defaults, the user should press the button for at least 3 secs.

```c
#define BTN\_ROLE\_RESET BTN\_ID\_1
```
6.6 LED Control

A variety of events need to be signaled via LEDs. Detection and monitoring of events, the way they are reported, and physical access to LEDs are platform/vendor-specific. There is a generic mechanism for controlling LEDs, divided into user mode (UserSpace) and kernel mode (LED module). The LED control code is already in your distribution, under the pkg/kernel/common/kleds/ directory. The module is separated into two parts:

1. kleds.c – An interface from OpenRG and this module.
2. plf_leds.c – A file in which you should put your own LED control code.

6.6.1 LED Module

The low-level code for the handling of LEDs is highly platform-specific. The low-level code provides an interface for user mode applications via openrg_module/ioctl. It is possible to implement the primitive on, off and toggle LED operations. The low-level LED module can be built and loaded conditionally. The LED module consists of the following:

1. plf_leds.h – This .h file will be included both in the module and in OpenRG. This module can be used to control different LEDs. Create a definition indicating the LED you are going to control, for example:

   ```c
   typedef enum {
       LED_SYS = 0,
       LED_COUNT = 1,
   } led_t;
   ```

2. plf_leds.c – This .c file controls the LED module.
   - leds_init() will be called when inserting the module.
   - leds_cleanup() will be called when removing the module.
   - leds_op() will be called whenever the userspace program uses openrg_module_ctrl.

   ```c
   printk("LED:
   ```

   You can add debugging code to check whether the parameter is passed successfully to the module:

   ```c
   return 0;
   ```

   You can also switch LEDs:

   ```c
   ```

   You can add more instructions to this module, enabling you to further interact with the hardware.

3. kleds.c, kleds.h – These files are used as abstract code for controlling the LEDs. You do not need to modify these files.
6.6.2 UserSpace

The UserSpace is the OpenRG side of the module. A separate (platform-independent) task is dedicated to event detection. It listens for device-status changes and activity on behalf of vendor-specific code. Vendor-specific code is linked to the Main Task. When given a chance to run, it subscribes for event notifications it is interested in. Upon event notification it signals them as appropriate. The vendor task could also prioritize events, e.g. "got IP" might take precedence over traffic signaling for PPP, given only one LED.

1. Add the following code to where you want to control the LED:

```c
#include <kernel/common/kleds/kleds.h>
kleds_req_t req;
MZERO(req);
req.led = LED_SYS;
req.op = LED_OP_ON;
openrg_module_ctrl(KOS_CDT_KLEDS, KLEDS_CMD_LED_OP, &req);
```

2. led_control.c

- This file is used to test the LED control module in UserSpace.
- This file also contains an example to control the blinking of a LED. Refer to the `blink_led()` function.
- Place the file in the directory, and change the Makefile:

```bash
JMK_TARGET=ledc
JMK_O_OBJs=ledc=led_control.o
```
Porting OpenRG to a Different Architecture

OpenRG can be ported to various hardware platforms. The porting process might require changes to OpenRG’s distribution, platform definition and initialization, boot sequence, memory configuration and I/O configuration.

The following section describes the OpenRG porting procedure, which is relevant for all platforms. Still, every platform has its own unique characteristics and may require different modifications. The rest of this chapter provides you with general porting tips, and with instructions for porting OpenRG to Intel’s IXP425-based reference design platforms. These platforms include the IXDP425 (also known as Richfield) and the Coyote platforms.

7.1 Porting Guidelines

Porting OpenRG to a certain architecture is basically a merge between the OpenRG codebase and a Board Support Package (BSP) provided by the vendor. A BSP consists of Linux kernel and drivers, which the vendor has adapted for a specific hardware platform. The source code of such Linux contains a number of changes made to meet the requirements of the platform’s various devices. To review these code changes, you must also obtain a standard Linux codebase of the same version as the supplied BSP. Note that successful porting largely depends on your careful review of the BSP’s drivers and kernel APIs.

7.1.1 Porting Prerequisites

To start porting OpenRG to a BSP, you will need the following:

Hardware
- A hardware debugger and scripts (if available)
- Boards with Linux already burnt to the Flash
- A removable Flash (if available)
- A JTag interface – a special socket on the board used for connecting a debugging device.

**Software**

- A platform-specific toolchain
- A user mode burning utility
- The following source code:
  - A full Linux 2.4/2.6 BSP
  - Clean Linux codebase of the same version as the BSP (for Diff generation)
  - A compilable bootloader, including bootstrap code
- BSP drivers for external peripherals (such as Flash, Serial, Ethernet, ADSL, DSP, etc.)

**BSP Documentation**

- Board schematics
- Hardware specification
- Programmer's guide

In addition, verify that the BSP code compiles properly. After compiling the BSP image, burn it to the board's Flash using a hardware debugger or a bootloader. When Linux is up and running, perform sanity checks of network and serial connections, DSP, etc.

### 7.1.2 Reviewing Code Differences

After verifying that the BSP you received is errorless, use a `diff` command to analyze the code differences between the clean Linux codebase and the BSP. You should check the contents of the BSP's new directories, as well as examine modified or new drivers. In addition, any modifications in the generic kernel code must be carefully reviewed and their purpose understood.

Ideally, all the BSP source code modifications must be concentrated in the following new directories:

- **ARM architecture:**
  - arch/arm/mach-<...>
include/asm-arm/arch-<...>

**MIPS architecture:**
- arch/mips/<...>
- include/asm-mips/mach-<...>

New driver directories

However, changes may also exist in such directories as arch/<...>, include/asm-<...>, net, kernel, fs, drivers, include/linux, include/net.

As the next step, examine the usage of default and new configuration flags, in order to properly map various parts of the BSP and OpenRG code. Note that a Linux 2.6-based BSP contains the `Kconfig` textual file, which explains the purpose of each configuration flag. In case you do not understand some part of the BSP code, you may look for it in a higher Linux version, or in other architectures—it is possible that this code has been merged to the BSP from there.

### 7.1.3 Merging the BSP into OpenRG

After the code changes have been reviewed, start organizing the new directory structure according to the following guidelines:

- Platform-specific code (`arch-<...>`) for CPU support and SoC drivers (such as ETH, DSL, UART, GPIO), and the `include` directory should be placed under the `vendor/<bsp_name>/kernel` directory.

- All generic external device drivers from other vendors (wireless, switch, DSP, etc.) should be placed under the `vendor/<driver_vendor>/drivers` directory.

- Import all new directories for future use, in case you would like to upgrade the distribution.

After preparing the directory structure, examine the BSP's kernel code. Remove as much non-relevant code changes as possible. Ideally, there should be no code modification within the generic kernel code (excluding Makefiles). In addition, try to replace the significant code changes with a generic or Jungo's implementation. If you fail to do so, leave the vendor's code changes intact, and protect them with configuration flags.

In addition, check if you can use the original OpenRG toolchain. This may help you to avoid toolchain bugs, libc compatibility problems (lib-gcc), etc. If you cannot leave the original toolchain intact, integrate the BSP's toolchain into OpenRG (look at the `jmk/env_root.mak` and `jmk/rg_root.mak` files as a reference).
7.1.4 Compiling an LSP Image

After merging the BSP code to OpenRG, you should compile a Linux Support Package (LSP) image, containing a minimal amount of modules. To compile the image, perform the following:

1. Determine the cross-compilation flags. To do so, compile the BSP, and look at compilation flags of its kernel and user mode. In addition, examine the `arch/<...>/Makefile`, as well as look at `pkg/build` of similar architectures. For more information about specific flags, consult the GCC Documentation.

2. Add the cross-compilation flags to `pkg/build/config_host.c`

3. Determine the kernel configuration flags, which are based on the BSP's default and auto-generated configuration flags. Use `defconfig` as a reference. Filter out the irrelevant flags, and leave a minimal set of necessary ones (Arch, ETH, UART). Later, you will be able to add more flags (Flash/MTD, WiFi, USB, etc.).

4. Carefully sort the flags into the following groups: `dist_config`, `hw_config`, and `feature_config`.

5. If necessary, make changes in `os/Makefile.vmlinux`.

6. Configure the required modules (Ethernet, Flash) according to the BSP.

7. Configure the load address, and the kernel entry:

   ARM
   - Load address – `PAGE_OFFSET` (`include/asm-arm/arch-<...>/memory.h`)
   - Kernel entry – `TEXT_OFFSET` (`arch/arm/Makefile`)

   MIPS
   - Load address – `LOADADDR` (`arch/mips/Makefile`)
   - Kernel entry – check in the BSP code

8. In case the BSP and OpenRG use incompatible Linux versions, you may try to modify the BSP drivers to fit the kernel APIs of OpenRG. Alternatively, if the code differences are concentrated in one place, modify or upgrade the relevant part of the BSP Linux APIs. However, if the differences are extensive and scattered throughout the kernel code, you may upgrade OpenRG's Linux.

   Note: It is highly recommended that you avoid performing partial or full upgrade of the kernel code.

Finally, compile the LSP image and burn it to the board. Once it is up and running, check functionality of various device drivers.
If the LSP image is functioning properly, create a full distribution according to requirements, add a Flash layout (`rg/vendor/<...>/flash_layout`) and mtd partitions. Then, compile a full image, run it, and debug if necessary.

### 7.1.5 Debugging FAQ

**Problem:** Kernel does not boot.

**Possible solutions:**

- Verify that UART is properly configured.
- Verify that the load address is correct.
- Verify that the boot-loader uses a correct load address and kernel entry point.
- Use `prom_printf` or similar (the architecture's `setup.c` or `prom.c`).
- Directly write to UART registers.
- Follow the boot sequence (`start_kernel`).
- Use a hardware debugger.

**Problem:** Kernel boots, but hangs or crashes after `console_init`.

**Possible solutions:**

- Use `Objdump` as explained in Section 20.4.6.
- Use `printk` as explained in Section 20.4.1.
- Follow the last console messages.

### 7.2 General Porting Tips

You can configure the OpenRG system depending on the amount of Flash memory available, to host one or more software images. Storing more than one image on the Flash is useful for redundancy purposes in case of failure during remote firmware upgrade. When you select the multiple images option, the system should be equipped with a boot loader, which is responsible to load the current image. The OpenRG boot loader (RGLoader) is a subset of OpenRG and if you use it on the board, you should port it first. After porting the boot loader, you should also port OpenRG, applying the changes done to the boot loader to OpenRG. The rest of this section describes the various changes you need to perform in order to port OpenRG to a new platform. The instructions are not related to a specific architecture.
7.2.1 Distribution

An OpenRG distribution consists of software features and hardware features. The distribution is defined in the pkg/build directory, which includes the following files:

1. config_opt.c – Defines the list of all available hardware platforms, distributions for these platforms, and all potential software features.

2. hw_config.c – Defines the hardware features available on specific platforms, including for example, the amount of memory, the Flash size, network devices, etc.

3. dist_config.c – Defines the software features (modules) for each distribution. For example, SNMP, PPP, Firewall, VPN, etc.

You may choose to modify the reference design distribution or create a new distribution, based on the reference design distribution. The second option is a little more work but it allows you to build both the reference design and your target board, using the same source tree.

7.2.2 Boot Sequence

OpenRG is designed to operate under various memory requirements, including stringent memory scenarios. During boot, the image is decompressed into the RAM, occupying approximately 8 MB. Flash memory is physically segmented into sectors and logically partitioned into sections. The Flash might have different size sectors. In any case, the logical sections must be aligned with the physical sectors' boundaries. The first section in the Flash memory is the boot loader, responsible for the boot process and the extraction of the compressed software image to the RAM. The OpenRG image consists of a compressed operating system and file system. The operating system is decompressed in its entirety to the RAM, while the file system is decompressed in a modular fashion according to system state, the available memory and active applications. The OpenRG boot sequence proceeds as follows:

1. The boot loader is copied from Flash to RAM if necessary.

2. The boot loader begins executing.

3. The kernel image is decompressed and copied from Flash to RAM.

4. The RAM disk is decompressed and copied from Flash to RAM.

5. The boot loader concludes by starting the kernel.

6. The kernel initializes devices, peripherals and the main kernel thread (PID 1).

7. The kernel executes /bin/init and becomes the init (user mode) process.

8. The init process adds several modules to the kernel using insmod.

9. The init process executes /bin/openrg to start OpenRG.
7.3 Intel's IXP425-based Platforms

Intel's IXP425 processor is the core of several reference designs including the IXDP425 and the Coyote boards. The porting work for these platforms is similar. This porting guide will focus on the Coyote platform. The example board is named Cute, which is based on the Coyote platform and is equipped with Ethernet LAN and Ethernet WAN.

7.3.1 Distribution

The distribution of Coyote is defined as described before in the pkg/build directory in files config_opt.c, hw_config.c and dist_config.c. In our example using the Cute board, you would need to make the following changes:

1. `config_opt.c` – Extend the array openrg_hw_options with another entry for the new board, similar to the COYOTE entry, which should be:

   ```c
   { "CUTE", "Cute" },
   ```

   In addition, add a new entry to the openrg_distribution_options array, similar to the COYOTE entry, which should be:

   ```c
   { "CUTE", NULL },
   ```

   Finally, add a new entry to the config_option_base array, which defines the new architecture of the machine you will be adding to the kernel. According to the example above, you would need to add the following:

   ```c
   { "CONFIG_ARCH_IXP425_CUTE", NULL },
   ```
2. **hw_config.c** – Add a new section that defines the specific hardware available on your board. You could copy the COYOTE section and modify it to reflect your board. The section below shows an example of how the Cute board, which has only Ethernet LAN and WAN, is defined:

```c
if (IS_HW("CUTE"))
{
    token_set_y("CONFIG_IXP425_COMMON_RG");
    token_set_y("CONFIG_ARCH_IXP425_CUTE"); /* CUTE machine */
    token_set("CONFIG_IXP425_FLASH_LAYOUT_SIZE", "16"); /* 16MB flash */
    token_set("CONFIG_IXP425_SDRAM_SIZE", "32"); /* 32MB SDRAM */
    token_set("CONFIG_IXP425_NUMBER_OF_MEM_CHIPS", "2"); /* in 2 chips */
    token_set("CONFIG_RG_CONSOLE_DEVICE", "ttyS1");
    token_set("CONFIG_IXP425_KGDB_UART", "1");

    if (token_get("CONFIG_HW_ETH_WAN"))
    {
        token_set_m("CONFIG_IXP425_ETH");
        dev_add("ixp1", DEV_IF_IXP425_ETH, DEV_IF_NET_EXT);
    }

    if (token_get("CONFIG_HW_ETH_LAN"))
    {
        token_set_m("CONFIG_IXP425_ETH");
        dev_add("ixp0", DEV_IF_IXP425_ETH, DEV_IF_NET_INT);
    }

    token_set("FIRM", "Cute_company");
    token_set("BOARD", "CUTE");

    /* Flash chip */
    token_set("CONFIG_IXP425_FLASH_E28F128J3", "m");
}
```

3. **dist_config.c** – Add a new section for each configuration of the software modules that you would like to build on that hardware. For example, you might want to build several products with different feature sets on the same hardware platform (remember to add each distribution name to the `config_opt.c` file). The following shows the definitions that will turn the Cute board into a simple Ethernet-Ethernet router with a firewall.

```c
if (IS_DIST("CUTE"))
{
    hw = "CUTE";

    token_set_y("MODULE_RG_FOUNDATION");
    token_set_y("MODULE_RG_UPNP");
    token_set_y("MODULE_RG_PPP");
    token_set_y("MODULE_RG_FIREWALL_AND_SECURITY");
    token_set_y("MODULE_RG_ADVANCED_MANAGEMENT");
    token_set_y("MODULE_RG_ADVANCED_ROUTING");

    /* HW Configuration Section */
    token_set_y("CONFIG_HW_ETH_WAN");
    token_set_y("CONFIG_HW_ETH_LAN");

    goto Exit;
}
```

The definition of the distribution includes the software modules that are included in it, and the hardware features of the board that are required. In this case, the distribution includes the foundation module, zero-configuration module, PPP module, firewall and security module, advanced management, and advanced routing module. The configured hardware devices are the Ethernet WAN and LAN network devices (which are the only hardware available on this board).
7.3.2 Platform

The IXP425 processor is based on the ARM architecture and each new platform using it should be introduced to the Linux kernel. All the source files referred to in this section are relative to the Linux kernel root directory, which is under the `os/linux-<version>` directory, for example, `os/linux-2.4`. The first step is to assign your board with a unique machine number. The machine numbers are defined in `arch/arm/tools/mach-types`, and you should add an entry describing your hardware platform to this file. The entry should be added at the end of the file. For example, the Coyote entry in this file is:

<table>
<thead>
<tr>
<th>Name</th>
<th>ARCH_IXP425_COYOTE</th>
<th>COYOTE</th>
<th>357</th>
</tr>
</thead>
<tbody>
<tr>
<td>coyote</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These names are used in various places to define the machine. The first and third columns are the platform name, where the first is in lower case and the third is in upper case. The second column should be `ARCH_IXP425_<platform>` and the last column should be a unique number. In our example, the platform name is "Cute" and we would add a line which could look as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>ARCH_IXP425_CUTE</th>
<th>CUTE</th>
<th>370</th>
</tr>
</thead>
<tbody>
<tr>
<td>cute</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The next step, is to set the machine type to register R1. The file you would need to change is `arch/arm/kernel/head-armv.S` and the lines related to the Coyote platform are:

```c
#elif defined(CONFIG_ARCH_IXP425_COYOTE)
    ldr     r1, =MACH_TYPE_COYOTE
#endif
```

When working on the Cute board, you would add the following lines to this file right before the `#else` line:

```c
#elif defined(CONFIG_ARCH_IXP425_CUTE)
    ldr     r1, =MACH_TYPE_CUTE
#endif
```

Next, you need to define the architecture's name and initialization routines. You should do this in the `arch/arm/mach-ixp425/arch.c` file as shown in the Coyote example below:

```c
#ifdef CONFIG_ARCH_IXP425_COYOTE
MACHINE_START(COYOTE, "Intel IXP425 Coyote")
    MAINTAINER("Intel - IABU")
    /* Memory Base, Phy IO, Virtual IO */
    BOOT_MEM(PHYS_OFFSET, IXP425_PERIPHERAL_BASE_PHYS, IXP425_PERIPHERAL_BASE_VIRT)
    MAPIO(ixp425_map_io)
    INITIRQ(ixp425_init_irq)
MACHINE_END
#endif
```

For the Cute example board we would add the following lines at the end of the file (simply replace every "COYOTE" with "CUTE" and every "Coyote" with "Cute"):

```c
#ifdef CONFIG_ARCH_IXP425_CUTE
MACHINE_START(CUTE, "Intel IXP425 Cute")
    MAINTAINER("Intel - IABU")
    /* Memory Base, Phy IO, Virtual IO */
    BOOT_MEM(PHYS_OFFSET, IXP425_PERIPHERAL_BASE_PHYS, IXP425_PERIPHERAL_BASE_VIRT)
    MAPIO(ixp425_map_io)
    INITIRQ(ixp425_init_irq)
#endif
```
7.3.3 Boot Sequence

7.3.3.1 Image Structure

The OpenRG image for IXP425-based platforms is constructed by the `Makefile` in directory `os/linux-2.4/arch/arm/boot`. There are several options that you might need to adjust in order to build the image that will suit your platform. The following is an excerpt from the Makefile, which describes the parameters that might be influenced by your platform available memory:

```c
ifeq ($(CONFIG_ARCH_IXP425),y)
    ZRELADDR = 0x00008000
    ZTEXTADDR = 0x00A00000
    ZBSSADDR = ALIGN(4)
    export INITRD = $(TOPDIR)/arch/arm/boot/ramdisk.gz

    ifdef CONFIG_RG_BOOTSTRAP
        export BOOTSTRAP = $(TOPDIR)/arch/arm/mach-ixp425/bootstrap.o
    endif

    ifdef CONFIG_IXP425_POST
        export POST = $(TOPDIR)/arch/arm/boot/post/post_crt.o \
                $(TOPDIR)/arch/arm/boot/post/post_mmu.o \
                $(TOPDIR)/arch/arm/boot/post/romInit.o \
                $(TOPDIR)/arch/arm/boot/post/romuart.o \
                $(TOPDIR)/arch/arm/boot/post/romstr.o
    endif

    INITRD_PHYS = 0x00A00000
    PARAMS_PHYS = 0x00002000
endif
```

In order to build an image that could bootstrap the platform, you need to turn on the `CONFIG_RG_BOOTSTRAP` option in the distribution process. The RGLoader distribution is supplied with this configuration option already turned on. In case your platform is not going to include a separate boot loader (which means you will only be able to store a single image), you should add this configuration option to your distribution. The line you should add to `dist_config.c` is:

```c
token_set_y("CONFIG_RG_BOOTSTRAP");
```

This configuration option will add the `arch/arm/mach-ixp425/bootstrap.S` file to the image created in the above Makefile. The code in this file is executed only on system bootstrap and is responsible to reinitialize the CPU, SDRAM and expansion bus. The `init.S` file, located in `arch/arm/boot/compressed`, copies `compressed/vmlinux` and `ramdisk.gz` from the Flash to the memory. It is also responsible for activating the `setup_initrd` function, which initializes the RAMDISK. The structure of the final image that is burnt to the flash memory is depicted in Figure 7.2.
7.3.3.2 Kernel Memory Layout

The following Makefile variables can be modified to control the RAM memory addresses that will be used to start up the system:

1. ZRELADDR is the address to which the final vmlinux kernel image will be decompressed by head.S. This address must be in the form 0xXXXX8000, i.e. the address must end with 0x8000. This offset is used in the linker instructions file compressed/vmlinux.lds to define the load address:

```
load_addr = LOAD_ADDR = $ZRELADDR
```

2. ZTEXTADDR defines the start address of the vmlinuz image (named piggy.gz on ARM-based platforms), i.e. the address to which head.S is loaded. It is used in compressed/Makefile and in the linker instructions file compressed/vmlinux.lds to define the _start address:

```
_start = TEXT_START = $ZTEXTADDR
```

3. ZBSSADDR defines the address of the decompressor heap. It must be defined if the decompressor runs from Flash. It defines the BSS start address in the linker instructions file:

```
BSS_START = $ZBSSADDR
```
In order to change the kernel memory layout you would need to use the following calculations. Let us define X, Y and Z as follows:

1. **X** The physical address of the beginning of the SDRAM. This will be mapped to the beginning of the linear mapping.

2. **Y** The virtual address of the beginning of the linear mapping.

3. **Z** The physical address of the SDRAM to which the RAMDISK must be copied.
   \[ Z = X + 0x2000000 \]

The following files should be modified:

1. **arch/arm/Makefile** defines the kernel virtual link address and should be set to \( Y + 0x8000 \). For example, on IXP425, \( Y \) is 0xC000000 and the TEXTADDR should be set to 0xC0008000.
   
   \[
   \text{TEXTADDR} = 0xC0008000
   \]

2. **arch/arm/boot/Makefile** defines the physical address to which the kernel should be loaded, and the physical address to which the compressor should be loaded. The first should be \( X+0x8000 \). The second should be defined so that no memory will be overrun. For example:

   \[
   \text{ZRELADDR}=0x14008000
   \text{ZTEXTADDR}=0x14080000
   \text{INITRD_PHYS}=0x16000000
   \]

3. **arch/arm/mach/arch.c** initializes the RAMDISK by calling the `setup_initrd` function with the address and size of the RAMDISK.

   \[
   \text{setup_initrd( __phys_to_virt(0x16000000), 8*1024*1024 );}
   \]

4. **include/asm/arch/memory.h** defines the kernel RAM virtual address beginning and the physical address in RAM to which it is mapped. For example:

   \[
   \text{PAGE_OFFSET} = 0xC0000000
   \text{PHYS_OFFSET} = 0x14000000
   \]

### 7.3.3.3 Debug Boot Sequence

When trying to debug the boot sequence, the following could be useful addresses at which to place breakpoints:

1. The decompressor jumps to the kernel at `call_kernel`, which could be found in file `arch/arm/boot/compressed/head.S`.

2. The CPU paging is switched on at `__ret`, which could be found in file `arch/arm/kernel/head-armv.S`.
7.3.3.4 Debugging with UART

Another useful technique is to print characters to the console, in order to indicate the boot sequence progress and determine where it had failed. Printing output to the console is simple, as all you need is to write the character to the UART register. Below is a code excerpt that sends one character to the console:

```assembly
mov   r1, #0xc8000003
mov   r0, #XX
strb  r0, [r1]
```

7.3.4 Memory

The board's memory configuration includes the SDRAM and Flash configurations.

7.3.4.1 SDRAM Configuration

The configuration of the SDRAM is done using the OpenRG configuration options. The board hardware configuration is defined in `pkg/build/hw_config.c` and should include the following configuration options to define the memory:

1. `CONFIG_IXP425_SDRAM_SIZE` should be set to the total size of the memory in MB.
2. `CONFIG_IXP425_NUMBER_OF_MEM_CHIPS` should be set to the number of chips that are used on the board.

For example, the Coyote board distribution would include the following configuration options, in order to define its 32MB of SDRAM, supplied on 2 memory chips:

```c
token_set("CONFIG_IXP425_SDRAM_SIZE", "32");
token_set("CONFIG_IXP425_NUMBER_OF_MEM_CHIPS", "2");
```

The values defined in the above mentioned configuration options are used in the `include/asm-arm/arch-ixp425/ixp425.h` file, which is part of the kernel, in order to select the correct memory configuration. The available memory configuration defined in `ixp425.h` is:

```c
#define IXP425_SDRAM_CFG_32MEG_2Chip (IXP425_SDRAM_CAS_3CLKS | IXP425_SDRAM_32Meg_2Chip)
#define IXP425_SDRAM_CFG_64MEG_2Chip (IXP425_SDRAM_CAS_3CLKS | IXP425_SDRAM_64Meg_2Chip)
#define IXP425_SDRAM_CFG_64MEG_4Chip (IXP425_SDRAM_CAS_3CLKS | IXP425_SDRAM_64Meg_4Chip)
#define IXP425_SDRAM_CFG_128MEG_2Chip (IXP425_SDRAM_CAS_3CLKS | IXP425_SDRAM_128Meg_2Chip)
#define IXP425_SDRAM_CFG_128MEG_4Chip (IXP425_SDRAM_CAS_3CLKS | IXP425_SDRAM_128Meg_4Chip)
#define IXP425_SDRAM_CFG_256MEG_4Chip (IXP425_SDRAM_CAS_3CLKS | IXP425_SDRAM_256Meg_4Chip)
```

7.3.4.2 Flash Configuration

The Flash memory is used to store the OpenRG system components, including the boot loader and its configuration, and the OpenRG image or images along with their configuration files and factory settings. The Flash memory configuration includes defining the correct type of memory chips used on the board and the Flash layout. The Flash memory driver uses the Linux MTD driver. This driver supports Flash memory chips that work according to the CFI API. The Flash driver supports the following memory chips: E28F128J3, E28F640J3 and E28F320J3.
Selecting the type of memory chip used on your board is done in the `hw_config.c` file. You need to define the proper configuration option `CONFIG_IXP425_FLASH_<TYPE>` where `<TYPE>` is the memory chip type. For example, in the Cute board example, if the board uses the E28F128J3 chip, which is 16MB, define the following in the `hw_config.c` file:

```c
/* Flash chip */
token_set("CONFIG_IXP425_FLASH_E28F128J3", "m");
```

In case your board uses a different memory chip, you would have to modify the Flash memory driver to support this memory chip. The driver is provided in `vendor/intel/ixp425/modules/ixp425_flash.c`. The Flash layout defines the sections used to store these images and configuration files. Each section entry defines its start address, size and type. The definition of the Flash layout is in the `vendor/intel/ixp425/flash_layout.c` file, which should be modified to meet your needs. The distribution configuration option `CONFIG_RG_FLASH_LAYOUT_SIZE` should be set to 8, 16 or 32, in order to indicate the Flash size in MB. The Coyote board has a 16MB Flash, which contains the boot loader and two OpenRG images, along with factory setting and two OpenRG configuration sections. The detailed Flash layout for 16MB Flash is in the `vendor/intel/ixp425/flash_layout_16mb.c` file. Also available, are Flash layouts for 32MB in the `flash_layout_32mb.c` file, and for 8MB in the `flash_layout.c` file.

### 7.3.5 Low-level I/O

Low-level I/O configuration includes the GPIO lines and the PCI bus.

#### 7.3.5.1 GPIO Lines

GPIO lines are handled by a set of functions, which are part of the kernel platform specific code, defined in `arch/arm/mach-ixp425/gpio.c`. These functions are used to configure and access the various GPIO lines and are invoked upon a GPIO interrupt. No modifications are needed to these functions.

#### 7.3.5.2 PCI Bus

The PCI bus requires a set of functions that initialize the GPIO lines and IRQs of your board. The Coyote platform defines these functions in the `arch/arm/mach-ixp425/coyote-pci.c` file, as depicted here:

```c
#include <linux/pci.h>
#include <linux/init.h>
#include <linux/delay.h>
#include <asm/mach/pci.h>
#include <asm/arch/irqs.h>
#include <asm/arch/pci.h>
#include <asm/arch/gpio.h>
#include <asm/arch/ixp425-pci.h>

... void __init coyote_pci_init(void *sysdata) {
  ...
```
You could use the Coyote code as the base for the code you need to write for your board. In the Cute board example, you would need to create a `cute-pci.c` file, which will handle your board's hardware initialization and mapping of PCI slots to IRQS. In order to inform the kernel of the PCI handlers' location, you would need to modify the `arch/arm/kernel/bios32.c` file, which sets the PCI vector according to the machine type. For example, the Coyote board uses the following lines to set the PCI vector:

```c
... extern struct hw_pci coyote_pci; ...
void __init pcibios_init(void) {
  ...
  #ifdef CONFIG_ARCH_IXP425
    ...
      if (machine_is_coyote()) {
        hw = &coyote_pci;
        break;
      }
    ...
  #endif
  ...
}
```

In the Cute board example, assuming you have created the `cute-pci.c` file as defined above with a `cute_pci` vector, add the following lines immediately after the lines used by the Coyote:

```c
extern struct hw_pci cute_pci;
  ...
if (machine_is_cute()) {
  hw = &cute_pci;
  break;
}
```

The `machine_is_cute()` function is automatically generated by the definition of the new machine type. Make the kernel recognize the right PCI to use on the board by modifying the Makefile under the `arch/arm/mach-ixp425` directory. Again, you may find the Coyote line, similar to the following:

```makefile
obj-$(CONFIG_ARCH_IXP425_COYOTE) += coyote-pci.o
```

Then, add a line for the Cute board, which would be:

```makefile
obj-$(CONFIG_ARCH_IXP425_CUTE) += cute-pci.o
```
Preparing for Production

OpenRG supplies an effective tool for making the mass-production process faster and easier. The issue of mass-production has several major differences from the usual development cycle. This chapter describes the complete Flash image and the factory settings, both designed for the mass-production purpose. The Flash image (`flash.img`) and the factory settings (`rg_factory`) files offer a solution both for the common manufacturing of all boards, and for the board-specific information that is unique for each unit.

8.1 Building a Complete Flash Image

8.1.1 The flash.img File

The `flash.img` file is a large file, which holds all needed binaries and data that you should burn on the board. The file is constructed of the bootloader (for example, `rgloader.img`), the `rg_factory` and the `opengr.png` itself.

During the development cycle, the `rgloader.img` is usually burnt once on the board. The factory settings are then burnt for the first time. From this point on, the changes on the Flash are mainly done by loading new images, either by using the CLI 'load' command (refer to Chapter 27) or by Remote Update (refer to the 'Firmware Upgrade' section of the OpenRG User Manual). It is critical to shorten any procedure done in the mass production stage. For this purpose, you can use the `flash.img` file.

Thus, in one simple operation, the board is ready; by burning the `flash.img`, each of the three elements is placed in the correct section of the Flash. For information on the Flash layout, refer to Chapter 6.
8.1.2 The build_flash Utility

You should prepare the flash.img file after the bootloader, rg_factory and openrg.img are prepared. The Flash device on each board should hold different factory settings. You can achieve this by using one of three options:

• Burn the same flash.img on each board, and change the factory settings (on-board). This option is recommended, as you will perform the burning process only once.

• Burn the same flash.img on all boards. Prepare a unique factory settings file for each board. Burn each board's unique factory settings into its factory settings section.

• Prepare a different flash.img for each board, each with different factory settings, before burning each board.

Any one of the options is possible, and depends on your chosen way for changing the factory settings. These options are detailed in the next section.

In each OpenRG distribution, there is an application for constructing a flash.img. This application is called build_flash, and is located under cd_image/images or build/pkg/perm_storage. This application receives command-line arguments that inform it how to build the final Flash image file.

The possible arguments are:

• -b <boot loader file> – A full name (including the path, if needed) of a pre-compiled rg_loader file.

• -f <factory settings file> – A full name (including the path, if needed) of the rg_factory file to be used. This field is optional; if skipped, no factory settings are written.

• -i <image file> – A full name (including the path, if needed) of the OpenRG pre-compiled image (usually, openrg.img). In platforms that support more than one image, this option may appear as many as the supported number of images, with the same or different image files.

• -c <counter> – A specific counter to the relevant image section. The first is the counter for first image, the next is for the second image, if it exists, etc. In case this is omitted, the sections will be enumerated in a successive order, starting with '1'.

• -o <output name> – The name of the Flash image file that will be produced. The default, in case this option is omitted, is flash.img.

• -p <position> – A value that indicates the position in the Flash where this output file should be written. Usually, this is 0x0, which is the default in case this option is omitted.

• -v – Verbose output. This will show the structure of the output file.

For example, suppose you want to build a new Flash image that is constructed of the following:
Preparing for Production

- The original RG-Loader image that is supplied by Jungo (rgloader.img)
- A factory settings file that you have changed manually, and is called my_factory_settings
- An image that you have compiled after all needed modifications, and is called my_firmware.img

Assuming all three files are located in the same directory, together with build_flash, perform the following:

```
$ ./build_flash -b rgloader.img -f my_factory_settings -i my_firmware.img
```

This command will produce a file named 'flash.img'. When burning this file to the Flash, the outcome will be a board that contains OpenRG's original bootloader, together with the modified factory settings and your new image. For instructions on burning the Flash image on the board, please refer to the OpenRG Installation Guide.

8.2 The Factory Settings

During the production process, some data is saved on the boards, varying from unit to unit. For example, the network MAC addresses, the product serial number and more.

In OpenRG, these settings are called 'Factory Settings'. The factory settings is a configuration set stored in a special location on the board. This set holds the values for board-specific parameters, and is read during the first operation of the board or when restoring the board's defaults.

The following sections describe this mechanism and how to use it.

8.2.1 How OpenRG Uses Factory Settings

The factory settings are stored in their own section on the Flash, of type FACTORY (for information about the Flash layout, refer to Chapter 6). On boot, OpenRG checks for the presence of a valid rg_conf. If none is found, as in the first time OpenRG boots, it uses the default configuration file (rg_default), merges the factory settings into it, and saves this set back to the Flash as the new rg_conf. OpenRG then reads rg_conf, which contains the factory settings already, and operates accordingly. On subsequent boots, OpenRG reads rg_conf, which already contains the merged factory settings. By this merging, the board-specific values found in the factory settings are added to the product general values, found in rg_default.

```
rg_default:
{dev
  (ixpl
    {enabled(1))
      .
      .
    )
  }
}
```

```
rg_factory:
{dev
```
rg_conf:

```json
{dev
  {ixp1
    {enabled(1))
    {mac(00:11:22:33:44:55))
    .
    .
  }
}
```

If no factory settings are available, OpenRG will stop at the BootLoader stage, showing a relevant warning message. At this stage, it will use a predefined minimal `rg_conf`, which contains a LAN Ethernet device with a predefined IP address of 192.168.1.1. This allows the burning of the factory settings to the board. On the next boot, OpenRG will find these factory settings and continue with the merge process described above.

When restoring the default settings, OpenRG repeats the process described above, merging the factory settings into `rg_default` and saving the result to the new `rg_conf`.

The `rg_default` configuration set is created at the compilation time and is a part of the OpenRG binary image. It contains system defaults that remain constant across deployed gateway products, such as initial Firewall configuration, physical interfaces configuration, etc.

The configuration parameters in `rg_factory` are written in the same fashion as in `rg_conf`; any parameter that may be written to `rg_conf` may also be written to `rg_factory`. Refer to the Configuration Entries Guide for a complete list of available configuration entries.

### 8.2.2 Changing the Factory Settings

The factory settings set, `rg_factory`, is initially saved on the Flash (usually as a part of the `flash.img` burning). You can change its content during one of three different stages:

1. During the compilation stage; this is suitable for options relating to all boards.

2. After the compilation; in this stage, the `rg_factory` is a text file, which you may easily edit.

3. On the board; when the `rg_factory` is already burnt on the Flash, you may manipulate it using special APIs.

The third way is the most convenient for board-specific configuration, such as MAC addresses. Note that although the factory settings can be changed during the 'compilation stage', it is preferable to write values that are common to all boards and that can be changed by the user, into `rg_default`. `rg_factory` is written in a 'set' format. For an explanation of working with sets, refer to Section 18.1.
8.2.2.1 Changing at Compilation Stage

*rg__factory* is produced by the *gen__rg__def.c* application, found under *pkg/main/*. This application first defines all parameters needed for OpenRG. At the end of this process, the function *factory__settings__set()* is called. The purpose of this function is to allow you to enter your own parameters. The function is found under *pkg/vendor/user__mode/factory__settings.c*. In the distributions of OpenRG's software, this function is empty. You should add your firmware-specific parameters to this function's code. The function prototype is:

```c
void factory_settings_set(set_t **rg_set, set_t **factory_set);
```

The first argument, *rg_set*, is the default OpenRG settings, and is used as an input argument. The second argument, *factory_set*, is used as the output argument. Each parameter that is added to this set, will be saved in the factory settings. The set does not start as an empty set; several sample values are already found in it when it is passed to this function. You can use these values as an example. The values are found in the *gen__rg__def.c* file. Here are a few lines taken from there:

```c
1  set__set_path_str(&rg__factory, Smanufacturer / Shardware / Sversion, "111");
2  set__set_path_str(&rg__factory, Smanufacturer / Shardware / Sserial_num, "222");
3  set__set_path_str(&rg__factory, Smanufacturer / Svendor_name, "Jungo Ltd.");
```

This code fragment means:

**Line 1** Add a new entry named manufacturer/hardware/version to the *rg__factory*, and set the value "111" in this entry. The words that start with a capital S are strings constants, defined at the beginning of the file. Thus, this expression passed as the second argument is simply a string literals concatenation, which produces the mentioned final string. This is OpenRG's convention, which you may choose to adopt or not.

**Line 2** Similarly, add the entry manufacturer/hardware/serial_num, and place there the value "222".

**Line 3** In the same way, add the entry manufacturer/vendor_name, and define its value to be "Jungo Ltd.".

You should use a similar way to define your own parameters. Within *factory__settings__set()* you should use the argument *factory_set* instead of using &rg__factory. After compiling the image by running *make*, a new file called *rg__factory* is created and added to the *build/pkg/main/* directory. This file will include all the entries and values that were added in *factory__settings__set()*.

After the file is completed, you should burn it to the board. One way is to load it within a complete *flash__img* file, as described in the beginning of this chapter. Another way is to separately load the *rg__factory* file to its correct section in the Flash. Thus, all boards may be first burnt with the common *flash__img*, and then each of them can be burnt with its very specific *rg__factory* file.

You may prefer, however, to burn all boards with the same *flash__img* and then manipulate the board-specific parameters 'on-board'. This possibility is described in the next section.
8.2.3 Applying Board-specific Factory Settings

There are default settings, such as the base MAC address and serial number, that are individual for each board. These settings are provided by a board vendor. The serial number is used for hardware identification only. The base MAC address is used for generating the default MAC addresses of the gateway's network devices and the default key for a WPA-secured wireless network.

When assigning a base MAC address to a board, the vendor also reserves a specific range of MAC addresses following the base MAC. The MAC addresses within this range will not be assigned to any other board. Depending on the platform, the range may include either eight or sixteen MAC addresses. Note that the default MAC addresses of OpenRG's network devices will be created within this range.

The default MAC addresses are created upon the first boot, and then each time after restoring OpenRG to defaults. Each MAC address is calculated according to its pre-defined offset from the base MAC. The default MAC addresses are stored directly in `rg_conf`, as their values remain static each time they are generated from the base MAC.

Note: If a board includes a Ralink wireless network device, the value of the base MAC address must be such that after adding the device's offset, the resulting value will be divisible by four.

In contrast to default MAC addresses, the default WPA network key, which is generated using a CLI command described further on, is randomly calculated from the base MAC address. Therefore, the default WPA network key is stored in `rg_factory` to preserve its value against deletion when OpenRG is restored to defaults.

The decision on whether to store the board's base MAC address and serial number in `rg_factory` depends on how the vendor supplies this board-specific information.

Some board vendors store the base MAC address and serial number on the board's Flash (NVRAM). In this case, the base MAC and serial number will be retrieved from the Flash by vendor-specific callback functions, which should be implemented in the ported OpenRG distribution. As the board's base MAC and serial number are already stored on the Flash, you will only need to generate the default WPA network key (from the base MAC) and the gateway pin code, and store them in `rg_factory`.

Most board vendors, however, do not provide the base MAC and serial number on the Flash. If your board is supplied by such a vendor, you should store both the base MAC and serial number in `rg_factory`, along with the generated WPA network key.

In both of the described cases you must use OpenRG's hidden CLI command `factory generate` to store the required board-specific settings in `rg_factory`.

The following is the command's usage example depicting the case in which you should manually store the base MAC address and serial number on the board.

```
OpenRG> factory generate 14:59:85:5b:ba:d8 259372820662
Generating factory settings:
Base MAC address is 14:59:85:5b:ba:d8
```
Preparing for Production

--

Serial number is 259372820662
Device ra0, password '67201a17'
Access Point WPS password '69150563'
Factory settings generated and saved. Need to restore factory setting to apply changes.

OpenRG>

The base MAC will be stored in the system/base_mac entry; the serial number—in manufacturer/hardware/serial_num. The generated WPA network key will be stored in dev/<wireless device name>/wpa/preshared_key.

Note: If you do not specify the base MAC and serial number, their values will be generated randomly.

In case the board's vendor has stored the base MAC and serial number on the Flash, run the generate command without specifying any parameters, as shown in the following example.

OpenRG> factory generate
Generating factory settings:
Base MAC address is 14:59:85:5b:ba:d8
Serial number is 259372820662
Device ra0, password '67201a17'
Factory settings generated and saved. Need to restore factory setting to apply changes.

OpenRG>

Although the base MAC address and serial number appear in the command's output, only the default WPA network key is generated and stored in rg_factory.

8.2.4 Factory Settings Values

All OpenRG's default settings can be changed in the factory settings. The list below includes the settings to be changed for production. You may want to make additional changes to the default settings. You can refer to the Configuration Entries Guide for a detailed description of all the fields that can be used in the OpenRG configuration file.

Most factory settings are identical on all boards, therefore it is recommended to change the default values of these settings at the compilation stage. For instructions, refer to Section 8.2.2.1. Note that the first list below displays the factory settings that are unique and must be changed per board after the initial flashing. The second list displays the rest of the factory settings that are common for all boards.

Note: Setting default values different from the ones supplied by OpenRG may require changes in the final User Manual.

The following factory settings should be modified according to your device and vendor settings.

Unique settings:

- dev/<DEVICE_NAME>/mac The MAC address of the device. Each device, either WAN or LAN, has a unique name under the dev entry. Each device must have a registered MAC address. This setting is different for each board.

- manufacturer/hardware/serial_num The product serial number as it will be displayed in the WBM 'About' page. This setting is different for each board.
• **network/rg_mac** OpenRG's initial MAC address, used for bridge devices. *This setting is different for each board.*

• **system/base_mac** The first available MAC address from which a random MAC address will be generated for logic devices that need MAC. *This setting is different for each board.*

**Common settings:**

• **admin/tod/server/%/name** IP or hostname of Time Of Day (TOD) server. The default value is only for the first server. A new server you add will not have an IP or hostname of TOD server by default.

• **bluetooth/mac** Bluetooth device MAC address.

• **cert/%/cert** Certificate data.

• **cert/%/name** Certificate name. Exists for owner X509_OPENRG only. Used for OpenRG's certificate identification. It is the same as field 'CN=OR_<name>' in the certificate data if the field exists.

• **cert/%/owner** This entry indicates whether the certificate is for identifying yourself as OpenRG (X509_OPENRG) or for confirmation of identification (X509_CA). In most cases you will need to change your given certificate owner to X509_OPENRG.

• **cwmp/acs_url** Auto Configuration Server (ACS) (TR-069 server) URL.

• **cwmp/username** Username used to authenticate OpenRG when connecting to ACS.

• **cwmp/password** Password used to authenticate OpenRG when connecting to ACS.

• **cwmp/conn_req_username** Username used to authenticate the ACS making a Connection Request to OpenRG.

• **cwmp/conn_req_password** Password used to authenticate the ACS making a Connection Request to OpenRG.

• **cwmp/provisioning_code** String used in Inform message as DSLHome InternetGatewayDevice.DeviceInfo.ProvisioningCode.

• **dev/<name>/dhcps/vendor_specific** The value for vendor-specific information.

• **dev/<name>/dhcps/lease/%/hardware_mac** MAC address of a static/dynamic DHCP lease.

• **dev/<name>/dhcps/lease/%/vendor_id** The class ID of the lease vendor, written as a null-terminated string of hexadecimal characters.

• **dev/<dev_name>_atm/pvc_scan/%/vpi** Virtual Path Identifier (VPI) parameter for an Asynchronous Transfer Mode (ATM) connection.
**dev/**<dev_name>_atm/pvc_scan/%/vci** Virtual Circuit Identifier (VCI) parameter for an ATM connection.

Note: Values of these VPI/VCI entries are used when performing the Permanent Virtual Circuit (PVC) scan. The number of defined VPI/VCI pairs may vary across platforms.

- **fs/codepage** Client codepage. This entry specifies the character encoding table to be used when reading files using the file server.

- **jnet/domain/id** The identifier of the JRMS domain to which OpenRG belongs.

- **manufacturer/sys_object_id** The specific system object ID value to be returned when queried by SNMP for sysObjectID.

- **manufacturer/hardware/version** The hardware version of the system, as it will be displayed in the WBM 'About' page.

- **manufacturer/vendor_name** The vendor company name as it will be displayed in the WBM 'About' page.

- **manufacturer/description** The string used in DSLHome Inform message as InternetGatewayDevice.DeviceInfo.Description.

- **manufacturer/product_class** The string used in DSLHome Inform message as InternetGatewayDevice.DeviceInfo.ProductClass.

- **manufacturer/vendor_oui** The Vendor Organizational Unique Identifier (OUI), made up of six hexadecimal digits, using all uppercase letters and including any leading zeros. Used in DSLHome Inform message as InternetGatewayDevice.DeviceInfo.ManufacturerOUI.

- **manufacturer/boot_rom_version** Boot Read Only Memory (ROM) version.

- **manufacturer/model_number** The model number.

- **rmt_upd/url** The URL of the first image or redirection file. For instance: 'http://update.jungo.com/openrg.rmt'.

- **ssh/host_rsa_1/private** SSH private host key for protocol version 1.

- **ssh/host_rsa_1/public** SSH public host key for protocol version 1.

- **ssh/host_rsa_2/private** SSH private RSA host key for protocol version 2.

- **ssh/host_rsa_2/public** SSH public RSA host key for protocol version 2.

- **ssh/host_dsa_2/private** SSH private DSA host key for protocol version 2.
• **ssh/host_dsa_2/public** SSH public DSA host key for protocol version 2.

• **system/contact** The specific contact information to be returned when queried by SNMP for sysContact.

• **system/factory_version** The version of the factory settings.

• **system/location** The textual description of the specific location of the node to be returned when queried by SNMP for sysLocation.

• **system/mac_company_id** The first three octets of the MAC address, which are vendor specific

• **system/name** The specific administratively-assigned name for the product to be returned when queried by SNMP for sysName.

• **system/panic_timeout** (for Linux-based solutions) The time in seconds to wait before rebooting on a panic situation. Should be set to a value different from zero, otherwise the board will not reboot on panic.

• **wbm/session_lifetime** Idle time in milliseconds before releasing a session. Session lifetime should be ‘1’ and above. When a session is released, you will have to login again to start a new session.

### 8.2.5 Pre-configured Factory Settings

In some cases, you may wish to pre-configure part of the gateway's factory settings. For example, you can define the ISP's PPP credentials (username and password) in `rg_factory`, eliminating the user from having to enter them manually. In this case, the Installation Wizard should not request these credentials and skip the Internet connection step.

The Installation Wizard is comprised of the following major steps:

1. Login setup
2. Internet connection
3. Wireless setup
4. Jungo.net setup

In the next sections, each of the above is referred to as a "pre-configured item".

#### 8.2.5.1 Pre-configuring the Installation Wizard

The `rg_factory` contains a section (set) named `preconf`. This set may consist of entries for each pre-configured item, eliminating the need for an Installation Wizard step. The `preconf set` is divided into the above steps, excluding the first one:
In the initial boot sequence, after the `rg_factory` entries are merged into `rg_conf`, the preconf set will reside under `rg_conf`, as its son. At this stage, a new Main Task entity (mt_preconf) will detect that the preconf set exists in `rg_conf`, and will create the required configuration in `rg_conf`. `openrg_reconf()` is later called, so that pre-configuration will take effect.

### 8.2.5.1.1 Login Setup

The login setup can already be avoided in the Installation Wizard, therefore there is no need for a designated pre-configured item. If you wish to pre-configure the login details, perform the following:

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rg_factory/admin/user/0/username &lt;requested username&gt;</code></td>
</tr>
<tr>
<td><code>rg_factory/admin/user/0/password &lt;requested password (obscured)&gt;</code></td>
</tr>
<tr>
<td><code>rg_factory/wbm/is_login_setup_done 1</code></td>
</tr>
</tbody>
</table>

### 8.2.5.1.2 Internet Connection

An example:

```plaintext
(int_conn
  (type="pppoe")
  (username="dina")
  (password="*****") /*obscured*/
  (vpi=8)
  (vci=35)
)
```

In this example, mt_preconf creates a PPPoE device. In a multiple WAN gateway, you must also add a physical WAN device name (otherwise the first physical WAN device is taken automatically).

```plaintext
(int_conn
  (phy="dsl_atm0")
  (type="PPP")  
  .  .  
)
```

### 8.2.5.1.3 Wireless Setup

OpenRG supports more than one Access Point (AP). In most cases, one is "standard", and the other are "virtual". This enables two separate wireless networks—a primary, unsecured network, and a secondary, secured (encrypted) network. To match this configuration, the device name must be specified in the preconf set.

Since the virtual device name is determined at compilation time (for example, a Ralink wireless driver is named "ra8", and a Broadcom driver is named "wl0.1"), the device in the preconf set will be the exact one used by OpenRG. For example:

```plaintext
(wlan
  (dev
    (ra0
```
The "dev" level is not redundant in the above structure—it facilitates the merge into the "dev" sub-tree under `rg_conf`.

Unlike the Internet connection pre-configured item, the device structure under "ra0" will be the same as the one in a real `rg_conf`.

This means that you can copy a ready-made wireless device configuration from `rg_conf` and paste it into the preconf set. This allows maximum flexibility in the configuration. However, for common use cases, you can copy a ready-made template from the ones provided later in this chapter.

Since usually only a small portion of the fields are copied and pasted, `mt_preconf` merges the fields (using `set_merge()`) with the ones in `rg_conf`:

- The device's set in `rg_conf` will not be deleted. It will be a basis for the "patch".
- If a field exists in the preconf set, it will overwrite its parallel field in `rg_conf`.
- If a field does not exist in the preconf set, its parallel field in `rg_conf` will not be changed.

### 8.2.5.1.4 Jungo.net Setup

An example:

```
{jnet
  (enabled="1")
  (email="johns@Jungo.net")
  (password="*****") /*obscured*/
  (domain
    (id="Jungo.net")
  )
  (set_by_user=1)
}
```

- The structure of the Jungo.net preconf set is the same as the one in a real `rg_conf`.
- The Jungo.net preconf set is merged (using `set_merge()`) into `rg_conf` by `mt_preconf`.

### 8.2.5.2 The Compilation Flag

`mt_preconf` is compiled by default because the `CONFIG_RG_PRECONF` flag is set to "y" (`CONFIG_RG_PRECONF="y"`). However, you can compile an image without `mt_preconf` by adding `CONFIG_RG_PRECONF="n"` to the compilation command.

Additionally, the user can use the WBM to select whether or not to use the Installation Wizard's pre-configured values, by selecting the appropriate check box in the 'Installation Wizard' section of the 'Settings' menu item under the 'System' tab.
8.2.5.3 Pre-configured Set Templates

This section contains sets of pre-configured entries for the most common use cases. All roots are sons of the `rg_factory` set. If you wish to disable the Installation Wizard process, use:

```
wbm/is_installation_done 1
```

8.2.5.3.1 Internet Connection for a DSL Gateway

`<phy>` stands for the name of the physical WAN interface (for example, "dsl_atm0").

- **PPPoE**

  ```
  preconf/int_conn/phy <phy>
  preconf/int_conn/type pppoe
  preconf/int_conn/username <username>
  preconf/int_conn/password <obscured password>
  preconf/int_conn/vpi <vpi>
  preconf/int_conn/vci <vci>
  preconf/int_conn/encaps <encaps>
  ```

- **PPPoA**

  ```
  preconf/int_conn/phy <phy>
  preconf/int_conn/type pppoa
  preconf/int_conn/username <username>
  preconf/int_conn/password <obscured password>
  preconf/int_conn/vpi <vpi>
  preconf/int_conn/vci <vci>
  preconf/int_conn/encaps <encaps>
  ```

You can leave `<phy>` unconfigured if there is no multiple WAN. Leave `<vpi/vci/encaps>` unconfigured for an automatic PVC scan.

8.2.5.3.2 Internet Connection for an Ethernet Gateway

- **DHCP**

  ```
  preconf/int_conn/phy <phy>
  preconf/int_conn/type dhcp
  ```

- **PPPoE**

  ```
  preconf/int_conn/phy <phy>
  preconf/int_conn/type pppoe
  preconf/int_conn/username <username>
  preconf/int_conn/password <obscured password>
  ```

You can leave `<phy>` unconfigured if there is no multiple WAN.
8.2.5.3.3 Wireless

:name: stands for the name of the wireless interface (for example, "ra0").

A typical wireless configuration consists of the following steps:

1. Enable/disable wireless:

   preconf/wlan/dev/<name>/enabled <1|0>

2. Set SSID:

   preconf/wlan/dev/<name>/wl_ap/wl_ssid <ssid>

3. Set security type:

   • None:

     preconf/wlan/dev/<name>/web_auth 0
     preconf/wlan/dev/<name>/wpa/privacy_enabled 0
     preconf/wlan/dev/<name>/wpa/accept_wpa2_stas 0
     preconf/wlan/dev/<name>/wpa/accept_8021x_wep_stas 0
     preconf/wlan/dev/<name>/wpa/accept_non_8021x_wep_stas 0

   • Web authentication:

     preconf/wlan/dev/<name>/web_auth 1
     preconf/wlan/dev/<name>/wpa/privacy_enabled 1
     preconf/wlan/dev/<name>/wpa/accept_wpa2_stas 0
     preconf/wlan/dev/<name>/wpa/accept_8021x_wep_stas 0
     preconf/wlan/dev/<name>/wpa/accept_non_8021x_wep_stas 0

   • WPA:

     preconf/wlan/dev/<name>/web_auth 0
     preconf/wlan/dev/<name>/wpa/privacy_enabled 1
     preconf/wlan/dev/<name>/wpa/accept_wpa2_stas 1
     preconf/wlan/dev/<name>/wpa/accept_8021x_wep_stas 0
     preconf/wlan/dev/<name>/wpa/accept_non_8021x_wep_stas 0
     preconf/wlan/dev/<name>/wpa/preshared_key <pre-shared key>

If there are two SSID's, the primary wireless device should use the 'None' option, and the virtual (secured) wireless device should use the 'WPA' option, but with:

   preconf/wlan/dev/<name>/wpa/accept_wpa2_stas 1

8.2.5.3.4 Jungo.net

   preconf/jnet/enabled 1
   preconf/jnet/email <email>
   preconf/jnet/password <obscured password>
   preconf/jnet/domain/id <domain_id>
   preconf/jnet/set_by_user 1
9

Software Configuration

OpenRG's feature set is achieved by setting values to configuration flags. You can modify the feature set by adding (or removing) a flag, setting the flag's value, and reconfiguring the development tree (refer to Chapter 19). Configuration flags are set and modified in the dist_config.c and feature_config.c files, under the rg/pkg/build directory. The list of configuration flags, reflecting the software's feature set, is written into a file named rg_configure. This file is auto-generated into rg/build during the configuration process, and must not be edited.

9.1 Feature Flags

There are two kinds of feature flags:

- CONFIG_<FEATURE_NAME>
- cCONFIG_<FEATURE_NAME>

The first feature flag is defined by the user and is used in 'ifdef' C code preprocessor statements. The second flag is generated automatically and can be used as a run-time boolean variable in C.

9.2 Adding Features

Adding new features to OpenRG consists of adding a feature flag, setting the flag's value and reconfiguring the development tree. Add the config flag in the rg_configure file according to the file's conventions. An example: Add the following lines to the rg_configure file:

```
CONFIG<feature_name>=y
cCONFIG<feature_name>=1
```
9.3 Removing Features

OpenRG distributions are supplied as a superset of features, including all the possible features a specific board and an OpenRG version have to offer. After evaluating the full range of features OpenRG offers in Evaluation mode, you may purchase a commercial license. This license will include only the specific features you purchased. In order to utilize this license and compile a commercial image without the uptime limitation and evaluation banners, you will first have to remove all the features that your license does not cover. If you try to load an image with unlicensed features, OpenRG will halt on boot with a non-valid license announcement. For additional information, refer to Section 3.3.
10
Tailoring the Web-based Management

This chapter discusses some of the more complex, yet common aspects that a programmer might encounter when developing WBM code. It is advised that you first read the comprehensive example in Chapter 5 before proceeding to issues raised in this section.

10.1 Links

Suppose that you want the user to be redirected to a device properties page when he clicks on the device name link. You should use the \texttt{p\_link\_mimic\_button()} function:

\begin{verbatim}
void p_scr_conn_list(html_t **handle)
{
  ...
  link = p_link_mimic_button(cell, g_btn_edit, dev->name, NULL);
  p_text(link, "
  ...
}
\end{verbatim}

This function mimics a button. When the link is pressed, the page scan function is automatically called, as if a virtual button (\texttt{g\_btn\_edit} in this case) was pressed. Note that this function does not write the text that appears on the link - it can be thought of as just a placeholder for the text, that sets the properties of the link.

The next line (\texttt{p\_text()}) is the actual setting of the text. The scan function should intercept a click on the button using \texttt{button\_num()}:

\begin{verbatim}
void s_scr_conn_list(void)
{
  int btn = button_num();
  ...
  if (btn == g_btn_edit)
  {
    /* Redirect to device properties page */
    active_page_set(PAGE_CONN_BASIC);
  }
\end{verbatim}

10.2 On-change Events

Section 28.3 mentions the btn_onchange button. There are some objects in certain pages that changing their state causes re-submission of the form, due to an event-driven Javascript code invoked on the browser, and that such a re-submission event is equivalent to clicking btn_onchange.

Such events allow displaying more accurate information for the user. Suppose that there is a drop-down menu for manipulating Remote System Notify Level (syslog). If the selection is one of "Error", "Warning" or "Information", the user will need to input an IP address (Remote System Host IP).

![Figure 10.1 System Logging](image)

However, if the user selects "None" in remote syslog level field, the IP address field must disappear.

![Figure 10.2 System Logging](image)

If you want the appearance and disappearance of the IP address to be executed without the user having to click the "OK" button, but only changing the value in the drop-down menu, do the following:

```c
void p_remote_log_section(html_t **handle)
{
    ...
    p_combo_box_list(cell, combo_info_create(sym_notify_level, 1), set_get_path_int(tmp, Sseverity_threshold), notify_level_list);
    if (int_entry_get(sym_notify_level) != LLEVEL_MASK)
    {
        p_edit_ip_addr(cell, sym_ip, inet_addr(ip));
    }
    ...
}
```

**Line 5** Change the onchange_activated argument from 0 to 1 when creating the drop-down menu (combo_info_create() call) in the page print function.

**Line 8** In the page print function, put the IP address field under a condition (so it will be printed only if the remote syslog level is not "None").

**Line 10** Notify level is other than "None", so print the IP address field.
10.3 Passing Parameters Between Scan and Print Functions

Sometimes the WBM programmer needs to convey data from within a scan function to a print function or vice versa.

- **From scan to print** Use the `attrib_t *param` argument of the `p_page()`. This argument will reside in `g_request->param` for accessing it from the print function. The `attrib_t` data structure allows passing multiple attribute-value pairs. If the passed parameter is allocated dynamically in the scan function, the scan function is responsible for freeing its allocation. This can be done safely after the call to `p_page()`.

- **From print to scan** This direction is more complex. Use hidden parameters. A hidden parameter is a hidden HTML field that is sent to the browser. When the user submits the form on the browser, all the hidden parameters are automatically posted to OpenRG.

Suppose you are adding two pages. In the first page, the user should select a device from a drop-down menu of devices. In the next page the user should select a description for the selected device. After submitting the form with the new device description, the system is expected to store the device description in `rg_conf`. The scan function that must do this task should know the device to which the description must be written.
As you can see in Figure 10.3, you need to convey information from (2) to (4). This is done in two steps—first you need to convey information from (2) to (3), and then from (3) to (4).

The first step is easy—the scan function directly calls the print function (recall that every scan function ends with a call to `p_page()`). For the second step you need to hide the device identifier in the page that you send to the browser. When the user submits the device description, this piece of information is posted to OpenRG and scanned (4).

(1) ```c
void p_scr_device_dropdown(html_t **handle)
{
    ...
}
```  

(2) ```c
#define PARAM_DEV_NAME "param_dev_name"

void s_scr_device_dropdown(void)
{
    char *dev_name;
    attrib_t *a = NULL, **attrib = &a;
    extract_data_t extract[] = {
        {SYM_DEVICE_DROPDOWN, {str:dev_name}, TYPE_COMBO_BOX_STR, ""},
        {NULL}
    };
```
/* Extract the drop-down menu selection */
extract_data(extract);

/* Set the next page */
active_page_set(page_device_description);

/* Pass the device identifier (dev_name) as an argument to the next page */
attrib_set(attrib, PARAM_DEV_NAME, dev_name);
p_page(a);
attrib_free(&a);
}

(3)
void p_scr_device_description(html_t **handle)
{
    html_t **frame;
    char *dev_name;
    attrib_t **attrib = &g_request->param;

    frame = p_page_header(handle, ...);

    if (g_request->param)
        dev_name = attrib_get(attrib, PARAM_DEV_NAME);
    else
        dev_name = entry_get(SYM_HIDDEN_DEV_NAME);

    /* Save the device identifier (dev_name) in the page for two reasons:
    * - s_scr_device_description() will know what device to manipulate
    * - If p_scr_device_description() is called after an error was discovered
      in s_scr_device_description(), then 'param' is NULL,
      * so the hidden parameter is used as a backup
    */
    p_hidden_param(frame, SYM_HIDDEN_DEV_NAME, dev_name);
    ...
}

(4)
void s_scr_device_description(void)
{
    char *dev_name, *description;
    set_t *dev_set;
    extract_data_t extract[] = {
        {SYM_DESCRIPTION, {str:&description},
        TYPE_STR_WITH_SPACES|TYPE_OPTIONAL, Tname},
        {NULL}
    };

    /* Extract description */
    extract_data(extract);
    if (g_request->errors)
        goto Exit;

    /* Retrieve the device identifier (dev_name) from the posted data */
    dev_name = entry_get(SYM_HIDDEN_DEV_NAME);

    /* Get the device set_t ** struct given its name */
    dev_set = dev_if_set(dev_if_get(dev_name));

    /* Write device description to rg_conf */
    set_set_path_str(dev_set, Sdescription, description);
    openrg_reconf(FLASH_DELAYED_TOP_PRIORITY);

    /* Return to the "parent" page */
    active_page_set(navigator_pop());

    Exit:
    p_page(NULL);
}

A print function that receives a parameter should save its value as a hidden parameter, not only for the corresponding scan function usage, but also for its own usage, when 'param' is NULL. When could this happen? If the validity checks in s_scr_device_description()
fail, this causes an error page to be displayed. When the user returns from the error page to
p_scr_device_description(), the error page has no knowledge of the value that is
needed to be passed to page p_scr_device_description(), so NULL is passed.

Other Methods for Passing Parameters

An alternative method for passing parameters between pages is the session context. Unlike
hidden parameters, the session context does not need to be passed to the browser and back.
Let’s rewrite the previous example using the session context method:

```c
#define SESSION_DEV_NAME "session_dev_name"

void p_scr_device_dropdown(html_t **handle, void *param)
{
    ...
}

void s_scr_device_dropdown(void)
{
    char *dev_name;
    extract_data_t extract[] = {
        {SYMDEVICE_DROPDOWN, {str:&dev_name}, TYPER_COMBO_BOX_STR, ""},
        {NULL}
    };
    /* Extract the drop-down menu selection */
    extract_data(extract);
    /* Set the next page */
    active_page_set(page_device_description);
    /* Store the device identifier (dev_name) in the session */
    request_session_set(SESSION_DEV_NAME, dev_name);
    p_page(NULL);
}

void p_scr_device_description(html_t **handle, void *param)
{
    html_t **frame;
    char *dev_name;
    frame = p_page_header(handle, ...);
    ...
    /* No need to touch the session here */
}

void s_scr_device_description(void)
{
    char *dev_name, *description;
    dev_if_t *dev;
    extract_data_t extract[] = {
        {SYM_DESCRIPTION, {str:&description}, TYPESTR_WITH_SPACES|TYPEOPTIONAL, Tname},
        {NULL}
    };
    /* Extract description */
    extract_data(extract);
    if (g_request->errors)
        goto Exit;
    /* Retrieve the device identifier (dev_name) from the session */
    dev_name = request_session_get(SESSION_DEV_NAME);
    /* Get the device set_t ** struct given its name */
    dev = dev_if_set(dev_if_get(dev_name));
    /* Write device description to rg_conf */
    set_set_path_str(dev_set, Sdescription, description);
}```
10.4 Editable Tables

Editable table handling in the WBM is considered quite a complicated task for beginners, since it involves more than one page (assuming that editing/adding a single entry is done in a separate page). The requirements are:

1. When clicking the "New" button of an entry, the user should be redirected to a new entry editing page. After filling this page and submitting it, the new entry should appear in the table.

2. When clicking the "Edit" button of an entry, the user should be redirected to the entry's editing page. After updating the information and submitting it, the updated entry should appear in the table.

3. When clicking the "Remove" button of an entry, the page should refresh and the entry must disappear.

The requirements imply that:

1. When clicking the "New" button of an entry, the next page (the single entry edit page) must be signalled to create a new entry.

2. When clicking the "Edit" button of an entry, the entry's unique identifier must be passed to the next page (the single entry edit page).

3. When clicking the "Remove" button of an entry, the page's scan function must be informed of the entry's unique identifier, in order to remove it from the database.

The most complicated point here is to convey data from the first page (the table's page) to the second page (the single entry edit page). The situation can be even more complicated if you need to convey extra data except for the entry's unique identifier. The following example illustrates what happens when you also need to convey a device identifier. Suppose you want to implement a table of all PVC (VPI.VCI) pairs of a given ATM device. You will need to add two new pages:

1. One displaying the table entries.

<table>
<thead>
<tr>
<th>VPI.VCI</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.36</td>
<td></td>
</tr>
<tr>
<td>New VPI.VCI</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 10.4  VPI.VCI Pair Table**
2. The other for editing/adding a single entry.

![VPI.VCI Settings](image)

**Figure 10.5 Single Entry Settings**

Since the second page (the single entry edit page) must know which device to manipulate, you need a way to pass not only the entry's unique identifier, but also the device's identifier. To do so, overload the buffer `g_request->button_value` with these two pieces of information in the following format: "device name/entry number" (it is safe to use "/" as a delimiter between the two fields, because device name cannot contain "/"). The second page will parse `g_request->button_value` and retrieve the device and the relevant entry. The first page print and scan function will be as follows:

```c
void p_scr_pvc_table(html_t **handle)
{
    html_t **cells[2];
    set_t **pvc;
    ...
    /* Print table header. The table will have two columns: PVC and action */
    p_multi_cell_line(table, CELLS_HEADER, 2, cells, NULL);
    multi_cell_line_fill(cells, Tpvc, Taction, NULL);
    /* Traverse device's PVC pairs */
    for (pvc=NULL; (pvc=dev_if_pvc_get(dev, pvc, 0)); )
    {
        char val[MAX_VAR_NAME];
        /* Create a row with two columns */
        p_multi_cell_line(table, CELLS_LINE_DATA, 2, cells, NULL);
        /* Print the PVC pair into the first column (cells[0]) */
        p_pvc(cells[0], pvc);
        /* Print the edit/remove buttons into the second column (cells[1]).
         * Here is one of the key points: we associate the button value of
         * the edit/remove button with "device name/entry number". When the
         * edit/remove button will be pressed, the scan function will be
         * invoked and g_request->button value will be equal to
         * "device name/entry number"
         */
        sprintf(val, sizeof(val), "
        p_buttons(cells[1], val, g_btn_pvc_edit, g_btn_pvc_remove, 0);
    }
    /* Print the add button. When the add button will be pressed, the scan
     * function will be invoked and g_request->button value will be equal to
     * the device name
     */
    p_add_entry_row(table, 2, cells, NULL, g_btn_pvc_add, 1, dev->name);
}

void s_scr_pvc_table(void)
{
    char dev_name[MAX_VAR_NAME], *idx;
    dev_if_t *dev;
    int btn = button_num();
    ...
    if (btn == g_btn_pvc_add || btn == g_btn_pvc_edit)
    {
```
The second page print and scan functions will be as follows:

```c
void p_scr_pvc_entry(html_t **handle)
{
    char dev_name[MAX_VAR_NAME], *idx;
    int vpi = 0, vci = 0;
    set_t **set = NULL;
   ...
    dev_button_value_read(dev_name, &idx);
    if (idx)
    {
        /* idx is not NULL -> we are in edit mode. Retrieve the PVC entry to
         * manipulate from the given device. The VPI.VCI pair will be printed
         * according to this entry */
        set = set_get(dev_if_set(dev_if_get(dev_name)),
                      set_path_create(Satm, Spvc, idx, NULL));
        vpi = set_get_path_int(set, Svpi);
        vci = set_get_path_int(set, Svci);
    }
    /* Print the PVC pair (VPI.VCI). If we are adding a new entry, vpi and vci
     * will be set to 0 */
    p_atm_vpi_vci(table, vpi, vci, SYM_VPI, SYM_VCI, 0);
    ...
}
```

```c
void s_scr_pvc_entry(void)
{
    int vpi, vci;
    char dev_name[MAX_VAR_NAME], *idx;
    set_t **set = NULL;
    dev_if_t *dev;
    /* dev_button_value_read() parses 'g_request->button_value'. After calling
     * it, dev_name will contain the device identifier (dev_name) to manipulate,
     * and idx will contain the entry's identifier. If idx is NULL, it means
that it is a new entry (add mode) */
dev_button_value_read(dev_name, &idx);

/* Extract the VPI.VCI pair from the request */
s_atm_vpi_vci(&vpi, &vci, SYM_VPI, SYM_VCI);
if (g_request->errors)
goto Exit;

if (idx)
{
    /* idx is not NULL -> we are in edit mode. Retrieve the PVC entry to
    * manipulate from the given device. We will write VPI.VCI pair into
    * this entry */
    set = set_get(dev_if_set(dev), set_path_create(Satm, Spvc, idx, NULL));
}
else
{
    /* idx is NULL -> we are in add mode. Create a new PVC entry for the
    * given device. We will write VPI.VCI pair into this entry */
    set = set_set(dev_if_set(dev), Satm /*/ Spvc);
    set = set_index_add(set, SET_IDX_ADD_AUTO);
}
/* Write the VPI.VCI pair into the appropriate entry */
set_set_path_int(set, Svpi, vpi);
set_set_path_int(set, Svcv, vci);

/* Return to the "parent" page and reconf */
active_page_set(navigator_pop());
openrg_reconf(FLASH_DELAYED_TOP_PRIORITY);

Exit:
    p_page(NULL);


10.5 Look and Feel

The WBM's look and feel is determined by a large set of properties that create the "theme" or
"skin". This includes the page structure, sets of colors, product and vendor logos, buttons, link
style, and more. These properties are customizable—you can add your own theme that will
inherit some of OpenRG’s properties, but will differ in other.
OpenRG supports switching the active theme in run-time. The set of supported themes is determined in compilation time, using appropriate compilation flags. For example, if OpenRG supports theme A and theme B, you can change the active theme in the 'System Settings' page, or using the following CLI command:

```
OpenRG> conf set /wbm/theme <theme name (A or B)>
```

### 10.5.1 The Look and Feel Concept

The main concept behind the WBM's look and feel is the **theme**. A theme represents a certain look and feel. Each theme is implemented as a list of callbacks. Each callback is responsible for a specific look and feel element. Therefore, to change a specific look and feel element of a specific theme, the relevant callback must be written.

All themes are entries in the `themes_table[]` array, located in `rg/pkg/web_mng/lib/theme.c`. Each entry in the array is a structure containing all the look and feel callbacks of the specific theme.

When creating a new theme, it is easiest to copy all the callbacks from a similar theme and then start changing them according to the look and feel of the new theme.

### 10.5.2 Creating a New Theme

In order to create a new theme, perform the following:

1. Declare a new theme configuration flag in `rg/pkg/build/config_opt.c`.

   ```c
   { "CONFIG_GUI_MYTHEME", NULL, OPT_THEME | OPT_HC }
   ```
2. Add a configuration flag to the distribution in `rg/pkg/build/dist_config.c`.

```c
    token_set_y("CONFIG_GUI_MYTHEME");
```

3. Define a new theme enumeration and an `rg_conf` entry string in `rg/pkg/include/enums.h`.

```c
    ENUM_START(theme_t)
    ...
        ESTR(GUI_MYTHEME, 19, "mytheme")
    ENUM_END
```

4. Add the theme to the current image in `rg/pkg/mgt/lib/mgt_theme.c`.

```c
    static theme_t themes[] = {
        ...
        #ifdef CONFIG_GUI_MYTHEME
        GUI_MYTHEME,
        #endif
    };
```

5. Add the theme to the WBM settings in `rg/pkg/web_mng/lib/wbm_pages.c`.

```c
    static code2str_t themes[] = {
        ...
        {GUI_MYTHEME, lang:T_PTR(Tmytheme),
        !eCONFIG_GUI_MYTHEME},
        {-1, NULL}
    };
```

6. Create a new entry in the theme table in `rg/pkg/web_mng/lib/theme.c`.

```c
    static theme_cb_t themes_table[] = {
        ...
        #ifdef CONFIG_GUI_MYTHEME
        {
            id: GUI_MYTHEME,
            <List of callbacks>
        },
        #endif
        {
            GUI_TERM, /* Termination of themes */
        }
    };
```

   Copy the list of callbacks from a similar theme into this file.

7. Create a `mytheme.c` file under `rg/pkg/web_mng/cgi` to contain your specific callbacks and menus.

The next stage is to change the required theme callbacks to reflect the new look and feel.

### 10.5.3 Changing the Look and Feel of the New Theme

This section describes how to change various elements of the look and feel—the page structure, the main and secondary menus, and the styles of the page objects.

#### 10.5.3.1 Page Structure

The structure of a WBM page is implemented in the `struct_cb()` callback function. This function creates the main frame of the page. It returns placeholders (pointers) to other main
WBM elements. For example, the Main Menu is pointed to by `g_request->sidebar`, and a specific page by its returned value. Other placeholders can be added as well, such as a secondary menu, using `g_request->tabs_area`, and a page title, using `g_request->page_title_cell`.

The following is an example of a simple frame, as explained above.

![Diagram of Main Frame Structure]

**Figure 10.7 Main Frame Structure**

The above main frame structure is implemented by the following code.

```c
html_t **mytheme_struct_cb(html_t **handle)
{
    html_t **table, **row;
    table = p_table(handle, MAKE_PERCENT(100), TABLE_TRANSPARENT, 0);
    row = p_row(table);
    g_request->sidebar = p_cell_id(row, ID_STR_PAGE_TOPBAR, ALIGN_NONE, VALIGN_NONE, 0, 0, 0, 0, CLR_TRANSPARENT, NULL, NULL, 0, NULL);
    row = p_row(table);
    return p_cell_id(row, ID_STR_PAGE_BODY, ALIGN_NONE, VALIGN_BOTTOM, 0, 0, 0, 0, CLR_TRANSPARENT, NULL, NULL, 0, NULL);
}
```

**10.5.3.1.1 Exercise 1: Changing the Page Structure**

In this exercise you will customize the structure of a page by adding a footer that displays OpenRG's version number.

![Image of WBM Page with Footer]

**Figure 10.8 WBM Page with Footer**

To achieve this, create a new `struct_cb()` with a footer.

```c
1    html_t **mytheme_struct_cb(html_t **handle)
2    {
3        html_t **table, **row, **cell, **ret;
```
What does this code do?

**Line 17** Save the pointer to the specific page.

**Line 20** Add new row for footer.

**Lines 21-22** Control spacing.

**Lines 23-24** Add a cell to contain the version string.

**Lines 25-27** Add the version string.

**Lines 28-29** Control spacing.

**Line 31** Return a pointer for the specific page.

### 10.5.3.2 Main Menu

The Main Menu (referred to as "sidebar" or "topbar"), is a part of the page structure in most WBM pages (excluding the Login page, etc.). It is identical in all pages, and hierarchical (of a tree form). Each hierarchy level can be customized.

The main menu is comprised of two parts:

- **Contents** – Defines the contents tree of the menu
- **Engine** – Defines the layout and structure of the menu

The contents part is implemented as a tree, built on `page_redirection_info_t`. The menu tree should be built in the function `mytheme_main_menu_register()` in `pkg/web_mng/cgi/mytheme.c`. This function should be run when opening the WBM task. It is called from `wbm_register_other()` in `pkg/web_mng/cgi/wbmt.c`.

The following figure depicts a simplified example of the main menu's structure.
Each entry in the tree has a unique identifier, defined in the following enumeration. Its purpose is to provide a specific ID for each menu item, as well as define the order in which the menu items are displayed.

```c
typedef enum {
    MAIN_MENU_SECTION_ID_HOME = 0,
    MAIN_MENU_SECTION_ID_SERVICES = 100,
    MAIN_MENU_SECTION_ID_SVC_FIREWALL = 110,
    MAIN_MENU_SECTION_ID_SVC_FW_OVERVIEW = 111,
    MAIN_MENU_SECTION_ID_SVC_FW_ACCESS_CTRL = 112,
    MAIN_MENU_SECTION_ID_SVC_FW_LOCAL_SRV = 113,
    MAIN_MENU_SECTION_ID_SVC_FW_DMZ_HOST = 114,
    MAIN_MENU_SECTION_ID_SVC_DDNS = 120,
    MAIN_MENU_SECTION_ID_SYSTEM = 200,
    MAIN_MENU_SECTION_ID_SYS_OVERVIEW = 210,
    MAIN_MENU_SECTION_ID_SYS_SETTINGS = 220,
    MAIN_MENU_SECTION_ID_SYS_SETTINGS_OVERVIEW = 221,
    MAIN_MENU_SECTION_ID_SYS_SETTINGS_TIME_SETTINGS = 222,
    MAIN_MENU_SECTION_ID_ADVANCED = 300,
} mytheme_main_menu_section_id_t;
```

The following code is a partial example of how the main menu tree is built during its register function.

```c
#define MYTHEME_MAIN_MENU_PREFIX  "mytheme_main_menu_"

static page_redirection_info_t *mytheme_main_menu_sections;

void mytheme_main_menu_register(void)
{
    page_redirection_info_t *level1, *level2;

    /* Home */
    level1 = page_redirection_info_register(&mytheme_main_menu_sections,
                                            MAIN_MENU_SECTION_ID_HOME, T_PTR(Thome),
                                            page_id_get_by_str(PAGE_HOME_STR),
                                            MYTHEME_MAIN_MENU_PREFIX "home",
                                            T_PTR(T_NULL), 0, NULL, 0, (char *[\]){Smgt, NULL});

    /* Services */
    level1 = page_redirection_info_register(&mytheme_main_menu_sections,
                                            MAIN_MENU_SECTION_ID_SERVICES, T_PTR(Tservices),
                                            NULL, MYTHEME_MAIN_MENU_PREFIX "services", T_PTR(T_NULL),
                                            0, NULL, 0, (char *[\]){Smgt, NULL});
    page_redirection_info_register(&level1->submenu,
                                MAIN_MENU_SECTION_ID_SVC_OVERVIEW, T_PTR(Toverview),
                                page_id_get_by_str(PAGE_SVC_OVERVIEW_STR), MYTHEME_MAIN_MENU_PREFIX
                                "svc_overview", T_PTR(T_NULL), 0, NULL, 0, (char *[\]){Smgt, NULL});
```
The engine part of the main menu is implemented by the `sidebar_cb()` callback function, which builds the main menu based on its tree (font, positioning, etc.). The engine displays relevant menu items of the current page, and can also add menu items that are not currently in the tree (e.g. language, user, logout, help, sitemap, etc.).

Each page displays a sub-tree of the menu tree, depending on the current page:

![The Main Menu Sub-tree](image)

**Figure 10.10** The Main Menu Sub-tree

Following is an example of a main menu engine displaying the main menu tree.

```c
void mytheme_main_menu_cb(html_t **handle)
{
    html_t **table, **row, **cell;
    page_redirection_info_t *main_menu = mytheme_main_menu_sections;
    int level = 1;

    table = p_table(handle, MAKE_PERCENT(100), TABLE_TRANSPARENT, 0);
    while (main_menu)
    {
        page_redirection_info_t *selected_item;
        row = p_row(table);
        cell = p_cell(row, ALIGN_CENTER, VALIGN_CENTER, 0, 0, 0, CLR_TRANSPARENT, NULL, NULL, 0);
        selected_item = p_main_menu_row(cell, main_menu, level);
        level++;
        main_menu = selected_item ? selected_item->submenu : NULL;
    }
}
```

### 10.5.3.2.1 Exercise 2: Customizing the Main Menu

In this exercise you will customize the main menu according to the main menu tree (see **Figure 10.10**.)
10.5.3.3 Secondary Menu

The secondary menu (referred to as the "tabs") is similar to the main menu, as it is also comprised of an engine and contents. The contents are specific per page, but are not required for every page. The secondary menu has a single level only, and is only displayed when contents exist for a specific page.

The secondary menu registration is similar to the main menu. Each entry has a unique identifier that defines the order of the menu items:

```c
typedef enum {
    TAB_PHONE_SET = 0,
    TAB_PHONEBOOK = 1,
    TAB_MONITORING = 2,
    TAB_GENERAL = 3,
} ata_tabs_order_t;
```

The registration of the secondary menu items to the required pages is implemented by the following code.

```c
tab_info_t *voip_tabs;

    tab_list_register(&voip_tabs, T_PTR(Tvoip));
    tab_register(&voip_tabs, TAB_PHONE_SET, &page_voip_phone_set, NULL);
    tab_register(&voip_tabs, TAB_PHONEBOOK, &page_voip_phonebook, hide_phonebook_tab);
    tab_register(&voip_tabs, TAB_MONITORING, &page_voip_monitoring, hide_monitoring_tab);
    tab_register(&voip_tabs, TAB_GENERAL, &page_voip, NULL);
```

10.5.3.3.1 Exercise 3: Adding a Secondary Menu to the Firewall

Move the firewall's third level menu items to the secondary menu.
Figure 10.12 Customizing the Menu Levels

The firewall’s secondary menu is implemented by the following code.

```c
typedef enum {
    MYTHEME_SEC_MENU_FW_OVERVIEW = 0,
    MYTHEME_SEC_MENU_FW_ACCESS_CTRL = 1,
    MYTHEME_SEC_MENU_FW_LOCAL_SRV = 2,
    MYTHEME_SEC_MENU_FW_DMZ_HOST = 3,
} mytheme_secondary_menu_order_t;

static tab_info_t *fw_tabs;
void mytheme_secondary_menu_register(void)
{
    tab_list_t *tabs;

    tabs = tab_list_register(&fw_tabs, T_PTR(Tfirewall));
    tab_register(&fw_tabs, MYTHEME_SEC_MENU_FW_OVERVIEW,
                 page_id_get_by_str(PAGE_FW_GENERAL_STR), NULL);
    tab_register(&fw_tabs, MYTHEME_SEC_MENU_FW_ACCESS_CTRL,
                 page_id_get_by_str(PAGE_FW_ACCESS_CONTROL_STR), NULL);
    tab_register(&fw_tabs, MYTHEME_SEC_MENU_FW_LOCAL_SRV,
                 page_id_get_by_str(PAGE_FW_PORT_FORWARDING_STR), NULL);
    tab_register(&fw_tabs, MYTHEME_SEC_MENU_FW_DMZ_HOST,
                 page_id_get_by_str(PAGE_FW_DMZ_HOST_STR), NULL);
}
```

Finally, call the firewall’s secondary menu registration function from its WBM initialization function. `rg/pkg/firewall/wbm/firewall_wbm_init.c`.

```c
void firewall_wbm_init(void)
{
    firewall_pages_init();
    if (vcCONFIG_RG_FIREWALL)
    {
        ...
        mytheme_secondary_menu_register();
        ...
    }
}
```

10.5.3.4 Styles

The styles of specific objects can be changed using the `style_cb()` callback function. To change the styles of HTML objects in the new theme, you must create a new `style_cb()` function. To use an alternative existing style, simply change to the required style, but to use a new style you must first add the new style to the `style_cb()` function and then add the style identifier to the required HTML object.

The styles are implemented by the following code. This is the code in `style_cb()`:
To add the style to a specific HTML object:

```
cell = p_cell(row, ALIGN_NONE, VALIGN_NONE, 0, 0, 0, 0,
CLR_TRANSPARENT, NULL, CLASS_ADMIN_USER, 1);
```

### 10.5.3.4.1 Exercise 4: Changing the Style of the Constant Links Bar

Change the size, color and fonts of the constant links bar, referred to as the "admin row", according to the following screen.

![Customizing Styles](image.png)

**Figure 10.13 Customizing Styles**

The style customization is implemented by the following code.

```
#define MYTHEME_FONTS "sans-serif"
void mytheme_style_cb(html_t **style)
{
    p_text(style, ".\ CLASS_ADMIN
    "[FONT-FAMILY: %s;"
    "Font-Size:14px;"
    "color: red;}], MYTHEME_FONTS);
    p_text(style, "TD.\ CLASS_ADMIN
    "A {FONT-FAMILY: %s;"
    "Font-Size:14px;"
    "color: red;}], MYTHEME_FONTS);
}
```

Finally, add the following code to `rg/pkg/web_mng/lib/rg_theme.c`.

```
void smb_style_cb(html_t **style)
{
    ...
    mytheme_style_cb(style);
}
```
10.6 Adding Static HTML Pages to the WBM

When developing OpenRG, you may wish to add static HTML pages to the WBM, and to create a corresponding HTTP shortcut for each of them. To achieve this, perform the following:

1. Create a new directory (for example, `http_static_pages`) under `pkg/web_mng`.

2. Save the HTML source of every relevant WBM page in the `http_static_pages` directory.

3. In the Makefile located under `pkg/web_mng`, add `http_static_pages` to the JMK_SUBDIRS variable.

4. In the `http_static_pages` directory, create a new Makefile with a variable called JMK_RAMDISK_HTTP_FILES, and add to it all the HTML pages you have saved in this directory. For example:
   ```
   JMK_RAMDISK_HTTP_FILES+=page1.html, page2.html
   ```

5. Add the following line to the variable 'RAMDISK_FILES_' located in `jmk/ramdisk_root_end.mak`:
   ```
   $(call RD_Export_Fmt, $(sav_JMK_RAMDISK_HTTP_FILES), /home/httpd/html)
   ```
   If the line is added as the last line, the previous line should end with a backslash.

6. Add the string 'JMK_RAMDISK_HTTP_FILES' to the variable 'global_ramdisk_objs' located in `jmk/subdirs_root.mak`.

7. Compile an OpenRG image and load it to the board. You should now be able to browse to the static WBM pages via their permanent URLs.
11

Tailoring the Role-based Permissions Engine

11.1 Overview

OpenRG’s user account management mechanism has been enhanced with a role-based permissions engine. This enhancement expands the range of user profile types (administrative and restricted) that can be defined in the system, and enables you to easily customize and fine-tune the set of permissions, which will be granted to each user profile. The term used in OpenRG for user profiles is roles.

The roles, feature access permissions, as well as the mapping between them, are defined prior to image compilation. The default OpenRG user accounts are also mapped to their corresponding roles prior to image compilation. Obviously, the default home user account is assigned the role HOME, and the administrator account is assigned the role ADMIN. If required, OpenRG can also be configured to allow changing a user’s default role to another one at runtime.

Each new user account registered in OpenRG must be assigned a specific role, thus inheriting its set of permissions. By using OpenRG’s CLI (either locally—via the serial interface, or remotely—via JRMS) you will be able to view permissions granted to users by their roles, as well as grant additional permissions (or deny them), as described further in this document.

The following figure outlines the structure and operation of the role-based permissions engine.
11.2 Defining Roles and Permission Sets in OpenRG's Code

The default list of user roles implemented in OpenRG is defined in `rg/pkg/main/gen_rg_def.c`, using the `set_user_permissions()` function.

OpenRG's feature access permissions are defined in `rg/pkg/include/enums.h` (see the `sec_perm_type_t` enum). This enum contains two sets of permissions: an Access set and a Feature set (see Figure 11.1). The permissions in the first set provide access to capabilities that their respective features offer, but not to their settings. For example, `SEC_PERM_ACCESS_FS` provides access to shared directories of OpenRG's file server, but it does not provide access to the file server's settings. On the other hand, its counterpart permission from the second set, `SEC_PERM_FILE_SERVER`, is responsible for providing access to the file server's settings via the WBM.

The actual mapping between roles and permissions is defined in `rg/pkg/main/def_permissions.c` (see the `sec_perm_t permissions[]` array).

Note: Permissions that are registered in the `sec_perm_t` enum, but are not defined in the `sec_perm_t permissions[]` array, will automatically be mapped to both the HOME and ADMIN roles.
11.3 Managing a Role's or a User's Set of Permissions at Runtime

When OpenRG is up and running, it is possible to view the set of permissions associated with a role, by issuing the following command from OpenRG's CLI:

```
OpenRG> conf print_permission admin/role/<index>/permission
```

where `<index>` is the role's index value (e.g., 3 – for the role ADMIN and 2 – for the role HOME). By default, the roles are indexed according to the order of their appearance in the `set_user_permissions()` function defined in `rg/pkg/main/gen_rg_def.c`.

After issuing the command shown above, the list of role permissions is printed, as in the following example:

```
{admin/role/3/permission/
  {access_wbm(1)}
  {access_serial_cli(1)}
  {access_telneta(1)}
  ...
  {reboot(1)}
  {restore_factory(1)}
  {firmware_upgrade(1)}
  ...
}
```

In addition, you can modify a role's set of permissions via the CLI, by running the following commands:

- To grant a new permission to a role, run:

  
  ```
  OpenRG> conf set_permission admin/role/<index>/permission <permission> 1
  OpenRG> conf reconf 1
  ```

  For example:

  ```
  OpenRG> conf set_permission admin/role/3/permission access_ssh 1
  OpenRG> conf reconf 1
  ```

  Note that this will grant the new permission to all users associated with this role.

- To disable a permission that was granted to a role, run:

  ```
  OpenRG> conf set_permission admin/role/<index>/permission <permission> 0
  OpenRG> conf reconf 1
  ```

  For example:

  ```
  OpenRG> conf set_permission admin/role/2/permission page_about 0
  OpenRG> conf reconf 1
  ```

  Again, all user accounts that are associated with this role will be affected accordingly.

As stated earlier, you can also grant a user additional permissions, which do not appear in the list of permissions assigned to the user's role. In addition, you can deny these permissions later on. Note, however, that you cannot deny permissions that have been inherited by a user from the associated role, without first disabling this permission in the role itself.

- To grant an additional permission to a user, run:

  ```
  OpenRG> conf set_permission admin/user/<index>/permission <permission> 1
  ```
Tailoring the Role-based Permissions Engine

OpenRG> conf reconf 1

For example:

OpenRG> conf set_permission admin/user/0/permission access_ssh 1
OpenRG> conf reconf 1

• To deny a permission granted to a user, but not inherited from the role, run:

OpenRG> conf set_permission admin/user/<index>/permission <permission> 0
OpenRG> conf reconf 1

For example:

OpenRG> conf set_permission admin/user/0/permission access_ssh 0
OpenRG> conf reconf 1

After making the necessary changes, you can check the user's permission settings by issuing the following command:

OpenRG> conf print_permission admin/user/0/permission

11.4 OpenRG's Permission Validation Mechanism

When a user logs into the WBM or CLI, OpenRG checks permissions of the user's account and of its associated role. To allow access to a certain feature, OpenRG must detect the corresponding permission (being enabled) at least in one of them.

The user's permissions are checked using a sophisticated validation mechanism, which comprises the following layers:

1. The configuration library (pkg/mgt/lib) – A layer at which a user's set of permissions is defined and read from. The following functions are the core internals of this layer:

   • set_set_path_permission – Sets a specific permission in the relevant rg_conf path.

   • set_get_path_permission – Gets the boolean value of a specific permission from an rg_conf path.

   

   Note: Since a user account inherits permissions from its associated role and it can also be assigned additional permissions (as described earlier), the set values of the user's permissions should not be calculated by directly calling the set_get_path_permission function. The function is_user_permitted should be used instead.

   • is_user_permitted (pkg/mgt/lib) – Returns 1 if the user has the specified permission set in rg_conf. Otherwise, it returns 0. For example:

```c
if (!is_user_permitted("admin", SEC_PERM_ACCESS_PRINTER))
{
    ...
}
```
2. **The Web-based management** – A layer that includes the following permission validation mechanisms:

- Validating the logged-in user’s permission to view/access a specific feature. The validation is performed using the `is_request_user_permitted()` function. If the logged-in user has the specified permission, the function returns 1 (otherwise, 0). For example:

  ```
  if (is_request_user_permitted(SEC_PERM_SYSTEM_SETTINGS))
  {
    ...
  }
  ```

- Validating the logged-in user’s permission to access a specific WBM page. The validation is performed using the `is_page_permitted()` function. If the user is permitted to view and submit the specified page, the function returns 1. For example:

  ```
  if (is_page_permitted(page_about))
  {
    ...
  }
  ```

11.5 **Associating Permissions with the GUI Elements**

Associating a permission with a GUI element (such as a page, service, button, or a top-bar node) can be done using one of the following methods:

- **Using a validation callback function with a context** – This validation function, attached to an element, receives a single void pointer (`void *`) argument, which functions as its context. This argument is used by the permissions engine when calling the callback function. The type definition of such a callback function is: `int (*cb)(void *ctx)`. The function returns 1 in case of successful validation.

Examples of using a callback function with a void pointer argument:

- For a page:

  ```
  void page_register_permission(char *page_id, ..., int (*permission_cb)(void *ctx), void *permission_ctx)
  ```

- For a service:

  ```
  void service_status_register_permission(service_id_t id, ..., int (*permission_cb)(void *), void *permission_ctx)
  ```

- For a button:

  ```
  int button_register_permission(char **caption, ..., int (*permission_cb)(void *), void *permission_ctx)
  ```

- For a top-bar node:

  ```
  page_redirection_info_t *topbar_register_permission(int id, ..., int (*permission_cb)(void *ctx), void *ctx)
  ```
• Assigning one specific permission to an element – The specified permission (treated as a context) is checked using the default callback function is_request_user_permitted_cb(). The function returns 1 in case of successful validation.

Examples of assigning a permission to an element using a specific context:

- For a page:
  ```
  page_register_perm(page_id, ..., SEC_PERM_<FEATURE_CODE>)
  ```

- For a service:
  ```
  service_status_register_perm(id, ..., SEC_PERM_<FEATURE_CODE>)
  ```

- For a button:
  ```
  button_register_perm(caption, ..., SEC_PERM_<FEATURE_CODE>)
  ```

- For a top-bar node: not used at this stage, but can be added upon a customer's request.

• Defining and using a default context – The default context for an element is checked using the callback function is_request_user_permitted_cb().

The default context can be set using one of the following methods:

- Compilation flags – Use JMK_SEC_PERM in the `envir.subdirs.mak` file to set a default permission for a whole sub-directory at compilation time. For example, in `pkg/voip/envir.subdirs.mak`:
  ```
  JMK_SEC_PERM=SEC_PERM_VOICE_ADVANCED
  ```

In case you wish to set a default permission for a specific file, use the JMK_SEC_PERM_<object-name> flag with a relevant permission as its value. For example, to provide the home user with access to the IP-PBX extension pages, add the following line in `pkg/voip/wbm/Makefile`:
```
JMK_SEC_PERM_pbx_extensions.o=SEC_PERM_VOICE
```

- Default permission values – If no compilation default is set for an element, the SEC_PERM_DEFAULT permission is used. In this case, access to the target element is allowed to both the administrator and home user. This default permission can be used with all of the mentioned elements, except for buttons.

If no compilation default is set for buttons, the SEC_PERM_ALL permission will be used. Note that in this case, the buttons will be available to all users.

Examples of assigning a default permission to an element:

- For a page:
  ```
  page_register(page_id, ...)
  ```

- For a service:
  ```
  service_status_register(ID, ...)
  ```
For a button:

button_register(caption, ...)

For a top-bar node:

page_redirection_info_t *topbar_register(int id, ...)
Integrating Applications

Adding applications to OpenRG is easy and straightforward. Your applications can exist within Main Task, or as stand-alone applications that interact with Main Task using Inter Process Communication (IPC).

12.1 Adding an Application to the Development Tree

1. Create a new directory as follows: # mkdir <DEST>/rg/pkg/<NEW_DIR>

2. Copy the user-mode application source to the new directory.

3. Create a Makefile for the user-mode application. Refer to this file as an example: <DEST>/rg/pkg/samples/process/Makefile

   Note: Additional information about working with makefiles can be found in Chapter 19.

4. In the user-mode application Makefile, openrg/pkg/<NEW_DIR>/Makefile, define the JMK_ROOT parameter to the relational location of the top directory (typically ../../ for all modules located in openrg/pkg/). Also, include the following environment definitions in the Makefile:

   ```
   ifndef JMKE_ENV_INCLUDED
   JMK_ROOT=../../
   include $(JMK_ROOT)/jmk/env_root.mak
   endif

   JMK_TARGET=<YOUR_TARGET>
   JMK_O_OBJS=<YOUR_OBJECTS>
   $(call JMKE_INCLUDE_RULES)
   ```
5. Add the user-mode application directory name to the JMK_SUBDIRS section of the openrg/pkg/Makefile as follows: JMK_SUBDIRS += <NEW_DIR>

6. Compile the User-mode application using the following command from the <DEST>/rg path: # make (this compiles the entire tree) Or Run # make from openrg/pkg/<NEW_DIR> to test the compilation (this compiles only your application). Note that when you compile only the application, you need to already have a configured and compiled tree. You must compile the tree again in order to add the application to the image file.

12.2 Adding an Application to the Image

The $(JMK_TARGET) variable is automatically added to the image.

12.3 Testing Your Application

You can test your application in two ways:

1. Compile an image with the application, install the image on the board. Verify that the application resides in the appropriate directory (/lib/modules and /bin, respectively) and then run the application.

2. Compile the application, transfer it to the board using TFTP and insmod/ run it. The received file should be chmod +x in order to run it.

   Note that a file transferred using this method will not be saved to Flash, and therefore will not appear after the next system boot.

12.4 Inter Process Communication

Existing applications can be used to customize and extend OpenRG's functionality. For example, an existing management interface or network media application can be easily incorporated within OpenRG. This section will explore the Inter Process Communication (IPC) method of integrating applications into OpenRG. A high-level design of the IPC is described, followed a detailed API reference. To sum up, a step-by-step tutorial is presented illustrating how to add a simple application. It is assumed that the reader is familiar with the OpenRG system architecture.
12.4.1 Why IPC?

Porting applications to the OpenRG environment renders them as separate processes running outside Main Task. This leaves OpenRG’s internal configuration file and services inaccessible to external applications. Clearly, a method must be constructed to allow processes running along-side Main Task to be able to perform operations such as:

- Read/write to the permanent storage configuration file
- Perform a Firmware update
- Perform reboot
- Gain access to another Main Task functionality

IPC is an inter process service (API) that grants remote processes read/write privileges to the OpenRG configuration file, and enables access to inherent gateway functionality.

12.4.2 High-level Design

The IPC service is implemented using TCP/IP sockets opened to the localhost server using a loopback network interface. This method has two advantages. First, it is fast. Using shared memory in Linux results in high-speed data transfer rates through the loopback interface. Second, the method is a multi-platform solution since TCP sockets are supported in the same way by all operating systems (this relates to APIs and not to the internal mechanisms). The IPC method can be viewed as a client/server system. The loopback network interface acts as a server for external client applications wishing to open IPC sessions. The server part of the API is implemented in Main Task. It accepts incoming TCP connections using the event_loop, and subsequently processes IPC service requests.

The IPC session is a blocking session. When it is running, OpenRG’s normal work-flow is halted. After a request name is read from the IPC socket, the request is dispatched to the request function (for example: mt_ipc_rg_conf_get or mt_ipc_reboot). The request function reads the remaining IPC request parameters and performs the real action (for example: mt_ipc_reconf reads the reconf_time parameter and calls openrg_reconf(reconf_time)). The external application is responsible for closing the blocking IPC client socket. During a single IPC session the external client can request multiple actions to be performed and be sure that no Main Task activity will take place in the interim. To use the IPC method construct your code according to the following framework. The operations inside the IPC session may vary.

1. Open an IPC session to Main Task. This will block Main Task from performing other tasks.
2. Read data from rg_conf using ipc_rg_conf_get.
3. Change received data according to your needs.
4. Write data back to rg_conf using `ipc_rg_conf_set`.

5. Commit changes to the permanent storage and reconfigure OpenRG for the changes to take effect using `ipc_reconf`.

6. Close IPC session. This will release Main Task to continue normal tasks.

![Figure 12.1  IPC Module Integration](image)

### 12.4.3 IPC APIs

#### 12.4.3.1 High-level Sample IPC APIs

The source code for the following APIs can be found in: `pkg/samples/rg_conf_ipc/ipc_client.ch`. The library is provided in source code as a sample of proper IPC usage.
- int ipc_open(void) Connects to the Main Task IPC service and opens an IPC session. Note that this function will block the normal Main Task work-flow, until the session is closed using ipc_close.

- void ipc_close(int fd) Closes an IPC session and disconnects from Main Task.

- set_t *ipc_rg_conf_get(int fd, char *path) Bring part of rg_conf to the external application.

- void ipc_rg_conf_set(int fd, char *path, set_t **set) Update part of rg_conf with data stored in the set.

- void ipc_reconf(int fd, flash_delayed_t time) Perform the reconfiguration routine and commits changes to permanent storage.

- void ipc_reboot(int fd) Reboot OpenRG.

- void ipc_remote_update(int fd, char *url) Initiate a remote upgrade session from a specified URL.

### 12.4.3.2 Generic OpenRG API Library

The following APIs are invocable from the libopenrg `<pkg/include/ipc.h>`. Note that this is an internal OpenRG Main Task API, and should not be normally used by an external application. External applications should use the high-level APIs described in the previous section.

- int ipc_connect(u16 port) Connect to Main Task and initialize an IPC session. For regular IPC sessions the port argument value should be RG_IPC_PORT_GENERIC.

- int ipc_listen(u16 port) Opens the IPC server on the specified port.

- int ipc_accept(int server_fd) Accept an incoming IPC connection on the IPC server.

- int ipc_write(int fd, void *buf, size_t count) Sends the specified number of bytes from the buffer to the IPC connection. This function is used both by Main Task and external applications.

- int ipc_read(int fd, void *buf, size_t count, int read_all) Receives at most 'count' number of bytes from the IPC connection to the buffer. If read_all set to 1 the function blocks until the exact count of bytes is received. This function is used both by Main Task and external applications.

- int ipc_u32_read(int fd, u32 *n) Reads one 32bit integer value from the IPC connection to 'n'.

- int ipc_u32_write(int fd, u32 n) Writes one 32 bit integer value from 'n' to the IPC connection.
**12.4.3.3 mt_ipc Utility APIs**

The IPC service is implemented as a Main Task utility, so it has the minimal default utility API. Note that the following APIs are used internally by Main Task. These APIs are listed for reference only, and should not be changed in any circumstance.

- **mt_ipc_init(void)** Initialize the rg_conf_ipc module, opens a TCP server socket and add it to the event_loop.

- **mt_ipc_uninit(void)** Remove the TCP server socket from the event_loop.

**12.4.4 Low-level Design**

The IPC service is implemented as a utility inside Main Task. The file implementing the service is `mt_ipc.c`. The TCP server is opened using a standard blocking socket and event_loop fd event functionality. When connection is accepted `mt_rg_conf_ipc` acts according to the high-level design using regular (blocking) read/write functions. Actual implementation of the TCP/IPC stack can be found in `pkg/util/ipc.c`. The current implementation supports only one-way messaging.

**12.4.5 Tutorial**

In this tutorial, you will use the IPC session to create and change the following custom OpenRG configuration entry:

`rg_conf/vendor/contact|name,email,phone`

1. Open an IPC session between the external application and OpenRG.

- `char *ipc_string_read(int fd)` Reads a string from the IPC connection. The required amount of memory is allocated in order to store the string. It is the responsibility of the user to free the allocated memory using the `free()` function.

- `int ipc_string_write(int fd, char *str)` Writes a string from the 'str' to the IPC connection.

- `int ipc_bind_listen_port(u16 port)` Opens a socket on the specified port for reading purposes. 'port' is in network order (call htons before calling this function).

- `int ipc_bind_listen_port(u16 port)` Reading data from a socket. This function should be used from the module task. The return value is a tally of how many bytes were read, or -1 for error (similar to the read function).

- `int ipc_write_port(u16 port, void *data, int data_len)` Writing data to a specified port. This function should be used from the module task. 'port' is in network order (call htons before calling this function). The return value is a tally of how many bytes were read, or -1 for error (similar to the write function).
Integrating Applications

fd = ipc_open();

Note:

• The IPC session is established between the external application and Main Task.
• OpenRG will halt its internal workflow until the external application terminates the IPC session (refer to step 8).

2. Request a portion of OpenRG’s configuration file (`rg_conf`) to the application’s local memory:
   ```c
   set = ipc_rg_conf_get(fd, "vendor/contact");
   ```

   In this example we are using an abstract 'vendor/contact' configuration entry. However, you can use this method to access any existing or user defined entry in the configuration file.

3. Print the current `rg_conf` entry value:
   ```c
   printf("Contact: Name:
         set_get_path_strz(&set, "name"),
        set_get_path_strz(&set, "email"),
        set_get_path_strz(&set, "phone"));
   ```

   Note:
   • No data will be printed when executing this command for the first time.
   • Refer to the `set_t` API reference for a detailed description of the `set_get_path_strz()` function.

4. Create or replace the existing configuration file entry value:
   ```c
   set_set_path_str(&set, "name", "John");
   set_set_path_str(&set, "email", "smith8@jungo.com");
   set_set_path_str(&set, "phone", "+123456789");
   ```

5. Update OpenRG’s configuration file according to the changes made by your external application:
   ```c
   ipc_rg_conf_set(fd, "vendor/contact", &set);
   ```

6. Reconfigure OpenRG according to your changes:
   ```c
   ipc_reconf(fd, FLASH_DELAYED_NOW);
   ```

   Note that during reconfiguration two significant actions are performed:

   1. The reconfiguration of devices and tasks to reflect set changes. For example:
      • A network device will be reconfigured according to a modified IP address.
      • The DHCP server task will be reconfigured according to a modified IP address lease range.

      Note that actual device/task reconfiguration performed only when relevant parameters are really changed and not on every invocation of the `reconf` function.
2. The reconfiguration can be committed to permanent storage at once, or after a specified delay. The second `ipc_reconf` parameter is the determines that there be a delay period between issuing the command and actual data transfer to the permanent storage. Use `FLASH_DELAYED_LOW_PRIORITY` to cause a delayed commit. Delayed commit enables several changes to be issued in a single write operation. It is advised to use this feature in order to protect Flash memory from unnecessary write operations, due to the physical limitation on the number of multiple accesses to the Flash.

7. Free the `rg_conf` sub-tree:

   ```c
   set_free(&set);
   ```

   Note that for any subsequent configuration file changes to take effect, a new IPC session must be established as described in step 1.

8. Close the IPC session:

   ```c
   ipc_close(fd);
   ```

   OpenRG will now continue its regular work-flow. Make sure not to keep IPC sessions opened for a long period of time, since this causes OpenRG to halt while waiting for the session to close. For example, if the Internet connection is disabled, it will not be restored until the IPC session is terminated.
13

Integrating Kernel Modules

13.1 Loading Modules During Boot Sequence

This section describes the `gen_load_kernel_modules` application, which you must modify when creating a new module. The `gen_load_kernel_modules` is an application that runs on the host during compilation of the OpenRG tree, and produces two files:

1. `load_kernel_modules.c`—implements all the OpenRG `insmod` functions (dynamic loading of modules).

2. `load_kernel.c`—links the kernel module with the kernel (static loading).
The `gen_load_kernel_modules.c` file, located under `rg/pkg/main` produces the `load_kernel_modules.c` and `load_kernel.c` files located under `rg/build/pkg/main`. The produced files include the following functions:

1. `rg_load_kernel_modules_initial()`—loads from user mode modules that must be loaded when OpenRG starts.
2. `rg_load_kernel_modules()`—loads or initiates modules both from kernel mode and user mode.

Both functions in both files include lists of `#ifdef` s of different configurations that call:

- In `load_kernel_modules.c`—`load_kernel_module()` with the module and optional parameters. For the definition of `load_kernel_module()`, refer to `rg/pkg/util/rg_system.c`.
- In `load_kernel.c`—configuration-specific `_init()` functions.

Both functions are called from the kernel in the `init` function (`rg/os/linux-2.4/init/main.c`). If you wish to create a new module, make sure you update `gen_load_kernel_modules.c` to include your new module, since the file is compiled with the image and is essential.

In a regular OpenRG distribution, include your new module using one of the following functions:

- `load_both_modes` – This function will load the module from kernel mode if `config` is defined, and from user mode if `config_MODULE` is defined.
  ```c
  load_both_modes("your_func_return", "your_init_func", "your_module", is_early, can_fail, "YOUR_CONFIG", NULL);
  ```

- `load_kernel_mode` – This function will load the module from kernel mode (`config` is defined).
  ```c
  load_kernel_mode("your_func_return", "your_init_func", is_early,
  ```
**load_user_mode** – This function will load the module from user mode (config_MODULE is defined).

```c
load_user_mode("your_module", is_early, can_fail, "YOUR_CONFIG", NULL);
```

The arguments for these functions are:

- **is_early** – Indicates that the module will be entered to `rg_load_kernel_modules_initial` and will be `insmod` in the earliest stages of OpenRG.

- **can_fail** – When set, the generated function will not return failure if this specific module fails loading, but will print a warning.

- **config** – The name of the CONFIG that enables your module.

- **Other arguments** – A comma-separated list of arguments for the module. The last argument should be NULL. Each argument should be either "-flag_name" or "name_of_variable", in which case the next argument should be "variable_value".

This will ensure that after you rebuild the image your module will be inserted each time the new image boots.

### 13.2 Storing Module Files In the Ramdisk

During OpenRG's boot sequence, the modules' object files are opened and read by the `insmod` utility. After all the modules are loaded to the kernel space, the object files are usually closed, and their file system is unmounted. There are cases, however, when a particular object file should always be available.

In general, a module's object file should not be removed if you have created a module, which is not loaded using the `gen_load_kernel_modules` utility. For example, there could be a case in which you may want the module to be loaded at some stage after the boot sequence is performed. To prevent removal of the object file from the Ramdisk, modify the Makefile located in the module's directory as follows:

```makefile
JMK_MOD_TARGET+=modulefile.o
JMK_RAMDISK_MODULES_PERMANENT_FILES+=modulefile.o
```

In this case, the `JMK_RAMDISK_MODULES_PERMANENT_FILES` variable replaces the `JMK_RAMDISK_MODULES_FILES` variable used to temporarily load the module object files. After modifying the Makefile, compile the image and burn it to the board.
14 Integrating Device Drivers

14.1 Adding Device Drivers

The following example is based on adding a network device driver to OpenRG, which is performed by adding a dev_add function call to the pkg/build/hw_config.c file. Before you add the function call, add your device to the dev_if_type_cfg_str device array in rg/pkg/build/device_config.c. The format should be as follows:

```c
{ DEV_IF_<device>, "DEV_IF_<device>" },
```

The dev_add function has the following format:

```c
dev_add(char* name, dev_if_type_t type, logical_network_t net)
```

Call dev_add with the following parameters:

- **name** Of type `char *` – The name of your device, usually a driver name followed by a unit number, for example eth0 for an Ethernet interface.
- **type** Of type `dev_if_type_t`, defined in `rg/pkg/include/enums.h` – The network interface type.
- **net** Of type `logical_network_t`, defined in `rg/pkg/mgt/dev/dev_type.h` – One of three default logical network types: Internal, External and DMZ.

An example:

```c
if (token_get("CONFIG_HW_ETH_WAN"))
    dev_add("rtl0", DEV_IF_RTL8139, DEV_IF_NET_EXT);
```

14.1.1 Adding Device Drivers to the Development Tree

1. Create a new directory as follows:
1. Copy the kernel module source to the new directory.

2. Create a Makefile for the kernel module, as described in Section 19.3.4.5.

### 14.1.2 Adding Device Drivers to the Image

In order to add a newly compiled module to the image, add the module name to JMK_RAMDISK_MODULES_FILES, as follows:

```
JMK_RAMDISK_MODULES_FILES+=<your_module>
```

For more information, refer to Section 19.3.4.8.

### 14.2 Defining Network Devices and Network Bridges

This section describes how to define network devices and network bridges in 'create config', when creating a new distribution (dist_config.c), or when defining new hardware specification (hw_config.c).

#### 14.2.1 Defining a Hardware Network Interface

Your hardware description, including your hardware's network interfaces is done in pkg/build/hw_config.c. To define the network interfaces of the hardware, add a dev_add() call for each network device you want to add. Devices can be either physical, such as Ethernet, or a fixed VLAN interface, for boards that have a single Ethernet interface with a hardware switch for LAN/WAN separation.

```c
/* Define a network interface for hardware. * name - the name of the interface, as it appears in rg_conf and * in ifconfig. * type - the type of the device. dev_if_type_t is defined in * pkg/include/enums.h. * net - define the network this device will represent - WAN/LAN/DMZ. */
void dev_add(char *name, dev_if_type_t type, logical_network_t net);
```

For example, the following is the definition of a box with two Ethernet controllers, one of type RTL and one of type EEPRO:

```c
1  token_set_m("CONFIG_8139TOO");
2  dev_add("rtl0", DEV_IF_RTL8139, DEV_IF_NET_EXT);
3  token_set_m("CONFIG_EEPRO100");
4  dev_add("eep0", DEV_IF_EEPRO100, DEV_IF_NET_INT);
```

Lines 1 and 3 control the compilation of the Linux kernel code for the device types. They affect the kernel code generation, but do not make these devices available for OpenRG. Line 2 defines an interface named 'rtl0', of type DEV_IF_RTL8139, which will act as an external, i.e WAN, interface. This will make it the default WAN interface for OpenRG. Line 4 defines
an interface named 'eep0' of type DEV_IF_EEPRO100, which will act as an internal, i.e. LAN, interface. OpenRG will automatically assign the address 192.168.1.1 for the first internal interface.

14.2.2 Defining a Network Bridge

A bridge is a software feature and is not hardware dependent, therefore it is defined as part of the distribution in pkg/build/dist_config.c.

```c
/* Create a network bridge.
 * name - The bridge interface name. Usually "br0".
 * net - Define the network this device will represent - WAN/LAN/DMZ.
 * ... - A list of char * enslaved devices, ending with NULL.
 */
void dev_add_bridge(char *name, logical_network_t net, ...);
```

For example, the following defines a LAN bridge over the Ethernet (ixp0) and a wireless interface:

```c
dev_add_bridge("br0", DEV_IF_NET_INT, STP, "ixp0", "wlan0", NULL);
```

14.2.3 Allowing Absent Devices

When one of the devices configured in create_config is missing, due to it being unplugged or malfunctioning, OpenRG normally reboots. To allow OpenRG to boot with an absent device, and keep the device in the 'disabled' state, call the void dev_can_be_missing(char *dev_name) function with the name of the device that is allowed to be missing.

14.3 Adding Network Device Types

OpenRG is preprogrammed to support a number of different network interface types. Among these interface types are Intel EtherExpress Pro, RTL, PPPoX, etc. If you are using hardware that is already supported by OpenRG, all you need to do is declare it, as specified in Section 14.2. When adding a new type of hardware, you will need to go through the following stages:

1. Create a new device type, by adding an enum to dev_if_type_t in pkg/include/enums.h.

2. Define device capabilities and operations by writing its 'ops' functions, including the IOCTL function. Refer to Section 14.3.2.

3. Add device ops to pkg/mgt/dev/dev_ops.c.

4. If the device is physical, and added in create_config, add a new option to the openrg_config_options array (in pkg/build/config_opt.c) in the form of CONFIG_RG_<device type>, i.e. CONFIG_RG_DEV_IF_EEPRO100. This option will be defined if the device is added using dev_add() in create_config.
5. Add the device to `dev_if_device_name_create()` located in `pkg/mgt/lib/mgt_utils.c`. For example:

```c
char *dev_if_device_name_create(dev_if_type_t type)
{
    type2template_t *item, list[] = {
        ...,
        {DEV_IF_MYDEV, "mydev", 0},
        {-1}
    };
    ...,
}
```

6. (Optional) If your code calls for a new device implementation that is not yet implemented by OpenRG, you may add new IOCTLs, rather than changing existing ones. Section 14.4 explains how to do this.

### 14.3.1 When Would You Add a New Network Device Type?

You would add a new network device type to OpenRG in the following cases:

- When adding a new hardware type.

- When each hardware type is a different device. Devices in OpenRG describe the hardware and its capabilities, the smallest capability difference calls for a different device. Luckily, adding devices is easy.

### 14.3.2 The dev_if_ops_t Structure

All interactions of OpenRG with your device are done through function pointers filled in the device's dev_if_ops_t structure. For example:

```c
dev_if_ops_t dev_if_uml_wlan_ops = {
    .type = DEV_IF_UML_WLAN,
    .ioctl = dev_if_uml_wlan_ioctl,
    .changed = dev_if_uml_wlan_changed,
    #ifndef CC_FOR_BUILD
    .reconf = dev_if_eth_reconf,
    .notify = dev_if_eth_notify,
    .dep_notify = dev_if_gen_dep_notify,
    .tryup = dev_if_eth_tryup,
    .init = dev_if_eth_init,
    .status_get = dev_if_eth_status_get,
    #endif
    ...
};
```

The following are examples of callbacks used in this structure:

- **changed** Checks if there is a need for device reconfiguration. To detect changes in `rg_conf` entries that are common to all Ethernet devices, point this callback to the generic function `dev_if_eth_changed()`. On the other hand, if you would like to check whether a specific device entry has been modified in `rg_conf`, use the COMP_SET() macro.

- **reconf** Reconfigures the device. To reconfigure the device according to changes in the common Ethernet device parameter(s), point this callback to the generic function
dev_if_eth_reconf(). IOCTLs may be used to modify a device driver's behavior according to the changes made in rg_conf.

**notify**  Notifies the device about a state change. The following is the prototype function of this callback:

```c
static void dev_if_uml_wlan_notify(dev_if_t *dev,
    dev_if_state_t from, dev_if_state_t to);
```

**dep_notify**  Notifies the device about a state change in one of its dependent devices. The following is the prototype function of this callback:

```c
static void dev_if_uml_wlan_dep_notify(dev_if_t *dev,
    dev_if_t *changed_dep_dev, dev_if_state_t from,
    dev_if_state_t to);
```

**tryup**  Tries to bring the device to a running state. The following is the prototype function of this callback:

```c
static void dev_if_uml_wlan_tryup(dev_if_t *dev);
```

For more information about device states, refer to Section 15.4.1.

### 14.3.2.1 dev_if_ops_t::type

type is the only variable in the dev_if_ops_t structure. It holds the device type, in order to identify the ops structure. Fill this field with a dev_if_type_t value, as defined in pkg/include/enums.h.

### 14.3.2.2 dev_if_ops_t::ioctl()

The device IOCTL implementation describes the device's capabilities and implements device-specific access functions, such as functions for returning its status, MAC address, maximum MTU, etc. IOCTL operations are defined in pkg/mgt/dev/ioctl_defines.h. All IOCTLs are implemented with their default return values in pkg/mgt/dev/dev_base.c, a virtual device, from which all other devices inherit their default values. The IOCTL function is comprised of a large switch statement, with each 'case' being a different IOCTL operation. Inheritance is achieved by not implementing a certain IOCTL, and adding a 'default:' statement that calls the IOCTL function of the virtual device you want to inherit from. For example:

```c
default:
    return dev_if_eth_ioctl(dev, op, res);
```

This makes the surrounding IOCTL function inherit from the generic Ethernet IOCTL. The most common IOCTL functions that you can inherit from are:

- dev_if_eth_ioctl – For Ethernet type devices
- dev_if_dsl_ioctl – For DSL/ATM link (physical) devices
- dev_if_wireless_g_ioctl – For wireless 802.1g devices
- dev_if_ppp_ioctl – For point-to-point devices and tunnels
- dev_if_base_ioctl – If no other IOCTL suites your needs
Note that all of the above IOCTL implementations inherit from `dev_if_base_ioctl`. You should inherit from the IOCTL function that closest matches your device, and implement only the IOCTL operations that are different in your IOCTL function. For example, the following defines the 'eep' Ethernet device, which is an Ethernet interface that has a default MAC address burnt into the chip:

```c
int dev_if_eep_ioctl(dev_if_t *dev, dev_if_ioctl_cmd_t op, void *res)
{
    switch (op)
    {
    case DEV_IF_HAS_HW_MAC:
        *(int *)res = 1;
        return 0;
    default:
        return dev_if_eth_ioctl(dev, op, res);
    }
}
```

The IOCTL function returns 0 for success, or -1 for failure. Note that OpenRG will reboot in case of an IOCTL failure. A pointer to the data that the IOCTL operation should fill is passed in 'res'. You should cast it to the appropriate data type before assigning.

### 14.3.3 Code Compilation for Localhost

The device infrastructure (`pkg/mgt/dev`) also compiles for localhost, in order to create the `gen_rg_def` executable. To allow `gen_rg_def` to compile, only the bare minimum of the device infrastructure should compile for localhost, in order to reduce function dependencies. You should protect with `#ifndef CC_FOR_BUILD` all code except for the device IOCTL implementation and the `dev_if_ops_t` 'type' and 'ioctl' member definition. Example:

```c
#ifndef CC_FOR_BUILD
void dev_if_eep_tryup(dev_if_t *dev)
{
    eep_up(dev);
}
#endif
int dev_if_eep_ioctl(dev_if_t *dev, dev_if_ioctl_cmd_t op, void *res)
{
    switch (op)
    {
    case DEV_IF_HAS_HW_MAC:
        *(int *)res = 1;
        return 0;
    default:
        return dev_if_eth_ioctl(dev, op, res);
    }
}
dev_if_ops_t dev_if_eep_ops = {
    .type = DEV_IF_EEP,
    .ioctl = dev_if_eep_ioctl,
    #ifndef CC_FOR_BUILD
    .tryup = dev_if_eep_tryup,
    ...
    #endif
};
```

### 14.4 Adding New IOCTLs

You may want to add new IOCTLs that OpenRG has not implemented, in cases in which your code calls for a new classification of devices. Try to use existing IOCTLs where possible,
as explained in Section 14.3.2.2. If you choose to write a new IOCTL, take into account that IOCTLs cannot fail. This means that you should not write a IOCTL that returns a property or a special value when that property is not applicable for a device.

Instead, create two IOCTLs, one to return if the device supports that property, and one to return the property itself. For example, you should not write a function that returns the device’s ATM link modulation type, or -1 if it does not support changing the modulation type. Instead, create two IOCTLs, DEV_IF_SUPPORTS_ATM_LINK_MODULATION_CHANGE and DEV_IF_ATM_MODULATION_GET. If the user calls DEV_IF_ATM_MODULATION_GET for a device that does not support it, the system will reboot.

### 14.4.1 Defining the IOCTL

There are two types of IOCTLs: those that work on a `dev_if_t *` and those that work on `dev_if_type_t`. Try to avoid adding IOCTLs that work on `dev_if_type_t`. In pkg/mgt/dev/ioctl_defines.h add the IOCTL definition. There are a number of macros you can use to define the new IOCTL:

- **DEV_IF_IOCTL_CMD_GET_CREATE(ret_type, func_name, ioctl_cmd)** Defines a IOCTL with the enum `ioctl_cmd`, and automatically creates an access function in the form of `ret_type func_name(dev_if_t *dev);`. For example, the following adds a IOCTL and creates
  ```c
  int dev_if_is_editable_mac(dev_if_t *dev):
  DEV_IF_IOCTL_CMD_GET_CREATE(int, dev_if_is_editable_mac,
  DEV_IF_IS_EDITABLE_MAC)
  ```

- **DEV_IF_IOCTL_CMD_TYPE_GET_CREATE(ret_type, func_name, func_no_type_name, ioctl_cmd)** Defines a IOCTL with the enum `ioctl_cmd`, and automatically creates the following two access functions for it:
  - A function that works on `dev_if_type_t` – `ret_type func_name(dev_if_type_t type)`.
  - A function that works on `dev_if_t *` – `ret_type func_no_type_name(dev_if_t *dev)`.

  For example, the following adds a IOCTL and creates two access functions:
  ```c
  char *dev_if_type_description_get(dev_if_type_t type)
  char *dev_if_description_get(dev_if_t *dev)
  ```
  ```c
  DEV_IF_IOCTL_CMD_TYPE_GET_CREATE(char *, dev_if_type_description_get,
  dev_if_description_get, DEV_IF_TYPE_DESCRIPTION_GET)
  ```

- **DEV_IF_IOCTL_CMD_SET_CREATE** Defines a IOCTL with enum `ioctl_cmd`, and automatically creates the `int func_name(dev_if_t *dev, set_type *data);` access function.
• **DEV_IF_IOCTL_CMD_ENUM_CREATE(ioctl_cmd)** Defines a IOCTL with enum ioctl_cmd. You must define its access functions manually in `pkg/mgt/dev/dev_ioctls.c`.

### 14.4.2 Implementing The IOCTL

IOCTLS are implemented in the devices that are capable of answering them, and in `dev_base`, from which all devices inherit. The `dev_if_base_ioctl()` function implements defaults for all IOCTLS. The IOCTL default can be either a value or an error, in case that the IOCTL has no logical default. For example, the `DEV_IF_IANA_TYPE` IOCTL default would be an error seeing each device has a distinct IANA type, and there is no default.
Part III  Advanced Development Concepts
# Table of Contents

15. System Architecture ................................................................. 165  
16. Main Task .............................................................................. 178  
17. Adding Applications to the Main Task .................................. 189  
18. Configuration Database ......................................................... 199  
19. The Build Mechanism ............................................................ 205  
20. Debugging ............................................................................. 224
15

System Architecture

This chapter will explore the OpenRG software architecture. It will shed light on the rules and principles that govern OpenRG, describe the components that comprise the system and the rationale presiding their interaction. The architecture will be illustrated by partitioning the system into its significant elements, and unfolding the methods of organization and integration used to forge a cohesive whole. In other words, the aim of this chapter is to communicate OpenRG's high-level design and provide the programmer with a broad system context.

15.1 Architectural Design Guidelines

Throughout its development, OpenRG has strictly maintained standards-based architecture, design and interfaces. Working within this design gives OpenRG several important advantages:

- **Robust development environment**
  Hardware vendors receive a familiar and widely documented development platform on which to develop drivers and add applications.

- **POSIX compliance**
  Standard well-documented APIs to applications. OpenRG uses the Linux POSIX APIs that enable you to add proprietary or third-party applications on top of OpenRG.

- **A standard driver interface**
  OpenRG maintains standard and well documented driver interfaces to enable support of standard devices.

- **Scalability**
  OpenRG’s standards-based design can easily keep up with advancing kernel versions, enabling smooth integration of new modules and technologies.

- **OS support**
OpenRG provides support for multiple operating systems. When adding support for an additional operating system, little or no work is needed on the applications, allowing the engineering effort to focus on porting the kernel and drivers.

15.2 Source Code to Image Mapping

Each piece of the OpenRG source code is mapped to a different area in the image. The OpenRG image consists of the ramdisk and kernel. The kernel contains all the kernel code. The ramdisk is divided into the cramfs and modfs. The cramfs contains the OpenRG application and external processes. The modfs contains all the kernel modules, which are dynamically loaded at runtime. The following figure demonstrates this mapping.

```
Figure 15.1 Source Code to Image Mapping
```

15.3 Component Partitioning

The following figure illustrates OpenRG's component partitioning.
15.3.1 Coordination Logic

OpenRG is a multi-threaded multi-process product. There are many kernel threads, as well as user mode processes.

The logic of the OpenRG management code (called `openrg`) is handled by `Main Task`. Main Task is a framework for running entities that provide services, and for controlling network devices. It is responsible for system initialization, reconfiguration, and communication of the reconfiguration events with the various entities, which in turn correspond with the functional tasks. Main Task is also responsible for coordinating dependencies between entities and network devices. For example, it coordinates between a DSL PPP link and a dependent underlying ATM device. Main Task is single-threaded. This characteristic has the following advantages:

- Synchronization issues between processes/threads are avoided. Tasks cannot be interrupted while processing—only when waiting for a callback.
- Context switches are minimized.
- Inter-Process Communication (IPC) is simplified—functional tasks communicate with the Main Task using a simple function call. Conversely, an ordinary multi-process design involves information packaging, IPC tunneling and parsing by the receiving side.
- Memory footprint is reduced by linking all of the modules together, eliminating the code needed for IPC, configuration and synchronization.

At the heart of the system lays the event loop. Main Task is designed as an event-driven application. Events are expired timers, network events (ready to read or write on a file...
descriptor) or an exception. Each component registers its own handlers for various events; this is usually done upon startup. Since events are used throughout the system, this does not appear explicitly in Figure 15.2. The Coordination Logic is divided into 2 parts:

- **Generic** – Always exists. The major generic part is the *Entities* mechanism, which serves as a uniform API between tasks and the core logic of the Main Task. Each registered entity is notified upon configuration changes, and may affect other entities. For more information, refer to Section 15.3.1.1.

- **Component Specific** – Belongs to specific components. For example, the TOD client entity, which is a part of the TOD client component, notifies the DHCP Server when a new time is received, so that it can update its lease status. Most components register themselves as an entity. Therefore, they are notified on configuration changes. For more information, refer to Section 15.3.1.2.

The following figure demonstrates the Main Task's core logic.

![Main Task’s core logic](image)

**Figure 15.3 Main Task's Core Logic**

### 15.3.1.1 Entities

An entity serves as a unified API between a task, the core logic of the Main Task, and the *Configuration Database*. An entity supports the following operations:

- **open** – The entity registers itself, so that it will be called on relevant events. In most cases, this operation is also responsible for task initiation.

- **changed** – When `rg_conf` is modified, the entity must find out if there is an impact on its managed tasks.

- **reconf** – The entity should reconfigure its tasks based on the old and new values of `rg_conf`.

- **close** – The entity should shut down its tasks.

For example, the DNS entity handles the DNS task according to OpenRG's `rg_conf` configuration database. This entity initiates the DNS task, and equips it with callback functions.
that enable the task to search for locally configured host names. These names are entered either manually by the user using the Web-based management, or by the DHCP server, which dynamically allocates IP addresses to computers on the network.

The architectural advantage of this design is that the task is unaware of the configuration database format, nor of the methods of polling the DHCP server for a host name—all of these responsibilities are left to the entity.

15.3.1.2 Tasks

A task is a set of functions that implement a certain protocol or service, for example PPP, DNS or HTTP. A task can be characterized as being self-contained and system independent. It does not rely on other tasks, configuration files or another kind of outside help. The OpenRG library, libopenrg, provides the task with rudimentary services such as an event dispatch loop, memory allocation and logging functions. The entire configuration that a task may require is imparted as parameters to its \texttt{open()} function. The task's genuine work is actually performed by the callback function, a pointer which is also given as a parameter to the \texttt{open()} function. It means that tasks are inherently portable and reusable. To illustrate, the Ping task can be employed in two different ways:

- The DHCP server may use the Ping task to test whether an IP address, it is about to lease, is already taken by pinging the address in question. The task parameters in this case will be a ping packet, a brief time-out reply and callback functions. All of these are implemented in the DHCP code.

- The Web-based management may use the Ping task to implement a 'Connectivity Diagnostics Test' interface screen. The task parameters will be the number of ping attempts, a ten second time-out and callback functions. Again, these parameters are implemented in the Web-based management code.

15.3.2 Configuration Database

Configuration Database is used by all components to store information and various system settings. Throughout this book, the term \texttt{rg\_conf} will be used to refer to the configuration database. For more information about the configuration database, refer to Chapter 18.

15.3.3 Devices

Devices are used to manage networking devices. Components that implement networking devices register handlers for the new device type. For more information about devices, refer to Section 15.4.

15.3.4 System Services

System Services include the following:
• **Error Logging (rg_error)** – A general logging mechanism. Each component may use the `rg_error` functions to log events, similar to the standard `syslog()` function. Some components register themselves in `rg_error`, and they are called later for each event report. For example, email notification is implemented by registering a mail function as its handler.

• **Language Support** – A general mechanism that allows dynamic language change. It is used by components wishing to display text to the user, typically in WBM pages. Each component may use the language support by registering a table of all strings in each of the supported languages. For more information, refer to Section 28.6.

• **Inter-Process Communication (IPC)** – Used for communication between the OpenRG process and other processes. For example, an L2TP daemon uses IPC to check the password of a specific user. IPC is implemented by listening on a predefined TCP port on lo interface; interested processes need to use either `ipc` or `mgt_client` libraries to send commands to the main process and receive the results.

### 15.3.5 Management Tasks

The management tasks export interfaces that allow monitoring and control from outside. Each component that wishes to be managed registers its own management functions in each of the management components. The following management components are defined:

• **Web Based Management (WBM)** – The main management method. It allows the user to control OpenRG using a web browser. Each component registers functions to display WBM pages and receive user inputs. Some pages are system-wide (e.g. login page) and are registered as part of WBM initialization, and others are component-specific (e.g. File Sharing) and are registered as part of the component initialization. For more information, refer to Chapter 28.

• **Command Line Interface (CLI)** – Used by the command-oriented management: console, telnet, ssh, remote management and JNET. Each component that wishes to be managed registers its own management functions (that will be called when the associated command string is requested) in the CLI component. Each component that wishes to manage OpenRG using CLI can use the CLI component to perform the requested commands. For more information, refer to Chapter 27.

• **TR-069** – A CPE WAN Management Protocol. Each component registers parameter get and set functions, which will be called when the ACS issues a parameter get or set command, respectively. For more information, refer to Chapter 37.

• **UPnP IGD/TR-064** – LAN side management protocols. These protocols have a predefined set of actions, which are already implemented, and use the generic devices interface and the firewall. For more information, refer to Chapter 38.

• **SNMP** – Implements its operation through the use of generic MIB structures. Each component implements its specific MIBs and adds the MIB structure to the SNMP agent. For more information, refer to Chapter 26.
15.3.6 OpenRG Components

Other (non-core) components comprise a set of the OpenRG tasks. Each of these tasks is responsible for a specific functionality. For example, OpenRG’s DNS Server component implements the DNS server functionality. The non-core components consist of the following:

- **Task** – Implements the functionality.
- **Entity** – Used for communication with the core logic of the Main Task.
- **Management** – WBM, CLI, etc.
- **Configuration Database** – API to rg_conf
- **Kernel modules** – Used primarily for interfacing with the kernel and/or hardware.

However, none of these parts are obligatory. For more information about components, refer to Part IV.

15.3.7 Kernel

The kernel can be either a modified Linux kernel or VxWorks. Note that some components depend on the Linux kernel functionality, and therefore do not work in VxWorks.

15.3.8 KOS/KNET

The KOS/KNET layer is an OS abstraction layer, allowing the kernel code to be reused with multiple operating systems. For more information, refer to Chapter 23.

15.4 Network Devices

Network devices are used to manage network device drivers. They are responsible for the following functions:

- Dependencies on other devices. For example, a PPPoE device depends on the underlying Ethernet device. This means that when the Ethernet device is down, the PPPoE device is down too.
- A common API for managing any networking device.

For more information on network devices, refer to Chapter 14.

15.4.1 Device States

A device can be in one of the following states:
• DEV_IF_ST_NOT_EXISTS – New devices start in this state, and deleted devices end in this state. Otherwise, a device must never be in this state.

• DEV_IF_ST_DISABLED – The device is disabled in rg_conf, and should not be enabled.

• DEV_IF_ST_DOWN – The device is enabled in rg_conf, but is still run-time disabled. This happens when a device depends on another device that is not up and running.

• DEV_IF_ST_UP – The device is run-time enabled, but its up sequence has not been finished yet, and it is not running.

• DEV_IF_ST_RUNNING – The device is up and running.

Devices change state through the use of the dev_if_notify() function. As an example, consider the following DHCP client scenario:

• A DHCP client task receives a lease from the DHCP server.

• A DHCP client task notifies the DHCP client entity of the new lease.

• A DHCP client entity changes device state to DEV_IF_ST_RUNNING.

State transitions are always performed gradually, each one changing to its adjacent state. For example, when a running device is disabled, its state is changed in steps: first from DEV_IF_ST_RUNNING to DEV_IF_ST_UP, then from DEV_IF_ST_UP to DEV_IF_ST_DOWN, then from DEV_IF_ST_DOWN to DEV_IF_ST_DISABLED.

15.4.2 Types of Devices

Device actions depend on the networking device they manage. For example, when managing an Ethernet device configured with a dynamic IP, a DHCP client entity can start only when the client receives a lease from the server. After receiving the lease, it changes the device state to DEV_IF_ST_RUNNING. On the other hand, a PPTP client device needs to start the PPTP client entity. To support all variants, each device has a type and a set of functions that perform a specific task for a specific device type.

The following is a description of several important device type-specific functions:

• notify – Perform operations that are needed when the device state is changed. For example, when an Ethernet device state changes from DEV_IF_ST_DOWN to DEV_IF_ST_UP, the function starts a DHCP client entity. When the device state changes from DEV_IF_ST_RUNNING to DEV_IF_ST_UP, the function closes the DHCP client entity.

• ioctl – Implementation of various query, get or set operations. An example query is "does this device support broadcast?". A get operation might be for a device speed. Statistics Reset is an example of a set operation.

• changed – Check if device parameters have changed.
• reconf – Reconfigure a device.

For more information on functions, refer to Section 14.3.2.

15.4.3 Network Devices in OpenRG

Network devices are represented by the dev_if_t structure. This struct contains all data relevant to a device, including the entities which are associated with it. Dependencies on other devices are represented by the depend_on_list field, which refers to a (possibly empty) list of devices, on which this device depends. A list of all devices is available in the global variable rg_devices.

Whenever a network device changes its state, it notifies other components. Components that wish to be notified register themselves by calling dev_if_state_notify_func_add(). An example of this use is DDNS. When a device for which DDNS was configured starts running, DDNS needs to start the update process through this device. To implement it, DDNS registers a private notify function. This function is responsible for starting the DDNS update process if called for the selected device, and if the notification is for a state change to DEV_IF_ST_RUNNING.

15.4.4 Debugging Devices

The most common method for monitoring and debugging a network device is tcpdump. You can compile an OpenRG image with the "CONFIG_RG_TCPDUMP=y" flag. The tcpdump application will be added to OpenRG's file system (under the bin/ directory), eliminating the need to fetch the application with protocols such as tftp. Instead, you will be able to run the application from a shell in OpenRG's CLI:

```
OpenRG> system shell
/# tcpdump
```

15.5 System Reconfiguration

Reconfiguration of OpenRG is performed each time any of the settings is modified. Many events may cause system reconfiguration, such as:

• A change made from one of the management interfaces: WBM, CLI, TR69, etc. For example:
  • The user changed a parameter in the WBM, and clicked the 'OK' button.
  • The service provider used TR-069 to modify OpenRG.

• A network event. For example, OpenRG has setup its WAN interface due to receiving a lease from a DHCP server.

• Network control. For example, OpenRG has provided a new IP address to a computer in its LAN.
OpenRG reconfiguration is always performed as follows:

- `rg_conf` is modified as needed.

- The system reconfiguration function, `openrg_reconf()`, is called. This function is responsible for saving the new configuration to the Flash (if needed), and for updating all relevant components.

For more information, refer to Chapter 18.

15.6 Packet Flow Through OpenRG

This section provides an overview of a typical packet flow through OpenRG. Although the description applies to a Linux kernel, the hooks mechanism and the flow of packets through the hooks is the same in VxWorks. In essence, OpenRG's packet flow differs from the standard kernel packet flow only by the existence of special hooks developed by Jungo, which are inserted in the Rx and Tx flows. These hooks are introduced in the following section, and described in detail in Section 23.2.1.

15.6.1 Knet Hooks – Introduction

The *Kernel-mode Network Hooks* (knet hooks) are building blocks of a generic, cross-platform packet processing mechanism. This mechanism enables you to register functions for processing network packets, either after the packets are received from a network device driver, or before they are sent to it. Packet processing functions may serve various purposes, such as:

- Network traffic filtering
- Packet statistics calculations
- Packet integrity checks
- Packet defragmentation
- Processing of special applications (such as DHCP, RTP)
- Network bridging

The **Rx** knet hooks are called after a network device driver prepares the `sk_buff` data structure holding the packet information—a step before the upper network layers (for example, the IP Stack) process this packet.

The **Tx** knet hooks are called when the network stack delivers the packet to a network device driver—a step before a packet is queued for transmission on a network device.
15.6.2 Rx Flow

The following diagram illustrates the flow of incoming packets.

![Diagram of Rx Process]

**Figure 15.4 Rx Process**

15.6.2.1 Network Driver

The incoming packets are handled by a network device in the following way:

1. A packet is received by the hardware.
2. An Rx packet ready interrupt is triggered.
3. The network driver schedules an Rx task to be executed.
4. The Rx task is invoked when the Interrupt Service Routine (ISR) is completed.

The Rx task process performs the following:

1. Fetches the packet's data from the hardware queue.
2. Checks for hardware errors.
3. Creates a containing logical structure for the packet, i.e. `sk_buff`.
4. Initializes some helper information in the `sk_buff`, e.g. protocol, Rx device.
5. Passes the packet up to the Linux kernel using `netif_rx()`.
6. Handles more packets if ready.

15.6.2.2 The Linux Kernel – Rx

The `netif_rx()` function performs the following:

1. Enqueues `sk_buff` to the system's input queue (sometimes referred to as the "backlog").
2. Schedules the dequeue task.

The dequeue task performs the following:

1. Dequeues a packet from the backlog.

2. Sets some more information in `sk_buff`, and manages some statistics.

3. Calls OpenRG Rx knet hooks for this packet:

   a. If the hooks mechanism's answer is "handled", packet processing is stopped, otherwise it continues normally.

   b. The hooks may modify every aspect of the `sk_buff` (especially the actual data and the "device" field) and still say "not handled". The OpenRG hooks scheme is explained in detail in Section 23.2.1.

   c. The kernel passes the packet to Linux routing.

15.6.2.3 The Linux Kernel – Routing

The Linux kernel decides on the packet's route according to:

- The packet's IP destination
- The routing table

If the destination is local, the Linux kernel passes the packet up to a higher protocol layer (e.g. TCP/UDP/ICMP). If the destination is remote, the Linux kernel forwards the packet according to the routing decision (destination). It sends the packet at the end of the routing chain, using `dev_queue_xmit`.

15.6.3 Tx Flow

The following diagram illustrates the flow of outgoing packets.

![Figure 15.5 Tx Process](image-url)
15.6.3.1 The Linux Kernel – Tx

1. `dev_queue_xmit` calls OpenRG’s Tx hooks for this packet. Hooks may "consume" or alter the packet. Such actions may be dropped by the firewall, perform NAPT, etc.

2. If the packet was not consumed by OpenRG hooks, it is enqueued to the device queue.

3. The Linux kernel schedules a dequeue task.

15.6.3.2 The Linux Kernel – Tx QoS

The devices queue is actually a tree structure of queues and classes to handle prioritized traffic and bandwidth control. The kernel enqueues the packet to the correct queue according to its priority. Dequeuing is intelligent and takes into account the bandwidth limit (dequeue is delayed when the limit is reached). Priority queues combined with intelligent dequeue provide the desired outcome—real traffic control. Dequeued packets are passed to the network driver for physical transmission.

15.6.3.3 Network Driver – Tx

The network driver performs the following:

1. Copies the packet's data to the hardware's Tx queue.

2. Signals the hardware to begin transmission.

3. Performs a cleanup of internal structures allocated for this packet.
16

Main Task

This chapter describes how to integrate the Watchdog feature, introduced in Chapter 5, to OpenRG's Main Task instead of using an external process. Each Main Task feature consist of two components: an entity and a task. The entity is a basic unit of operation within the Main Task which manages a specific function. Each entity manages a task that carries out its functionality. A task performs a certain function according to the configuration defined by the entity. The Main Task is the coordination logic under which all the entities run. In this chapter you will learn how to implement the Watchdog example as an asynchronous entity and task.

16.1 Compiling the Main Task Watchdog Tree

The first step, is to compile the Watchdog tree with a Main Task asynchronous approach:

1. Clean the existing build by running the 'distclean' target:

   $ make distclean

2. Add the advanced tutorial configuration token

   "CONFIG_RG_TUTORIAL_ADVANCED=y" to the build system file `rg/pkg/build/dist_config.c`, by adding the following line under your matching distribution:

   ```
   token_set_y("CONFIG_RG_TUTORIAL_ADVANCED");
   ```

3. Build and compile the OpenRG development tree:

   ```
   $ make config DIST=<distribution name> LIC=<license file path>
   $ make
   ```

   Wait for the compilation to complete. You now have an image with the Watchdog feature implemented as an asynchronous task.
4. Load the image to the board and access the Watchdog feature. You will see that its behavior is identical to the Watchdog behavior when it is implemented as a user-mode process. However, if you look under the /bin directory on the target, you will not find the watchdog program which is now part of the openrg program.

16.2 Understanding Asynchronous Programming and Main Task Architecture

The logic of the OpenRG management code (called openrg) is handled by Main Task. Main Task is a framework for running entities that provide services, and for controlling network devices. It is responsible for system initialization, reconfiguration, and communication of the reconfiguration events with the various entities, which in turn correspond with the functional tasks. Main Task is also responsible for coordinating dependencies between entities and network devices. For example, it coordinates between a DSL PPP link and a dependent underlying ATM device. Main Task is single-threaded. This characteristic has the following advantages:

16.2.1 The Asynchronous Nature of Main Task

Since OpenRG management is being handled in a single process which runs all the time, operations should never perform blocking actions. Blocking actions are handled by an event dispatch loop. Main Task provides a callback mechanism to enable actions to be performed while other actions wait for a result. The following is a description of the Main Task's general behavior:

1. Main Task runs a task for the first time by calling the task's open() function.

2. During the task's life cycle, it may need to wait for an external event to occur. This delay may be the result of the task waiting for a timeout or a packet.

3. The task registers a request for the desired event in the Main Task's event dispatch loop, specifying a callback function, and returns to its caller - usually Main Task.

4. When the registered event occurs, the dispatch loop invokes the callback function to handle the event.

16.2.2 The Task

A task is a set of functions that implement a certain protocol or service, for example PPP, DNS or HTTP. A task can be characterized as being self-contained and system independent. It does not rely on other tasks, configuration files or another kind of outside help.
It follows that tasks are inherently portable and reusable. The entire configuration that a task may require is imparted as parameters to its `open()` function. The task’s genuine work is actually performed by the callback function, a pointer to which is also given as a parameter to the `open()` function.

### 16.3 The Entity

An entity serves as a unified API between a task, the core logic of the Main Task, and the *Configuration Database*. An entity supports the following operations:

- **open** – The entity registers itself, so that it will be called on relevant events. In most cases, this operation is also responsible for task initiation.

- **changed** – When `rg_conf` is modified, the entity must find out if there is an impact on its managed tasks.

- **reconf** – The entity should reconfigure its tasks based on the old and new values of `rg_conf`.

- **close** – The entity should shut down its tasks.

For example, the DNS entity handles the DNS task according to OpenRG’s `rg_conf` configuration database. This entity initiates the DNS task, and equips it with callback functions that enable the task to search for locally configured host names. These names are entered either manually by the user using the Web-based management, or by the DHCP server, which dynamically allocates IP addresses to computers on the network.

The architectural advantage of this design is that the task is unaware of the configuration database format, nor of the methods of polling the DHCP server for a host name—all of these responsibilities are left to the entity.

### 16.4 Creating a New Directory and Modifying the Makefile

Start by creating a new subdirectory under `rg/pkg/watchdog` named `task` for the new Watchdog entity and task.

```
~$ cd ~/rg/pkg/watchdog
~/rg/pkg/watchdog$ mkdir task
```

As the new Watchdog does not need the process directory anymore, change the Makefile accordingly. Remove the `process` directory from the JMK_SUBDIRS variable and add the `task` directory instead. In addition, add to the JMK_L_OBJS variable, which is the Watchdog task object that you will create in the `task` directory.

```
...  
JMK_SUBDIRS+="task"
JMK_L_OBJS+="task/watchdog_task.o"
...  
```
16.5 Implementing the Watchdog Entity

To become familiar with working with Main Task and creating new entities and tasks, re-implement the Watchdog (previously implemented using a user-mode process) in OpenRG's Main Task architecture—entity and task. Create the file `mt_watchdog.c` under the task directory (`rg/pkg/watchdog/task`). As discussed previously, every entity is required to implement four functions—open, changed, reconf and close, in order to enable basic entity operations (see Section 16.3). Add these functions to the `mt_watchdog.c`:

```
rkg/pkg/watchdog/task/mt_watchdog.c
#include <main/mt_main.h>
#include <watchdog/mgt/watchdog_mgt.h>
#include "watchdogt.h"

/* Free entity memory, close task and unregister entity */
static void mt_watchdog_close(rg_entity_t *e)
{
    mt_entity_remove(e);
    free(e);
}

static reconf_type_t mt_watchdog_changed(rg_entity_t *e)
{
    return NO_RECONF;
}

static void mt_watchdog_reconf(rg_entity_t *e)
{
}

/* Standard OpenRG entity operations:
* - Changed: check if entity needs reconfiguring
* - Reconf: reconfigure the entity when one of its fields changes
* - Close: close the entity, free all resources
*/
static mt_ops_t watchdog_ops = {
    mt_watchdog_reconf,
    mt_watchdog_changed,
    mt_watchdog_close
};

rg_entity_t *mt_watchdog_open(void)
{
    rg_entity_t *e = (rg_entity_t *)zalloc_e(sizeof(rg_entity_t));
    mt_entity_add(e, &watchdog_ops, ENT_WATCHDOG);
    rg_error(LCONSOLE, "Watchdog task opened");
    return e;
}
```

Let's review the code:

- **mt_watchdog_open** The open function: at this point, the new entity only allocates a context and registers itself with the entity management mechanism. For further information regarding contexts and task designs, refer to Chapter 12.

- **mt_watchdog_close** The close function: frees the allocated resources if such exist, and unregisters the entity.

- **mt_watchdog_changed** The changed function: since there is no implementation yet, this function returns "no reconf needed".
**mt_watchdog_reconf**  The reconf function: there is no need for reconfiguration yet.

You can see that for now, the reconf and changed functions are empty. You will add content to them as you progress further into the example. The next step is to define the new entity ID in the entity enum in `rg/pkg/main/entity.h`.

/* This enum is used as an index to an array, so don't go wild * on the numbers... */
typedef enum {
    ENT_UNKNOWN = 0,
    ENT_DHCP = 1,
    ...,
    ENT_WATCHDOG = 90,
    ...,
    ENT_COUNT /* Always last, the number of entity types */
} rg_entity_type_t;

You have added the entity ENT_WATCHDOG enum definition as value 90. In case of adding a new entity, make sure it is added before ENT_COUNT, which is the marker for the last entry. Note that the entry already exists, you do not need to add it. Since you are going to add a task controlled by this entity, there is a need for a structure to hold the configuration parameters passed to the task. In the Watchdog case, you need to pass two parameters:

1. boot message
2. margin

You may recall from the discussion above that a task is not allowed to access the `rg_conf` or other tasks—it is the entity’s role. So you must provide the task with the means to trigger `rg_conf` changes as well as query its various fields. This is done by supplying the task with a callback function. The Watchdog task requires the `is_watchdog_enabled` callback, which queries the Watchdog `mgt` and returns 1 if the Watchdog sends the keep-alive signal, and 0 otherwise.

Create the task API header file called `watchdogt.h` (notice the ‘t’ notation to mark `task`) under the `task` directory. This header file will serve as the API between the entity, the task, and Main Task.

`rg/pkg/watchdog/task/watchdogt.h`

```c
#ifndef _WATCHDOGT_H_
#define _WATCHDOGT_H_
...
/* API parameters between the watchdog entity and the watchdog task */
typedef struct watchdog_params_t {
    int margin;
    msg_t boot_msg;
} watchdog_params_t;
/* API callbacks provided by the watchdog entity to the watchdog task */
typedef struct watchdog_cb_t {
    int (*is_enabled)(void); /* Is the watchdog task enabled? */
} watchdog_cb_t;
#endif
```

© 1998-2012 Jungo Ltd.
Now let’s go back to Main Task. Since the task will need to query the configuration parameters in the open and reconf functions, define a function that reads the configuration database using mgt, and build the task parameter structure:

**rg/pkg/watchdog/task/mt_watchdog.c**

```c
#include <watchdog/mgt/watchdog_mgt.h>
...
static void watchdog_param_get(watchdog_params_t *param)
{
    watchdog_conf_t conf;
    watchdog_conf_get(&conf);
    strncpy(param->boot_msg.msg, conf.boot_msg.msg, sizeof(msg_t));
    param->margin = conf.margin;
}
```

The new task will need to implement the open, close and reconf functions as well. These functions will be called from the Watchdog entity when needed:

1. **open**—called when the task is initially opened.
2. **close**—called when the task is closed.
3. **reconf**—called when OpenRG needs to reconfigure the task due to parameter changes in the configuration database.

The `watchdog_open` function will need to return a handle to its private context. This so called 'task context' is saved in the Main Task entity. Add the needed declarations to the task header:

**rg/pkg/watchdog/task/watchdogt.h**

```c
/* Handle for the watchdog task */
typedef struct watchdog_t watchdog_t;
watchdog_t *watchdog_open(watchdog_cb_t *cb);
void watchdog_close(watchdog_t *t);
void watchdog_reconf(watchdog_t *t, watchdog_params_t *param);
```

Line 4 is the task **open** function declaration and it gets the callback structure for future use. Add the opening of the Watchdog task to the `mt_watchdog` entity:

**rg/pkg/watchdog/task/mt_watchdog.c**

```c
static watchdog_cb_t cb = {
    watchdog_is_enabled,
};
rg_entity_t *mt_watchdog_open(void)
{
    rg_entity_t *e = (rg_entity_t *)zalloc_e(sizeof(rg_entity_t));
    e->t = watchdog_open(&cb);
    if (!e->t)
        goto Error;
    mt_entity_add(e, &watchdog_ops, ENT_WATCHDOG);
    return e;
Error:
    free(e);
    rg_error(LPANIC, "Failed opening watchdog task");
    return NULL;
```

© 1998-2012 Jungo Ltd.
The entity opens the task on line 10 and saves the handle in the entity context (e-->t). All future references to the task will use this context. You may also note that when failing to open the task, you should go to the error handler where the entity is destroyed and a NULL is returned to signal an error.

Line 2 declares the callback structure that will be passed to the task. Note that the callback function used here to test if the Watchdog is enabled is the \texttt{mgt} function \texttt{is_watchdog_enabled}. Sometimes, the API will not match, and you will have to write your own function that will call the \texttt{mgt} function.

The Main Task entity is almost complete. Remember that the \texttt{changed} and \texttt{reconf} functions are still empty. Add the needed code to the \texttt{changed} function:

```c
1     /* Check if there was a change in one of the parameters:
2      * - margin
3      * - boot message
4      */
5     static reconf_type_t mt_watchdog_changed(rg_entity_t *e)
6     {
7         set_t **old_set = &saved_rg_conf, **new_set = rg_conf;
8
9         if (COMP_SET(new_set, old_set, Swatchdog))
10            return NEED_RECONF;
11
12        return NO_RECONF;
13    }
```

The Watchdog entity's relevant fields are only those under the \texttt{rg_conf} path of \texttt{/watchdog}. Therefore, in line 9 you should recursively compare the new copy and the old copy of the \texttt{rg_conf} sets under the path. The function returns the value \texttt{NEED_RECONF} if the \texttt{COMP_SET} macro indicates that the sets are different, and \texttt{NO_RECONF} otherwise. If the sets are different, OpenRG will call the \texttt{reconf} function.

Implement the \texttt{reconf} function:

```c
/* Get the new params from rg_conf and apply them to the task */
static void mt_watchdog_reconf(rg_entity_t *e)
{
    watchdog_params_t param;
    watchdog_param_get(&param);
    /* Reconfigure the task with new parameters */
    watchdog_reconf(e->t, &param);
}
```

At this point, you have set up the ground for the task's functionality. Before going ahead and implementing the task, let's review all the functions that you implemented in the entity:

\textbf{mt\_watchdog\_open} Called when OpenRG loads, allocates resources, registers and opens the task.

\textbf{mt\_watchdog\_close} Called when OpenRG goes down, frees resources, closes the task and unregisters.

\textbf{mt\_watchdog\_changed} Invoked when a change is triggered in OpenRG, and checks if the relevant fields in OpenRG have changed.

\textbf{mt\_watchdog\_reconf} Invoked if the \texttt{changed} function returned a \texttt{NEED_RECONF} value. If so, it loads the new configuration parameters from \texttt{rg_conf} and calls the task reconfiguration function.
**watchdog_param_get**  A static helper function that loads parameters using Watchdog mgt into the configuration structure used as an API between the Main Task entity and the task.

The last step is to export the API to the **watchdog.h** file. Add the following declarations to the file:

```c
rg_entity_t *mt_watchdog_open(void);
```

### 16.6 Registering the Entity in Main Task

In order to initiate the entity, you need to register it with the Main Task during the startup sequence. To do so, enhance the **watchdog_init.c** init and uninit functions to register and unregister the entity with Main Task. You should add the following lines to register the `mt_watchdog_open` entity for system startup sequence and unregister it.

**rg/pkg/watchdog/watchdog_init.c**

```c
#include <main/mt_main.h>
#include <watchdog/task/watchdogt.h>
...
void watchdog_init(void)
{
    rg_error(LCONSOLE, "Initializing Watchdog components.");
    mt_system_up_register(mt_watchdog_open);
    ...
}

void watchdog_uninit(void)
{
    rg_error(LCONSOLE, "Un-initializing Watchdog components.");
    ...
    mt_system_up_unregister(mt_watchdog_open);
}
```

The two functions above define the operations needed to be taken when OpenRG is booting, and when it is about to shutdown. In the `init` function, you register the entity in the Main Task framework, by using the `mt_system_up_register` that gets the open function pointer. The `uninit` function uses the `mt_system_up_unregister` function in order to unregister the entity from Main Task.

### 16.7 Implementing the Watchdog Task

From the above implementation and the defined API, you know that the Watchdog task contains the following functions:

1. `watchdog_open`
2. `watchdog_close`
3. `watchdog_reconf`
The open function should allocate its internal context and act upon the parameters received from the entity to set up a polling timer event. To set up a timed event in OpenRG asynchronous programming, use the following function:

```c
void event_timer_set(double ms, etimer_func_t func, void *data)
```

The first parameter, 'ms', holds the time in milliseconds in which the timer will expire and the callback function `func` will be invoked. The third parameter, 'data', is an opaque pointer that will be handed to the callback function—you will use it as a private context. You can delete or cancel the timer at any time by calling:

```c
void event_timer_del(etimer_func_t func, void *data)
```

The first parameter, 'func', is the callback function that you have used when setting the timer, and 'data' is the opaque pointer used as the context. First, you define the task's inner context, which should have the following fields:

**Margin** The task keeps this variable to determine the polling period.

**Callback Structure** The task must save the callback functions provided by Main Task so it can invoke them when needed.

Define the task's inner context under the Watchdog directory, and create the Watchdog task implementation (watchdogt.c):

```c
/* Watchdog task context */
struct watchdog_t {
    int margin;
    watchdog_cb_t *cb;
};
```

Implement the open function. This function should allocate the private context and invoke the reconf function. Add the function as follows:

```c
/* Open the task. Allocate task context and start operation, return a handle to * the task context */
watchdog_t *watchdog_open(watchdog_params_t *param, watchdog_cb_t *cb)
{
    watchdog_t *t = zalloc_e(sizeof(watchdog_t));
    t->cb = cb;
    watchdog_reconf(t, param);
    return t;
}
```

The function returns the private handle to the entity so it may invoke the reconf and close functions when needed. Implement the close function. It should remove pending events (if such exist) and free allocated resources:

```c
/* Close task, free all task resources if any */
void watchdog_close(watchdog_t *t)
{
    /* Remove all pending events */
    event_timer_del(watchdog_poll_func, t);
    free(t);
}
```

With the open function in place, you can implement the watchdog_reconf function. The reconf function gets the parameters structure, watchdog_params_t, from the Watchdog entity. It applies them to the kernel module configuration using the openrg_module_ctrl function, as discussed in Section 5.7.4.

**rg/pkg/watchdog/task/watchdogt.c**
/* Re-configure task, save new parameters in task context and update kernel module */

void watchdog_reconf(watchdog_t *t, watchdog_params_t *param)
{
    msg_t msg;

    /* Save the margin in context to set the timer after every signal */
    t->margin = param->margin;

    /* Copy the message to send to watchdog module */
    strncpy(msg.msg, param->boot_msg.msg, sizeof(msg_t));
    rg_error(LCONSOLE, "Got new margin: [%d], t->margin);  
    rg_error(LCONSOLE, "Got new boot message: [%s], msg.msg);  

    /* Update the kernel module with the new parameters */
    openrg_module_ctrl(KOS_CDT_WATCHDOG, WATCHDOG_IOCTL_SIG_MARGIN, &t->margin);
    openrg_module_ctrl(KOS_CDT_WATCHDOG, WATCHDOG_IOCTL_BOOT_MSG, &msg.msg);
}

The function, however, is not completed yet. When it gets the parameters, you still need to remove the pending timer event and put a new one instead with the new margin. This issue will be discussed later.

Now you get to the most important part, the keep-alive polling. The keep-alive signal is derived directly from the margin. You need to send a keep-alive signal every half margin to stay on the safe side. As the first step, write the callback function for the keep-alive polling. What should it do?

- Check if the Watchdog is enabled.
- If so, send the keep-alive signal and set an event timer to the next margin half.

Add the poll function:

```c
/* The polling function, called every margin/2 and sends a keep-alive signal if the watchdog is enabled */
static void watchdog_poll_func(void *data)
{
    watchdog_t *t = data;

    /* If watchdog is enabled */
    if (t->cb->is_enabled())
    {
        /* Send the Keep alive signal to the kernel */
        openrg_module_ctrl(KOS_CDT_WATCHDOG, WATCHDOG_IOCTL_KEEPALIVE, NULL);

        /* Set the timer for the next event */
        event_timer_set(t->margin / 2 * 1000, watchdog_poll_func, t);
    }
}
```

Notice that line 8 uses the callback function to check if the Watchdog is enabled. The task is almost complete now. You still have to make the reconf function set the timer event when the parameters change.

Add the following code to the end of the reconf function to start sending the keep alive signal to the kernel:

```c
/* Start sending keep alive signal to the kernel */
watchdog_poll_func(t);
```

Place the completed task, watchdogt.c, in the task directory.
16.8 Writing the Makefile

The Watchdog entity, `mt_watchdog.o`, and the Watchdog task, `watchdogt.o`, should be compiled and combined into a single object file, as defined earlier in the Watchdog Makefile:

```
ifdef JMKE_ENV_INCLUDED
    JMK_ROOT=../../../
    include $(JMK_ROOT)/jmk/env_root.mak
endif

JMK_O_TARGET=watchdog_task.o
JMK_O_OBJS=mt_watchdog.o watchdogt.o
JMK_EXPORT_HEADERS_DIR=watchdog/task
JMK_EXPORT_HEADERS+=watchdogt.h
$(call JMKE_INCLUDE_RULES)
```

You are all set. Compile your new image, load it to the board and test it.
17

Adding Applications to the Main Task

17.1 Generic Task Design

17.1.1 Rules of Thumb

- A task must never communicate with any other task.

- A task must never read its own configuration data or rely on any source of information other than that defined in its APIs.

- A task is accessed only via its `<name>_open()`, `<name>_close()` , and `<name>_reconf()` functions. The task notifies of events via callbacks. If the task needs more information while running (such as access to a users database) it will do it via callbacks too.

- Tasks which use the `rg_conf` database must describe the fields they use in their API specification.

- All global parameters should be moved into the context structure.

17.1.2 Context and `<name>_t`

Each task that calls a blocking function gives a callback function, which will be called when the blocking operation finishes. Tasks need to save information across the function that started
the blocking operation and the callback. This information is stored in a `void *context`, which is passed to the function and given back in the callback.

When Main Task and another task call each other's functions, they both pass around contexts. From the task's point of view the Main Task's data is a `void *context`, and its own data is `<name>_t *<name>`. The context is the first parameter passed to the function. For callback functions that the task needs to call, the first parameter is the `void *context` of Main Task.

### 17.2 Generic Task API

#### 17.2.1 Task API by Example (PPP)

```c
/* PPP task state change notifications */
struct ppp_callbacks
{
    int (*ppp_if_up)(void *context, char *ifname);
    int (*ppp_if_down)(void *context, char *ifname);
    int (*ppp_user_info_get)(void *context, ppp_user_t *user);
    /* And possibly many many more... */
};
/* Starts PPP with ppp_parameters_t. User registers his callbacks
   * for interface state notification. Returns ppp_t handle.
   */
ppp_t *ppp_open(ppp_parameters_t *p, struct ppp_callbacks *cb, void *context);
/* Stops PPP in a graceful way, sending LCP TERM over the network. This
   * function can take time to complete
   */
void ppp_close_con(ppp_t *t, void (*ppp_close_end)(void *context),
                   void *context);
/* Stops the PPP task from executing. Frees ALL memory and resources,
   * and unregisters itself from ALL callbacks.
   */
void ppp_close(ppp_t *t);
/* Passes command line arguments for the PPP task for manual operation: This
   * function MAY call ppp_open() after parsing those arguments, or it can call
   * some PPP-related configuration tool.
   */
int ppp_main(int argc, char *argv[]);
/* Change task active parameters WHILE task is active */
ppp_reconf(ppp_t *ppp, new_ppp_parameters)
/*...and any other PPP API functions which can be called from anywhere */
```

`<name>_open()` The task open function receives the init parameters needed by the task. The task must immediately copy the structure, as it will probably be freed when the function returns.

It also receives a structure with callback functions that the task can call when it needs data or when it needs to announce state changes. Those callback functions are called with the `void *context` pointer given in the parameters.
The open function returns a task handle—a pointer to the task's context structure (NOT void *context). This handle will be used in subsequent calls to the task. It is used to differentiate between multiple instances of the same task.

<name>_close() and <name>_close_con()  The close_con function is applicable to tasks that handle networking traffic, such as PPP or VoIP. This function tries to shut down the protocol by sending the necessary communication to the peer, as when deregistering from a VoIP gateway. close_con does not free the task_t, and it is expected that the caller will call the task's close() function from within the close_con callback. The task close function receives the handle of the task instance to close and closes it immediately, freeing all resources, and unregistering from all callbacks. <name>_close() can be called from within one of the task's callbacks (refer to Section 17.3).

<name>_main()  The main function is used for command line parsing only. It can be used in one of two ways:

1. To start a configuration tool, or

2. To call the task open function interactively.

<name>_reconf()  The reconf function should be only implemented on tasks that can be reconfigured during run-time, without having to close and reopen all connections. At the early stages of development, this function will not be needed. For example, if the PPP user name has changed, one can simply close the PPP task and then start it again with the new parameters.

17.3 Writing <name>_close()

17.3.1 The Problem

A callback of a task calls (indirectly) to the close of the task itself. An example involving rmt_update and wget is used here to illustrate:

1. Remote update starts.

2. It creates a wget task to download the file (calls wget_open()), and gives it rmt_update_got_buffer() as a callback for the data downloaded.

3. A packet arrives to wget_got_packet() which calls the user-callback with the buffer: rmt_update_got_buffer().

4. rmt_update_got_buffer() determines that the header is invalid, and aborts the update by calling wget_close(), and returns.

5. The execution returns to wget_got_packet().

The call sequence and call stack:

packet arrives
17.3.2 What Callbacks Should be Protected?

There are two types of callbacks: information callbacks and action callbacks.

Information callbacks include iterators as well as functions used to lookup data in a database, to check the current line state or to get the MAC address of an interface. In such cases the callback function is not expected to make modifications related to `event_loop` either directly or indirectly (calling `<name>_close` of a task that has events registered, for example). Therefore, calls to information callbacks need not be protected.

Action callbacks include callbacks to handle send, recv, line up, line down and new lease. The function called needs to act upon this notification of the state change, or handle the incoming/outgoing data. When calling such a callback, the caller must know that the callback might close its own task. Therefore, calls to action callbacks must be protected, as described below.

17.3.3 The Solution

You need to:

1. Add a `<name>_t->closing` member to your task structure.

2. In the `<name>_close()` function, replace the call to `free(<name>_t)` with:
   
   `<name>_t->closing = 1;
   event_timer_set(0, free, <name>_t);

   It is important that "<name>_t->closing = 1;" be the first line in your <name>_close() function.

3. All calls to "action callbacks" in the `<name>` task should have the following immediately after them:
   
   `<name>_t->callback(<name>_t->context, data);
   if (<name>_t->closing)
      return -1;`}

Be sure not to do any operations after "action callbacks" if `<name>_t->closing`. Do not bother checking in the beginning of `<name>_close()` whether `<name>_t->closing` since this is a compile time bug, not a run time bug. If code is written correctly this should never happen, just as `free()` should never be called twice on the same pointer.

17.3.4 A Good Example

Return to the remote update/wget example used above.
Adding Applications to the Main Task

rmt_upd.c

```c
void rmt_update_got_buffer(void *context, u8 *buf, int len)
{
    rmt_upd_t *t = (rmt_upd_t *)context;
    if (check_headers(buf, len))
    {
        /* Bad headers */
        wget_close(t->wget);
        return;
    }
    do some more work;
}
```

wget.c

```c
static int wget_got_packet(wget_t *t, u8 *buf, int len)
{
    t->cb->recv(t->context, buf, len);
    if (t->closing)
        return -1;
    return estream_read(t->control_estream, PACKET_SIZE);
}

void wget_close(wget_t *t)
{
    t->closing = 1;
    if (t->control_estream)
        estream_close(t->control_estream);
    nfree(t->ackbuf);
    event_timer_set(0, free, t);
}
```

17.3.5 A Bad Example

```c
static int wget_got_packet(wget_t *t, u8 *buf, int len)
{
    t->cb->recv(t->context, buf, len);
    /* the following line will crash... */
    return estream_read(t->control_estream, PACKET_SIZE);
}
```

Why is this bad? First, it accesses non-valid memory `t->control_estream` (it can be freed if the callback calls `wget_close()`). Second, it registers a callback in the event loop, and as a result the task will continue to run. If it does not crash due to bad memory access, it will continue calling `rmt_update_got_packet()`, even though it was closed.

17.4 Porting Processes to Tasks

The following must be considered when converting a daemon into a task:

- What are the configuration options or command line parameters? These need to be mapped to C structures.
- What needs to be saved in the task's context structure?
- What are the blocking operations? If the daemon uses select, then there is probably less work in porting to `event_loop`. 
Adding Applications to the Main Task

• How does the task handle memory allocation? Does it free its memory? How will you find all memory to free in the task's close() function (which can be called in every async breakpoint)?

• How will the task communicate with main task? If the task needs to call the user back, then the user has to provide a callback function and a context pointer in task_open(). If the user needs to be able to call the task, then we should add a function such as task_reconf() to the task. Do not use messages for communication between tasks; they should only be used for communication within a single task. Keep the APIs C function-based.

• If a task was select-based it should be ported using the events library (wrap the select with event_fd).

17.5 Naming Conventions

• Use the task naming conventions as described in Section 17.2.1.

• All of the task's functions should start with the task name (<name>).

• Data structure names should be of the form 'object_t'.

17.6 Asynchronous Programming

17.6.1 Purpose

The purpose of this section is to outline the various features which can be found in the event and estream libraries. This guide is for programmers who intend to write single-threaded code using the event and estream libraries.

17.6.2 Writing Asynchronous Code

Writing asynchronous code (ASYNC) usually revolves around setting triggers for events and then handling those events. Any function that can block execution (waiting for information from somewhere else, normally from the network) receives a callback function pointer from its caller. The ASYNC function then performs its work asynchronously, and calls the callback (notifying the caller) when the work is done.

17.6.3 Examples

17.6.3.1 A Simple Write Example

SYNC:

int say_hello(int fd)
Adding Applications to the Main Task

ASYNC: This is a fictional example; async_write does not exist.

```c
void say_hello(int fd, void (*end_func)(int ret, void *context),
              void *context)
{
    say_hello_t *o = malloc(sizeof(say_hello_context));
    o->func = end_func;
    o->context = context;
    o->fd = fd;
    async_write(fd, "hello", 5, say_hello_end, o);
}

void say_hello_end(int ret, void *context)
{
    say_hello_t *o = (say_hello_t *) context;
    if (ret<0)
    {
        printf("failed writing hello\n");
        ret = -1;
        goto Exit;
    }
    printf("wrote hello to network on fd %d\n", o->fd);
    ret = 0;
Exit:
    o->func(ret, o->context);
    free(o); /* use say_hello_free() if a complex structure */
}
```

Note: Notice how a private context (say_hello_t) is used to share data between say_hello() and say_hello_end(); the first function allocates it, and the second one frees it.

17.6.3.2 A Simple estream with States Example

SYNC:

```c
int read_message(int s, int fd)
{
    int len;
    char msg_buf[MSG_MAX_SIZE + 1];

    len = read(s, (void*)msg_buf, MSG_MAX_SIZE);
    msg_buf[len] = '\0';
    fprintf(fd, "Read (%d): %s\n", len, msg_buf);
}
```

ASYNC: This is a real life estream solution.

```c
int read_message(estream_t *estream, int fd, void (*end_func)(int ret,
                  void *context), void *context)
{
    read_message_t *o = (read_message_t *)malloc(sizeof(read_message_t));
    if (!o)
        return -1;
    o->func = end_func;
    o->context = context;
    o->fd = fd;
    estream_push(estream);
```
In this example you converted a read call on a stream (TCP socket) to an estream read. Notice that the estream (wrapping the TCP socket) is passed around and has its callback changed for the next operation. In order to save the former callback, the estream stack feature is used and the old user callback and data (and status as well) is pushed down the stack. Then you can place a new user callback function, without overwriting the old one. Once through, the new user function is popped and data and the older ones get back. Popping the data frees it, naturally.

Notice the naming convention as well—the function which handles the logic has a logical name (read_message). Then there is read_message_end, which gets called on the end of the operation, and read_message_free, which will only get called if the estream is closed before read_message_end is called. And a general comment: when using TCP/UDP sockets in your ASYNC code, make sure that they are set to be non-blocking. This can be done with fcntl().

17.6.3.3 A Comprehensive Loop Example

SYNC:

```c
int peers_send_message(peer_t *peer_list)
{
    int ret = 0;
    message_t *msg = message_alloc();
    if (!msg)
    {
        ret = -1;
        goto Exit;
    }
    message_init(msg);
    for (ptr = peer_list; ptr; ptr = ptr->next)
        peer_send_message(ptr, msg); /* Blocking call */
    message_uninit(msg);
}
```
Adding Applications to the Main Task

```c
Exit:
    nfree(msg);
    return ret;
}

ASYNC:

typedef void (*ret_func_t)(int ret, void *o);

typedef struct {
    peer_t *ptr;
    message_t *msg;
    ret_func_t func;
    void *context;
    int ret;
} peers_send_message_t;

/* Prototype needed for cyclic function calls */
static void peers_send_message_loop1(void *context);

static void peers_send_message_end(peers_send_message_t *o)
{
    message_uninit(o->msg);
    free(o->msg);
    o->func(o->ret, o->context);
    free(o);
}

static void peers_send_message_loop2(int ret, void *context)
{
    peers_send_message_t *o = (peers_send_message_t *)context;
    /* perform loop step */
    o->ptr = o->ptr->next;
    /* set up another iteration */
    event_timer_set(0, peers_send_message_loop1, o);
}

static void peers_send_message_loop1(void *context)
{
    peers_send_message_t *o = (peers_send_message_t *)context;
    if (o->ptr) /* The loop condition went here */
        peer_send_message(o->ptr, o->msg, peers_send_message_loop2, o);
    else
        peers_send_message_end(o);
}

void peers_send_message(peer_t *peer_list, ret_func_t ret_func,
                        void *context)
{
    peers_send_message_t *o;
    o = (peers_send_message_t *)zalloc_l(sizeof(peers_send_message_t));
    if (!o)
        goto Error;
    o->func = ret_func;
    o->context = context;
    o->ret = 0;
    o->msg = message_alloc();
    if (!o->msg)
        goto Error;
    message_init(o->msg);
    o->ptr = peer_list /* Initialization of the loop */
    peers_send_message_loop1(o);
    return;
Error:
    nfree(o);
    ret_func(-1, context);
}

Pay attention to the fact that the second iteration will be invoked by an event requested by
peers_send_message_loop2(), and not by a direct function call. This will prevent
stack overloading with function calls.
```
17.6.4 Convention Notes

- Use 'o' for the object you are working with in the function. This object holds all the data that you will be using in the function, and will be referenced many times, so a short name like 'o' is best. Also being systematic by using 'o' enables you to easily see this is the main object of the ASYNC call.

- When you need to hide underlying ASYNC function calls, do malloc for a new context. Make sure you do not 'skip' levels off indirection. For example, if (a) calls (b) which calls (c), do not have (c) call (a) upon completion. Cascade back the same way.

- If there is a user callback, always allow the user to provide a context pointer.

- Name the functions according to their roles. Refer to Section 17.6.3.2 for an example.

- Freeing the context and calling o->func should occur in <name>_end(). If the need to free the context and call o->func occurs in more than one place, then <name>_end() should only free the context and call o->func.

17.6.5 Synchronous Vs Asynchronous API

<table>
<thead>
<tr>
<th>SYNC</th>
<th>&lt;-&gt;</th>
<th>ASYNC</th>
</tr>
</thead>
<tbody>
<tr>
<td>usleep, sleep</td>
<td>&lt;-&gt;</td>
<td>event_timer_set</td>
</tr>
<tr>
<td>write</td>
<td>&lt;-&gt;</td>
<td>estream_write</td>
</tr>
<tr>
<td>read</td>
<td>&lt;-&gt;</td>
<td>estream_read</td>
</tr>
<tr>
<td>select (on an fd)</td>
<td>&lt;-&gt;</td>
<td>event_fd_set</td>
</tr>
<tr>
<td>signal</td>
<td>&lt;-&gt;</td>
<td>event_signal_set</td>
</tr>
<tr>
<td>accept</td>
<td>&lt;-&gt;</td>
<td>estream_accept</td>
</tr>
<tr>
<td>connect</td>
<td>&lt;-&gt;</td>
<td>estream_connect_&lt;name&gt;</td>
</tr>
</tbody>
</table>

Table 17.1 event_timer, efd, estream and esignal API
18

Configuration Database

Previous chapters have acquainted you with OpenRG's design. For example, Chapter 15 describes OpenRG's architecture and logic, while Chapter 7 deals with its practical operation. This chapter will take you further into OpenRG's internals, describing its configuration structure and mechanism.

18.1 System Configuration Structure

18.1.1 The Configuration File (rg_conf)

OpenRG's configuration file, `rg_conf`, utilizes a Set system configuration structure, also known as a 'Sets Database' (SDB). The SDB is based on a tree data structure, where each node has a name and can either hold a data value (the tree leaves) or have one or more sons. Accessing a set is done using a 'path'—a slash-separated string where each value between slashes corresponds to a set in the tree.

The SDB supports the following actions: searching a path, creating a new path, assigning a value to the corresponding path, comparing two sub-trees to one another, duplicating a sub-tree and the indexing of sons. The SDB can be serialized from a tree-structure into a readable string buffer, where a left parenthesis represents a new set. The buffer can then be saved by the user. Buffer un-serialization is also supported, in order to reconstruct the SDB tree structure from the string buffer. To view the `rg_conf` contents, open a Telnet connection to OpenRG, and type `conf print /`, as follows:

```
$ telnet 192.168.1.1
Username: 
Password: ******
OpenRG> conf print /
```

The data is always stored in the SDB in string format (char *value):

```
struct set_t {
```
18.1.2 The RAM Configuration File (rg_conf_ram)

After OpenRG starts up according to its rg_conf configuration, temporary operational data is generated. Such data are Firewall and IPSec dynamic rules. This data is stored in rg_conf_ram—a random access memory buffer. Since this data changes dynamically and is only relevant for a limited amount of time, it is volatile and will be deleted when OpenRG is turned off.

Similar to a PC's RAM memory, the rg_conf_ram data is lost every time OpenRG is restarted. Thus, OpenRG's configuration at any given time is comprised of the persistent information in rg_conf and the volatile information in rg_conf_ram. To view rg_conf_ram contents, open a Telnet connection to OpenRG, and type conf ram_print as follows:

```
$ telnet 192.168.1.1
Username: admin
Password: *****
OpenRG> conf ram_print /
```

18.2 The Configuration Entries

The rg_conf SDB contains many entries, all listed in the Configuration Entries Guide, which can be downloaded from http://www.jungo.com/openrg/doc/5.5/configuration_guide/pdf/openrg_configuration_guide.pdf. For each entry, a brief description of its purpose, as well as its type, relevance, and default value, are provided.

18.3 Obscuring Data

There is secret information kept on the Flash (in rg_conf), such as login passwords. This information should not be trivial to view, but on the other hand needs to have simple encryption and decryption. The obscurity functions simply serve to make viewing the information non-trivial, yet they are not true security, since anyone who knows the algorithm can view the data.

18.4 Booting OpenRG

Every time OpenRG is restarted, the boot sequence checks for an rg_conf section. If one is not available, it produces a new one by cross-referencing between the information in rg_default, a string of default configuration parameters that resides in OpenRG's image, and the Factory Settings, the aforementioned string containing specific serial data that resides on the Flash image.
As part of OpenRG's initialization process, the **rg_conf** section is copied from the Flash image into two instances on the board's memory: **rg_conf** and **saved_rg_conf**. These instances are used for OpenRG's real-time reconfiguration procedure, as described in the sections to come. Should a restart fail, the boot sequence will attempt to restart OpenRG again. When a counter located in **system/boot/failure_boots**, reaches three consecutive restart attempts between which the timeframes did not exceed 30 second periods, OpenRG will overwrite **rg_conf** by restoring factory defaults before attempting to restart.

### 18.5 Restoring Defaults

OpenRG's defaults can be restored either from the 'Maintenance' menu item of the 'System' tab, or by using the CLI command **system restore_default**. An additional method is by deleting the **rg_conf** section from the Flash image. In all of these cases, a new file will be produced as described in Section 18.4, overriding any previous file (should one exist).

### 18.6 Reconfiguring OpenRG

OpenRG is designed to operate under changing circumstances. For example, you can activate network devices, or perform system-wide adjustments, which you would like to keep in case OpenRG is restarted. The state of OpenRG's various entities is stored in the configuration database—the **rg_conf** file. This file is initially copied from the Flash or generated during the boot sequence. Any persistent change in the system state modifies the contents of this configuration file.

Every entity, for example WBM, DHCP, Firewall or VPN, that changes the system state by modifying **rg_conf**, triggers Main Task's **openrg_reconf()** function. Main Task checks with all entities whether the change affects them, and then invokes the entity's **reconf** function to apply the change. The **reconf** function compares and copies the changed **rg_conf** section to the **saved_rg_conf** section. Once they are identical again, the **rg_conf** is saved to the Flash as well. The reconfiguration process described above is identical for volatile changes in **rg_conf_ram**, with the exception of writing to the Flash.

### 18.7 Modifying the Configuration File

Modifying the **rg_conf** file is a basic operation and is achieved using the following CLI commands:

- **conf set**
- **conf reconf**

The **conf set** command parameters are the path and the value of a specific entry in OpenRG's configuration file. After setting the required values using **conf set**, use **conf print** to view the current value of the entry you set in **rg_conf**. This can be used for monitoring and debugging your work. The **conf print** command can also be used for
checking a specific value at any other time. Use `conf reconf` to activate OpenRG with the new set of values.

Note: Before executing `conf reconf` you must execute the complete batch of `set` commands. For example, to create a new PPP connection you need to set the connection name, type, protocol, password, etc. Only then can you execute `conf reconf`.

The `conf reconf` command can also be executed by modules of OpenRG that are currently active—for example, when a DHCP client receives an IP address from the DHCP server. Keep this in mind to prevent such occurrences from interfering with your work.

You can assign a priority to an entity's configuration update. This is especially useful, for example, if several entities are waiting for reconfiguration. In this case, you can assign a priority to each of them, according to their importance, in order to lessen the load imposed on the system by the concurrent reconfiguration requests. Depending on the set priority, you can instruct the system to perform an entity's entry reconfiguration immediately, in several minutes, or within the next hour. To do so, add one of the following `flash_delay` parameters to the `conf reconf` command, according to the desired priority:

- **1** – Now
- **2** – Top priority
- **3** – High priority
- **4** – Low priority

**An Example**

Set OpenRG's DNS hostname to `OpenRG_DEMO`:

```
OpenRG> conf set dns/hostname OpenRG_DEMO
Returned 1073773404
OpenRG> conf reconf 3
Returned -1
```

## 18.8 Modifying RGLoader's Configuration

Modifying RGloader's configuration file is a basic operation and is achieved using the `bset` CLI command. This command has the following options:

- **url**  The location where the image is stored. The following URL is used if no location is supplied to the 'load' command line.
  
  `default = tftp://192.168.1.10/openrg.img`

- **autoboot**  Defines whether to automatically load an image from the Flash. Autoboot is enabled by default (set to '1').
**network_boot** Defines whether to automatically load an image from a LAN host's TFTP server. To enable this option, perform the following:

1. Run the following command in RGloader's prompt:
   ```
   OpenRG boot> bset network_boot 1
   ```

2. Specify the URL of your LAN host's TFTP server (where the image is located), unless the default URL is used. For example:
   ```
   OpenRG boot> bset url tftp://192.168.1.30/openrg.img
   ```

3. Save the changes in RGloader:
   ```
   OpenRG boot> flash_commit
   ```

Note that if you wish to manually boot OpenRG from the network (without enabling the network_boot option), run the following command in RGloader's prompt:

```
OpenRG boot> boot -u <TFTP URL>/openrg.img
```

**timeout** The number of seconds to wait before autobooting. The default value is set to 3 seconds. During this boot delay, you may cancel the pending automatic boot sequence, by pressing the Esc key. You can also change the autoboot delay period. In the following example, the autoboot delay is set to 5 seconds:

```
openrg boot> bset timeout 5
openrg boot> flash_commit
```

### 18.9 Troubleshooting the Configuration File

Manually changing the **rg_conf** settings in a wrong way may cause OpenRG to hang. In this case, you may restore the default configuration settings by deleting the faulty **rg_conf**. But prior to doing so, you may wish to find out what caused this malfunction. You can generate a dump out of the incorrect configuration data, which is stored on one of the Flash sections. In order to make the data readable, you should convert the dump to a binary file, and obtain a utility that will process it.

Such a utility is **rg_conf_inflate**, located in `rg/build/pkg/tools`. This utility converts the binary dump output to the **rg_conf** text file. The following are instructions for creating an **rg_conf** dump and for processing it by the **rg_conf_inflate** utility.

**Note:** The following procedure implements the `cat` and `xxd` utilities, which are provided with all Linux distributions.

To create an **rg_conf** dump, perform the following:

1. Calculate the CONF section's dump size:
   a. In OpenRG's CLI, run the `flash_layout` command. The board's Flash sections are listed.
Note: There are two Flash sections containing OpenRG configuration settings. To identify the section on which the faulty `rg_conf` is stored, look at each section's "Counter" value. A section containing the newer data has a higher "Counter" value.

b. Locate the target CONF section. Assuming that Section 03 is the relevant CONF section, consider the following `flash_layout` command output:

```
...  
Section 03 Type CONF       Range 0x01520000-0x01540000 MaxSize 0x0001FF6C
    Size 0x00002498 Name 'rg_conf'
    Checksum 0x00000000 Start Offset 0x00000000
...  
```

The dump size is the sum of the section's 'Size' value, and the values of `sizeof(flash_section_header_t)` and `sizeof(int)`. Note that the sum of the `sizeof(flash_section_header_t)` and `sizeof(int)` values usually equals 148 (0x94). Therefore, the dump size in the current example is the sum of 0x2498 and 0x94, which equals 0x252C.

2. Enter the following command to generate the dump:

```
OpenRG> flash_dump -s 3 -l 0x252C
```

3. Remove all the redundant information from the dump:

- Lines before and after the `flash_dump` CLI command
- The first address column
- The last text column
- All white spaces

This could be done by executing the following command (assuming that the word "dump" appears in the CLI output):

```
$ cat dump | cut -d: -f2 | cut -d'|' -f1 | sed -e 's/ //g' > dump.str
```

When the dump contains only the information you need, convert it to a binary file by running the following command:

```
$ xxd -r -p <stripped_dump_filename> <binary_output_filename>
```

To convert the binary dump output to an `rg_conf` text file, perform the following:

1. Change to the `rg/build/pkg/tools` directory.

2. Run the following command:

```
rg_conf_inflate <binary_output_filename>
```
19

The Build Mechanism

OpenRG is built by a flat make system, utilizing a single Makefile. This concept yields a more efficient and relatively short compilation process. In addition, this make system utilizes a multi-processor environment automatically, running simultaneously on multiple processors for maximum efficiency. This chapter explains OpenRG’s build mechanism and is aimed at helping you write Makefiles using the OpenRG Makefile rules.

19.1  The Build Process

The OpenRG build process is aimed at creating two files:

openrg.img  An OpenRG image loadable via a serial connection.

openrg.rmt  An OpenRG image loadable via the Web-based management. For more information, refer to the ‘Upgrading the Gateway’s Firmware’ section of the OpenRG Administrator Manual. These files contain the OpenRG Linux kernel and the Ramdisk. The Ramdisk contains all the user-mode executables, shared objects, kernel modules and extra files (Web images, configuration files, etc.). This section describes the two stages of the Build process—the tree configuration stage and the ‘make’ stage.

19.1.1  The Tree Configuration Stage

The command for this stage is: make config DIST=<distribution name> During this stage, the configuration of the tree is created. Additional initial tasks, such as the creation of the auto-generated files, links, etc., are performed. The first tasks are performed from the root Makefile:

•  rg/pkg/build/create_config is built, and creates the following files:
• `rg/rg_config.mk`

• `rg/rg_config.h`

• `rg/pkg/util/rg_c_config.c`

• `rg/pkg/include/rg_config.h` - linked to `rg/rg_config.h`

For more information, refer to Section 19.4.

• The compiler wrapper `rg/pkg/build/rg_gcc` is built. OpenRG uses a cross tool chain for building for many architectures. The `rg_gcc` is a wrapper that decides which `gcc` and basic compilation flags to use.

• The `libc` and kernel headers are copied to `rg/pkg/build/include`. The OpenRG build can use both `uclibc` and `glibc`. In order to simplify the `–I<PATH>` compilation flags, the `libc` headers are copied to `rg/pkg/build/include`. They must contain the kernel headers, which are therefore copied to:

  • `rg/pkg/build/include/linux`

  • `rg/pkg/build/include/asm`

  • `rg/pkg/build/include/asm-generic`

The next step is to lower 'make config' to the subdirectories (JMK_SUBDIRS):

• JMK_ARCHCONFIG_SUBDIRS – In order to decide which subdirectories need 'make config', the build system checks the JMK_ARCHCONFIG_SUBDIRS variable. If there is no such variable in the Makefile, then JMK_ARCHCONFIG_SUBDIRS=JMK_SUBDIRS. If a Makefile does not specify the JMK_ARCHCONFIG_SUBDIRS, the default JMK_SUBDIRS are used.

• JMK_ARCHCONFIG_FIRST_TASKS, JMK_ARCHCONFIG_LAST_TASKS – If certain tasks are required in the 'make config' stage, they will be added either to JMK_ARCHCONFIG_FIRST_TASKS or to JMK_ARCHCONFIG_LAST_TASKS. The difference between the two is that JMK_ARCHCONFIG_FIRST_TASKS are performed before descending to JMK_ARCHCONFIG_SUBDIRS, and JMK_ARCHCONFIG_LAST_TASKS are performed after descending to JMK_ARCHCONFIG_SUBDIRS.

• JMK_EXPORT_LIBS – Libraries are either archives (*.a) or shared objects (*.so). These files are needed in the linkage stage of TARGETs (executable). Each JMK_TARGET that uses a library needs to specify its name and path. In order to use only one path for all libraries, the libraries are copied to `rg/pkg/lib`. By specifying `JMK_EXPORT_LIBS+=x.a`, the build system will copy `x.a` to `rg/pkg/lib`.

• JMK_EXPORT_HEADERS – When a C file includes a header file, it must specify the correct `–I<PATH>` of the file in the compilation flags (JMK_CFLAGS). Many files include
many headers from various locations in the tree, causing the command line to be very long. In order to simplify the JMK_CFLAGS, all headers used in multiple C files are copied to \texttt{rg/pkg/include}. A Makefile indicates that headers must be copied to \texttt{rg/pkg/include}, by adding them to JMK_EXPORT_HEADERS.

- JMK_LINK_DIRS – Not all directories are converted to OpenRG’s Makefiles. Such directories must be copied into the \texttt{build} directory (JMKE_BUILDDIR). JMK_LINK_DIRS instructs the build system to do so.

### 19.1.2 The Make Stage

The command for this stage is: \texttt{make}. The main stage of the build process is comprised of the following steps:

1. 'make all'—the libraries, modules, targets and kernel are built:
   - \texttt{libc} is built, which is the basic library used by all other TARGETs.
   - \texttt{lib_rg_c_config} is built, which is the library containing the \texttt{vcCONFIG} variables that are created during the 'make config' stage.
   - The subdirectories \texttt{vendor} and \texttt{pkg} descend to JMK_SUBDIRS.

2. All the files that are built for the target during the 'make all' process are copied into the Ramdisk image. This is OpenRG’s equivalent to 'make install'. OpenRG’s Ramdisk is built under \texttt{rg/build/pkg/build/disk_image}. This directory has three subdirectories:
   - \texttt{cramfs_dir} A read-only file system containing all executables, libraries, WBM images, etc. (all targets).
   - \texttt{ramdisk_dir} A read/write mirror image of \texttt{cramfs_dir}, but containing pointers to its executables (instead of the executables themselves). This results in a dramatically smaller directory, which also contains writable volatile subdirectories (e.g. \texttt{/tmp}). Volatile means that they are erased each time the power is turned off.
   - \texttt{modfs_dir} Contains all kernel modules, and is freed when all 'innsmod' operations are completed.

\texttt{rg/pkg/build} is the last directory in the 'make ramdisk' stage, because it runs the \texttt{make_ramdisk.sh} script that does the final preparation of the Ramdisk image.

3. The next step is building the kernel under \texttt{rg/os/linux}.

4. The last step is creating the images (\texttt{openrg.img} and \texttt{openrg.rmt}) that contain the kernel and the Ramdisk. In addition, a link to the \texttt{.rmt} file is created, providing distribution-specific information about the file, in the following format: <product name>-<version>-<distribution (in capital letters)>.

For example: openrg-4.0.2-UML.rmt
19.2 The Multi-distribution Build Concept

The multi-distribution build system enables the same source tree to be configured to more than one distribution at a time, thus omitting the need to run 'make distclean' when switching between configurations. When compiling a certain distribution in the OpenRG tree, the compilation's output is created in a separate output directory.

The distribution's output directory is an exact copy of the source tree. Every file in the output directory is a pointer to its respective source file. Performing changes in source files automatically changes the corresponding output files of all existing distributions. When a certain package is not converted into OpenRG's Makefiles, using the multi_dist build may be problematic. All the sources must be copied to the output directory in order for it to build.

19.2.1 A Simple Make

```
rg.dev$ make config DIST=UML && make
```

- The `rg.dev/build.UML` directory is created, containing the output of the compilation (`*.o`, auto-generated, links, images, etc.).
- A link named "build" is created in `rg.dev/`, pointing to `build.UML`.

19.2.2 Compiling Two Different Distributions

The following example tests the code changes after building DIST=UML, on a COYOTE platform.

```
rg.dev$ make config DIST=COYOTE && make
```

- The `rg.dev/build.COYOTE` directory is created, containing the output of the compilation.
- At this point, the following components exist simultaneously:
  1. `build.UML`—the output directory for the UML compilation.
  2. `build.COYOTE`—the output directory for the COYOTE compilation.
  3. `build`—a link that points to `build.COYOTE`.

19.2.3 Switching Between Distributions

The ability to switch between different distributions allows you to test a fix done in `rg/pkg/main`, for example, on both the UML and the COYOTE trees that have been built. Currently, the build points to `build.COYOTE`. Perform the following:
main$ make

The main directory is compiled to the COYOTE tree. To compile the UML distribution, perform:

main$ make BUILD=build.UML

This will change the build to point to build.UML, and then compile rg/pkg/main to the UML tree. Each 'make' you run from now on will be aimed at the UML tree. To change back to COYOTE, perform:

main$ make BUILD=build.COYOTE

This will change the build to point to build.COYOTE, and then compile rg/pkg/main to the COYOTE tree. Each make you run from now on will be aimed at the COYOTE tree.

19.2.4 Different Configurations of a Single Distribution

You may want to build DIST=UML CONFIG_SSI_PAGES=y without deleting the build.UML directory, in order to check DIST=UML both with and without the CONFIG. The output directory is created according to the name of the distribution, therefore when performing the following line, the new configuration will override the last configuration in build.UML:

rg.dev$ make config DIST=UML CONFIG_SSI_PAGES=y

If you would like to create a new UML distribution without overriding the older one, you can change the default behavior of the JMKE_BUILDDIR creation according to the DIST, by passing the BUILD parameter in the 'make config' command line:

rg.dev$ make config DIST=UML CONFIG_SSI_PAGES=y BUILD=build.UML-SSI && make

So far there are three compiled trees: build.UML, build.COYOTE, build.UML-SSI, where build points to build.UML-SSI. Note that reconfiguring without running 'make distclean' may work, but is not recommended. For example:

$ make config DIST=UML && make # Creates build.UML
$ make config DIST=UML CONFIG_SSI_PAGES=y && make # Reconfigures build.UML

19.2.5 Defining the Build Directory Outside the Tree

In cases such as the source tree residing on a slow hard disk or network drive, it is recommended to create the build directory outside the source tree. This requires using the full path name:

rg.dev$ make config DIST=IXP BUILD=/mnt/fast_hard_disk/my_ixp_dir

19.2.6 Parallel Builds

You can open two different shells and compile the tree simultaneously to different build directories, as long as the BUILD argument is supplied to the 'make' command or exported in the shell. For example:
• export BUILD=build.UMLmake config DIST=UMLmake

• BUILD=build.IXP make config DIST=IXPBUILD=build.IXP make

Note that if the output of both compilations is written to the same disk drive, parallel build is not recommended, as it may result in longer build time than two consecutive single builds.

19.3 Makefile Examples

19.3.1 A Simple Makefile

ifndef JMKE_ENV_INCLUDED
  JMK_ROOT=../../
  include $(JMK_ROOT)/jmk/env_root.mak
endif

JMK_TARGET+=example
JMK_O_OBJS+=example.o

$(call JMKE_INCLUDE_RULES)

The Makefile in this example creates an executable that runs on the board, referred to as a JMK_TARGET.

Lines 1-4 The first four lines exist in every Makefile. JMK_ROOT is the location of the root of the tree. jmk/env_root.mak contains all the variables of the OpenRG build system, and must be included prior to any other Makefile code.

Line 6 This line instructs the build system to create an executable for the target (board) "example".

Line 7 This line informs the build system that the example target is built from one object file named example.o.

Line 9 The last line of every Makefile is the inclusion of jmk/rg_root.mak. This file contains all OpenRG Makefile rules. The default rules (that reside in jmk/compile_root_end.mak) build and compile example.c to example.o, and then link example.o to example.

19.3.2 Compiling Multiple Targets

ifndef JMKE_ENV_INCLUDED
  JMK_ROOT=../../
  include $(JMK_ROOT)/jmk/env_root.mak
endif

JMK_TARGET+=ex1 ex2
JMK_O_OBJS_ex1+=a.o b.o
JMK_O_OBJS_ex2+=x.o y.o

$(call JMKE_INCLUDE_RULES)

The Makefile in this example creates an executable for the targets ex1 and ex2. Note that these targets depend on the existence of the pairs a.c, b.c and x.c, y.c, respectively.
Line 5 ex1 is built from a.o and b.o.

Line 6 ex2 is built from x.o and y.o.

19.3.3 Compilation Flags

Compilation flags (JMK_CFLAGS) are used when compiling a C file to an object file. For certain compilations, a specific flag may be added:

```bash
ifndef JMKE_ENV_INCLUDED
    JMKE_ENV_INCLUDED=1
endif

JMK_ROOT=../../
include $(JMK_ROOT)/jmk/env_root.mak
endif

JMK_TARGET+=ex1 ex2
JMK_O_OBJS_ex1+=a.o b.o
JMK_O_OBJS_ex2+=x.o y.o

JMK_CFLAGS+=-DEXAMPLE_ONLY

$(call JMKE_INCLUDE_RULES)
```

Line 8 This line adds -DEXAMPLE_ONLY to the compilation flags. All objects will compile with this flag.

19.3.3.1 Specific Compilation Flags

```bash
ifndef JMKE_ENV_INCLUDED
    JMKE_ENV_INCLUDED=1
endif

JMK_ROOT=../../
include $(JMK_ROOT)/jmk/env_root.mak
endif

JMK_TARGET+=ex1 ex2
JMK_O_OBJS_ex1+=a.o b.o
JMK_O_OBJS_ex2+=x.o y.o

JMK_CFLAGS_a.o+=-DEXAMPLE_ONLY

$(call JMKE_INCLUDE_RULES)
```

Line 8 This line adds -DEXAMPLE_ONLY to the compilation of a.o only. All other objects are built using the default JMK_CFLAGS.

19.3.3.2 Removing Compilation Flags

```bash
ifndef JMKE_ENV_INCLUDED
    JMKE_ENV_INCLUDED=1
endif

JMK_ROOT=../../
include $(JMK_ROOT)/jmk/env_root.mak
endif

JMK_TARGET+=ex1 ex2
JMK_O_OBJS_ex1+=a.o b.o
JMK_O_OBJS_ex2+=x.o y.o

JMK_CFLAGS+=-DEXAMPLE_ONLY
JMK_CFLAGS_REMOVE_a.o+=-DEXAMPLE_ONLY

$(call JMKE_INCLUDE_RULES)
```

Lines 8-9 In this example, all objects except for a.o, are built with the -DEXAMPLE_ONLY flag.
19.3.4 Compilation Targets

When a certain object file is used for more than one JMK_TARGET, it is added to a library. When the TARGETs are built, they are linked with this library. A library can be either an archive file (.a) that is used for static linking, or a shared object (.so) that is used for dynamic linking.

19.3.4.1 Building an Archive File

JMK_A_TARGET is an archive file used for static linking. JMK_LOCAL_A_TARGET is used by LOCAL_TARGETs.

```bash
ifndef JMKE_ENV_INCLUDED
    JMK_ROOT=../../
    include $(JMK_ROOT)/jmk/env_root.mak
endif

JMK_A_TARGET+=libex.a
JMK_O_OBJS+=a.o b.o
$(call JMKE_INCLUDE_RULES)
```

Lines 4-5 In this example, an archive named `libex.a` is built. It is created from `a.o` and `b.o`.

19.3.4.2 Building a Dynamic Library

JMK_SO_TARGET is a dynamically linked archive file.

```bash
ifndef JMKE_ENV_INCLUDED
    JMK_ROOT=../../
    include $(JMK_ROOT)/jmk/env_root.mak
endif

JMK_SO_TARGET+=libex.so
JMK_O_OBJS+=a.o b.o
$(call JMKE_INCLUDE_RULES)
```

Line 4-5 In this example, a shared library named `libex.so` is built. It is created from `a_pic.o` and `b_pic.o`. Notice that although the instruction is `JMK_O_OBJS+=a.o b.o`, the created files are `a_pic.o` and `b_pic.o`. Object files for shared objects are built with the `fpic` flag, and are therefore named with the `_pic` suffix.

19.3.4.3 Building a Local Host Application

JMK_LOCAL_TARGET is an executable that assists building the `openrg.img` by creating an application that runs on the host computer (rather than on the target board). LOCAL_TARGETs are used for creating auto-generated files, compressing the image and more.

```bash
ifndef JMKE_ENV_INCLUDED
    JMK_ROOT=../../
    include $(JMK_ROOT)/jmk/env_root.mak
endif

JMK_LOCAL_TARGET+=ex
JMK_O_OBJS+=a.o b.o
```
Line 4-5 In this example, an executable that will run on the host and not on the built target. Notice that although the instruction is JMK\_O\_OBJ\$+=a\_o b\_o, the objects from which exe is built are named local\_a\_o and local\_b\_o.

19.3.4.4 Building a Host and a Target Application

In order to compile a target for both the target machine and a local machine, add the target to the JMK\_CREATE\_LOCAL variable:

```
JMK\_TARGET = sample
JMK\_O\_OBJ\_sample = x.o y.o z.o
JMK\_CREATE\_LOCAL += sample
```

The output of the Makefile will be:

- A sample linked out of x.o, y.o and z.o, and compiled using $(CC).
- The local\_sample argument, linked out of local\_x.o, local\_y.o and local\_z.o, using $(CC\_FOR\_BUILD).

JMK\_A\_TARGET targets that are added to JMK\_CREATE\_LOCAL, are named liblocal\_%.a:

```
JMK\_A\_TARGET = libtest.a
JMK\_O\_OBJ\_libtest.a = a.o b.o c.o
JMK\_CREATE\_LOCAL += libtest.a
```

The output is a libtest.a library and a liblocal\_test.a local target.

19.3.4.5 Building a Kernel Module

JMK\_MOD\_TARGET is a variable that defines a kernel module. A kernel module is kernel code that is not compiled into the kernel, but rather added during run-time to the kernel using the 'insmod' command.

```
ifndef JMKE\_ENV\_INCLUDED
  JMK\_ROOT=../../
  include $(JMK\_ROOT)/jmk/env\_root.mak
endif

JMK\_MOD\_TARGET+=ex\_mod.o
JMK\_O\_OBJ\_a = a.o b.o x.o
JMK\_OX\_OBJ\_a = x.o

$(call JMKE\_INCLUDE\_RULES)
```

Line 4-6 Module objects are created differently for kernel 2.4 and kernel 2.6. In this example, if the kernel is 2.4 then a kernel module named ex\_mod.o is built from a\_mod\_24.o, b\_mod\_24.o and x\_mod\_24.o. If the kernel is 2.6, a kernel module named ex\_mod.o is built from a\_mod\_26.o, b\_mod\_26.o and x\_mod\_26.o. The source files are a.c, b.c and x.c. x.o exports symbols and therefore is added to JMK\_OX\_OBJ\$ and not JMK\_O\_OBJ\$.

Note: When compiling a module externally to the OpenRG tree, and the module uses Linux's exported symbols, the module insertion might fail with the error 'unresolved symbol' in insmod due to the export symbol filter mechanism. To avoid this, either
compile it as part of the OpenRG tree, or ensure that your OpenRG image is compiled with `CONFIG_RG_KERNEL_NEEDED_SYMBOLS=n`. This mechanism, when enabled, reduces the footprint by removing unused kernel symbols (functions) from the image.

### 19.3.4.6 Building a Static Kernel Module

```c
ifndef JMKE_ENV_INCLUDED
   JMK_ROOT=../../
   include $(JMK_ROOT)/jmk/env_root.mak
endif

JMK_MOD_TARGET+=ex_mod.o
JMK_O_OBJJS+=a.o b.o
e
JMK_MOD_2_STAT+=ex_mod.o

$(call JMKE_INCLUDE_RULES)
```

In this example, `ex_mod` is statically linked with the kernel, but not as a module. This means that 'insmod' will not affect it because it is already compiled into the kernel.

### 19.3.4.7 Building an Open-source Directory

The following example assumes that the current directory is `rg/pkg/name`, and that the following is the Makefile:

```c
ifndef JMKE_ENV_INCLUDED
   JMK_ROOT=../../
   include $(JMK_ROOT)/jmk/env_root.mak
endif

JMK_SUBDIRS+=a b c
JMK_LINK_DIRS+=c

$(call JMKE_INCLUDE_RULES)
```

When using an open-source package, you must convert its Makefiles to those of OpenRG, otherwise the sources of the package will not be found (by its original Makefiles). To resolve this, copy the directory of the package to the `build` directory. This is done using the `JMK_LINK_DIRS` variable. In the above example the `c` directory will be recursively copied to the `build` directory.

Note that the copy occurs during the 'make config' stage, so if you add `JMK_LINK_DIRS` to a Makefile, you must first run 'make config'. Note that executing 'make -C pkg/name/c' is not possible in this case, since the Makefile of the `c` directory is not an OpenRG Makefile, and therefore does not move to the `build` directory to run the compilation. The only option is to execute 'make -C pkg/name' in order to build `rg/pkg/name/c` correctly.

### 19.3.4.8 Ramdisk

To copy a file to the Ramdisk, the source file must reside in the current directory. If it is in a subdirectory, a Makefile should be added there, and the 'make' command should recurse into that directory:

<table>
<thead>
<tr>
<th>JMK_RAMDISK_LIB_FILES:</th>
<th>Copied to ramdisk under /lib directory.</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMK_RAMDISK_ETC_FILES:</td>
<td>Copied to ramdisk under /etc directory.</td>
</tr>
</tbody>
</table>
The Build Mechanism

The Build Mechanism

The Build Mechanism

If the file does not reside in one of the above directories, it can be added to the variable JMK_RAMDISK_FILES using the form src__dst. This is done by copying pap-secrets from the current directory, as /etc/ppp/pap-secrets:

ifndef JMKE_ENV_INCLUDED
JMKE_ROOT=../../
include $(JMKE_ROOT)/jmk/env_root.mak
endif

JMK_TARGET+=sample
JMK_RAMDISK_FILES+=pap_secrets__/etc/ppp/pap-secrets
JMK_RAMDISK_BIN_FILES+=sample
JMK_RAMDISK_MODULES_FILES+=sample
JMK_RAMDISK_TASKS+=ramdisk_hostfs
ramdisk_hostfs:
 $(call RAMDISK_LN_FUNC,/dev/ubd0,$(JMKE_RAMDISK_DEV_DIR)/root)
 $(MKDIR) $(JMKE_RAMDISK_RW_DIR)/mnt/hostfs
 $(call JMKE_INCLUDE_RULES)

Line 7 pap-secrets is copied to /etc/ppp/pap-secrets in the Ramdisk.

Line 8 sample is copied to the Ramdisk /bin directory.

Lines 10-13 The /dev/root link is created, pointing to /dev/ubd0. The /mnt/hostfs directory is also created.

• Files that are copied to /lib and /bin are completely emptied when copied to the Ramdisk.

• JMK_RAMDISK_MODULES_FILES are partially emptied (the symbol table is retained) when copied to the Ramdisk.

19.3.4.9 Linkage

JMK_LIBS_<target> contains full path libraries needed for <target>. For libraries that are not part of the rg-tree or libc, use __local_<lib>. This will add -l<lib> to the linkage flags but will not add any dependency.

JMK_LIBS_local_test:=$(JMKE_BUILDDIR)/pkg/util/liblocal_jutil.a $(JMK_LIBS_local_test)

local_test will be linked with the library liblocal_jutil.a.

JMK_LIBS_local_test:=__local_resolv $(JMK_LIBS_local_test)

This line will add -lresolv to the linkage flags.

19.4 Configuration Files

The first task of the Build process is the creation of the configuration files:

rg_config.mk, rg_config.h, rg_e_config.c These files contain OpenRG's configuration variables (CONFIGs).

rg_config.mk A Makefile included in envir.mk, which informs all OpenRG Makefiles which configuration to use.
rg_config.h  A header file included in OpenRG's source files, which defines the CONFIGs and cCONFIGs, and declares extern int/char * vcCONFIGs.

rg_c_config.c  This source file defines the vcCONFIG variables. It is used to create librg_config.a.

All the CONFIGs that can be used in OpenRG reside in the openrg_config_options array (in pkg/build/config_opt.c). Each CONFIG option has flags that define its type. A CONFIG with no flags is a regular CONFIG, which may either have the value 'y' or no value. For example, let us assume that CONFIG_NAME is a regular CONFIG with no flags. It is declared in openrg_config_options array as follows:

{ CONFIG_NAME, NULL },

If you do not turn CONFIG_NAME on, it will have no entry in any of the configuration files. If you set CONFIG_NAME=y in the configuration, rg_config.mk will contain the following line:

CONFIG_NAME=y

rg_config.h will contain the line:

sect_rg_config_files#define CONFIG_NAME 1

rg_c_config.c will have no entry for this variable. In our C code you can use pre-compiler macros:

#ifdef CONFIG_NAME

However, you cannot use C code:

if (CONFIG_NAME)

In this case, if the CONFIG is turned off, there will be no definition of this macro, and the compilation will fail. In order to use our CONFIG in regular C code and not only in pre-compiler macros, two more types of CONFIGs have been added—the cCONFIG and the vcCONFIG. A CONFIG with the OPT_HC flag will also create the cCONFIG and vcCONFIG of this CONFIG. For example, let us say that CONFIG_NAME is declared with OPT_HC. It is declared in the openrg_config_options array as follows:

{ CONFIG_NAME, NULL, OPT_HC },

If you do not turn CONFIG_NAME on, it will have two entries in rg_config.h:

#define cCONFIG_NAME 0
extern int vcCONFIG_NAME;

In addition, CONFIG_NAME will have an entry in rg_c_config.c:

int vcCONFIG_NAME = 0;

In the C code you can use pre-compiler macros:

#ifdef CONFIG_NAME

You can also use:

if (cCONFIG_NAME)

or

if (vcCONFIG_NAME)

You can use the above because even if the CONFIG is turned off, cCONFIG and vcCONFIG are defined (with the value 0). The major difference between cCONFIG and vcCONFIG is the fact that cCONFIG is a pre-compiler macro, while vcCONFIG is a C variable. The advantage of a C variable is that an object file, compiled with a vcCONFIG value, can have its value
changed after it is distributed. The value in a compiled object cannot be changed with the pre-
compiler cCONFIG.

You should therefore always use vcCONFIG instead of cCONFIG when possible. Makefiles
may run shell scripts, such as make_ramdisk.sh, which must be aware of the CONFIG
values. If all the CONFIG variables are exported, some shell commands will exit with an error
stating that the environment list is too long. In order to avoid this error, only CONFIGs that
are used by scripts such as make_ramdisk.sh are exported. To export a CONFIG, add the
OPT_EXPORT flag into the openrg_config_options array:

{CONFIG_RG_MODFS, NULL, OPT_EXPORT},

19.5 Debugging Makefiles

There are several methods for debugging the OpenRG Makefiles:

• Using the 'make debug' option. When adding '-p' to the 'make' command, it will print all the
targets being built. The output is verbose and hard to follow, but gives a good understanding
of the build process.

• Removing the '@' from Makefiles rules. Many *.mak files define the rules with '@' in the
beginning, which means "do not print this command". By removing the '@', the build output
is much more verbose.

• Adding warnings to the Makefiles. This can give a good understanding on the variables and
their values, explaining why targets are built or not.

• Using pre-compiler output. It can be very useful to create an output file for the pre-
compilation, in order to see which header files were actually included, which MACROs were
defined, etc. To do so, add "-E" to the compilation flags in the Makefile:

```
JMK_CFLAGS=-E
```

This line will create the .o file with the pre-compiled output of the compilation instead of
the binary object. Remember to remove this line when you are done debugging and trying to
build the real object.

• Removing the remark sign ('#') from a debug rule in jmk/debug_root.mak. This will add
compilation output lines when running make.

```
# debug_subdirs=y
# debug_create_config=y
# debug_libs_a_to_so=y
# debug_ramdisk=y
```

• Forcing a single job compilation (using only one processor). This eliminates the mixing of
compilation output lines from different processors, making them easier to follow. Use the
following command:

```
make ENV_RG_PARALLEL_MAKE=n
```

Note: You can set the number of jobs by changing the numeric value above; the
default value is the number of CPUs plus 1.
19.6 Makefile Variables

**JMK_TARGET** An executable that runs on the target.

**JMK_A_TARGET** A library archive (.a) that is used on the target.

**JMK_SO_TARGET** A shared library (.so) that is used on the target.

**JMK_O_TARGET** An object file that is linked with a JMK_TARGET.

**JMK_LOCAL_TARGET** An executable that runs on the host.

**JMK_LOCAL_A_TARGET** A library archive (.a) that is used on the host.

**JMK_LOCAL_O_TARGET** An object file that is linked with a JMK_LOCAL_TARGET.

**JMK_LOCAL_CXX_TARGET** An executable that runs on the host, built from a C++ source.

**JMK_MOD_TARGET** A kernel module for the target.

**JMK_MOD_2_STAT** A Makefile instruction to link JMK_MOD_TARGET statically with the kernel.

**JMK_OTHER_TASKS** A general Makefile target, created in the 'make all' stage, after the subdirectories and LOCAL_TARGETs but before the TARGETs.

**JMK_OTHER_TARGET** A general Makefile target, created in the 'make all' stage, after the subdirectories but before the LOCAL_TARGETs.

**JMK_ARCHCONFIG_FIRST_TASKS** A general Makefile target, created in the 'make config' stage before the subdirectories (excluding JPKG_SRC).

**JMK_ARCHCONFIG_JPKG_FIRST_TASKS** A general Makefile target, created in the 'make config' stage before the subdirectories, and including JPKG_SRC.

**JMK_ARCHCONFIG_LAST_TASKS** A general Makefile target, created in the 'make config' stage after the subdirectories (excluding JPKG_SRC).

**JMK_ARCHCONFIG_JPKG_LAST_TASKS** A general Makefile target, created in the 'make config' stage after the subdirectories, and including JPKG_SRC.

**JMK_O_OBJS** The objects that are compiled and linked to create the TARGETs.

**JMK_O_OBJS_$(TARGET_NAME)** Same as JMK_O_OBJS, but for a specific TARGET_NAME (not for all the TARGETs in the Makefile).

**JMK_L_OBJS** The objects that are linked to create the TARGETs.

**JMK_L_OBJS_$(TARGET_NAME)** Same as JMK_L_OBJS, but for a specific TARGET_NAME (not for all the TARGETs in the Makefile).
**JMK_AUTOGEN_SRC**  An auto-generated source file. When building a JPKG distribution, this file is not compiled and exported. The file should be auto-generated in the JPKG tree.

**JMK_LINKS**  A source file that is a link to another file. When building a JPKG distribution this file is not compiled and exported. The link target should be exported.

**JMK_EXPORT_AS_SRC**  When in a binary JPKG distribution, export the source file instead of the binary.

**JMK_DONT_EXPORT**  When in a JPKG distribution, do not export this file.

**JMK_JPKG_TARGET_${JMKG_TARGET}**  Build this JMK_TARGET even though this is a JPKG distribution.

**JMK_JPKG_TARGET_BIN_${JMKG_TARGET}**  Build this JMK_TARGET even though this is a binary JPKG distribution.

**JMK_CFLAGS**  The compilation flags for the target.

**JMK_LDFLAGS**  The linkage flags for the target.

**JMK_LOCAL_CFLAGS**  The compilation flags for the host.

**JMK_LOCAL_LDFLAGS**  The linkage flags for the host.

**JMK_LIBS**  The list of libraries passed to the linker, for the target.

**JMK_LOCAL_LIBS**  The list of libraries passed to the linker, for the host.

**JMK_CFLAGS_${JMKG_TARGET}**  Same as JMK_CFLAGS and JMK_LOCAL_CFLAGS, but for a specific TARGET_NAME (not for all the TARGETs in the Makefile).

**JMK_LDFLAGS_${JMKG_TARGET}**  Same as JMK_LDFLAGS and JMK_LOCAL_LDFLAGS, but for a specific TARGET_NAME (not for all the TARGETs in the Makefile).

**JMK_LIBS_${JMKG_TARGET}**  Same as JMK_LIBS and JMK_LOCAL_LIBS, but for a specific TARGET_NAME (not for all the TARGETs in the Makefile).

**JMK_CFLAGS_REMOVE_${JMKG_TARGET}**  Remove compilation flags for a specific JMK_TARGET.

**JMK_LDFLAGS_REMOVE_${JMKG_TARGET}**  Remove linkage flags for a specific JMK_TARGET.

**JMK_SUBDIRS**  The list of subdirectories for the 'make all' stage.

**JMK_ARCHCONFIG_SUBDIRS**  The list of subdirectories for the 'make config' stage. If JMK_ARCHCONFIG_SUBDIRS is not defined, then JMK_ARCHCONFIG_SUBDIRS=JMK_SUBDIRS.
**JMK_LINK_DIRS**  In the 'make config' stage, copy this directory to the build directory. This is used for open-source packages that with Makefiles not converted to those of OpenRG.

**JMK_EXPORT_LIBS**  Create a link from `rg/pkg/lib/` to the specified JMK_EXPORT_LIBS.

**JMK_EXPORT_HEADERS**  Create a link from `rg/pkg/include/` to the specified JMK_EXPORT_HEADERS. These files are always exported to the distribution.

**JMK_EXPORT_HEADERS_DIR**  Use this directory as a subdirectory of `rg/pkg/include/` for the created link.

**JMK_INTERNAL_HEADERS**  Headers that are used by the package only. These will be exported to the distribution only in the source distribution (JPKG_SRC).

**JMK_CD_EXPORTED_FILES**  Export files to the distribution, both binary and source.

**JMK_JPKG_EXPORTED_DIR**  Export the directory as is to the distribution, both binary and source.

**JMK_JPKG_EXPORTED_DIR_SRC**  Export the directory as is, only to source distribution.

**JMK_RUN_UNITTEST**  The utility that runs when the 'make run_unintests' command is given.

**JMK_RUN_UNITTEST_DATA**  The list of files needed by the JMK_RUN_UNITTEST utility. The files are copied to the build directory.

**JMK_RAMDISK_FILES**  The list of files to be copied to the Ramdisk.

**JMK_RAMDISK_LIB_FILES**  The list of files to be copied to the Ramdisk under the /lib directory.

**JMK_RAMDISK_ETC_FILES**  The list of files to be copied to the Ramdisk under the /etc directory.

**JMK_RAMDISK_BIN_FILES**  The list of files to be copied to the Ramdisk under the /bin directory.

**JMK_RAMDISK_VAR_FILES**  The list of files to be copied to the Ramdisk under the /var directory.

**JMK_RAMDISK_MODULES_FILES**  The list of modules to be copied to the Ramdisk under the /lib/modules directory.

**JMK_RAMDISK_MODULES_PERMANENT_FILES**  The list of module object files that will be retained in the Ramdisk after the boot sequence. For more information, refer to Section 13.2.

**NOT_FOR_RAMDISK**  Used to override the generic copy of targets to the Ramdisk.
**JMK_RAMDISK_TASKS** Used for more complex actions that should be done at the ramdisk stage. Every task you define must have a rule that creates it.

**JMK_RAMDISK_DIRS** Used for directories that should be created in the ramdisk directory.

## 19.7 *.mak Files

The conventions for file naming are as follow:

- **xxx[_env]_root[_end].mak** – File is included once from the root of the flat logic. If the name is xxx_env_root.mak, the file is included before loading any Makefiles. Otherwise it is included after all Makefiles were loaded.

- **xxx[_env]_subdir.mak** – File is included in every subdir. If the name is xxx_env_subdir.mak, the file is included before loading the directory's Makefile. Otherwise it is included after the directory's Makefile.

**jmk/env_root.mak** Included at the beginning of every Makefile in the OpenRG tree, this file defines the variables of the OpenRG build system.

**jmk/rg_root.mak** Included at the end of every Makefile in the OpenRG tree, this file defines the rules of the OpenRG build system.

**jmk/general_dep_root_end.mak** Included from jmk/rg_root.mak, this file defines all the dependencies of the OpenRG build system.

**jmk/subdirs_root.mak** Included from jmk/rg_root.mak, this file contains the engine that includes the Makefile of all the sub-directories to the main Makefile.

**config_root.mak** Included from OpenRG's root (rg/Makefile.main), this file contains the rules for the 'make config' stage. config_root_first_phase.mak and config_root_second_phase.mak are included from config_root.mak.

**jmk/ramdisk_env_root.mak** Included from jmk/env_root.mak, this file defines some of OpenRG's build system variables, mainly for the creation of the Ramdisk.

**jmk/ramdisk_root_end.mak** Included from jmk/rg_root.mak, this file defines the rules for copying files to the Ramdisk.

**rmt.mak** Included from OpenRG's root (rg/Makefile.main), this file contains the rules to create the remote managent image (openrg.rmt).

**bootldr.mak** Included from the OpenRG's root (rg/Makefile.main), this file creates boot loader images for specific distributions.

**create_includes.mak** Included from OpenRG's root (rg/Makefile.main), this file defines the rules for creating the links of the kernel headers and libc headers.
The Build Mechanism

`jmk/old/docs.mak` Included from `jmk/old/rg.mak`, this file defines the rules for creating documents.

`envir.subdirs.mak` Included by all Makefiles in the sub-directories, this file is used to enforce compilation flags for all subdirectories of a specific directory. For example, let us look at `vendor/envir.subdirs.mak` that contains the line `JMK_WARN2ERR=n`. When executing:

```
rg.dev$ cd vendor/intel/ixp425/modules && make
```

The vendor/intel/ixp425/modules/Makefile includes vendor/envir.subdirs.mak. What it actually executed is:

```
-include \
  rg/envir.subdirs.mak \
  rg/vendor/envir.subdirs.mak \
  rg/vendor/intel/envir.subdirs.mak \
  rg/vendor/intel/ixp425/envir.subdirs.mak \
  rg/vendor/intel/ixp425/modules/envir.subdirs.mak
```

The hyphen (-) before ‘include’ instructs the Makefile to ignore it if the file does not exist.

`jmk/old/make_cd.mak` Included from OpenRG’s root (rg/Makefile.main), this file is used to create deliveries.

`jmk/target_binfmt_elf.mak` Included from `jmk/general_dep_root_end.mak`, this file defines the rules for creating elf TARGETs.

`jmk/target_binfmt_flat.mak` Included from `jmk/general_dep_root_end.mak`, this file defines the rules for creating flat TARGETs.

### 19.8 Using C++

You may choose to use C++, in which case you must first properly build the development environment and enable use of the C library. This section explains the steps you must take in order to use C++ with OpenRG.

#### 19.8.1 Setting the Development Environment

Before building and running a C++ application, you must first build uClibC++ as well as configuring OpenRG slightly differently:

1. Rebuild your tree and add add the `CONFIG_RG_CXX` to the make 'config' command, so the command should look like:

```
mak config DIST=... LIC=... CONFIG_RG_CXX=y && make
```

   This will compile the uClibC++ and load it as dynamic library into your board.

2. In order to link your program with the C++ library, add the following to your Makefile:

```
JMK_LDFLAGS+=-L$(JMKE_BUILDDIR)/pkg/uclibc++/src -luClibc++ -ldl
```

   You can use the following Makefile and simple "Hello World" program as an example:

   ```
   1. The Makefile:
   ```
ifndef JMKE_ENV_INCLUDED
    JMK_ROOT=../../
    include $(JMK_ROOT)/jmk/env_root.mak
endif

JMK_TARGET=hello
JMK_O_OBJS_hello+=main.o
JMK_LIBS+=$(OPENRG_LIBS)
JMK_RAMDISK_BIN_FILES+=$(JMK_TARGET)
JMK_LDFLAGS+=-L$(JMKE_BUILDDIR)/pkg/uclibc++/src -luClibc++ -ldl
$(call JMKE_INCLUDE_RULES)

2. The "Hello World" program (main.cpp):

```cpp
#include <iostream>

int main(void)
{
    std::cout << "Hello, World\n";
    return 0;
}
```

There are several test applications within the OpenRG tree located at `rg/pkg/uclibc++/tests` that provide some C++ examples that can be used as a reference.

## 19.8.2 Linking

When linking a C++ application with OpenRG's C libraries, every function called from the C++ application needs to have a new definition with the 'extern "C"' extension. For example, the `console_printf` C function can be used for C++ files by adding to your application the following line:

```cpp
extern "C" int console_printf(const char *format, ...);
```
20

Debugging

20.1 Debugging Methods

Based on various configuration options, the OpenRG system is comprised of several components of the following types:

- Linux kernel or VxWorks
- Kernel modules
- OpenRG user-mode main program
- Additional user-mode programs

There are several methods you could use to try and identify problems and crashes in the system. Once the system has crashed, you could analyze the crash dump and try to understand the cause of the crash. When trying to reproduce a problem, you might want to add debug print commands around the area you suspect to try and identify where the problem started. When doing so, you should be careful and take into account that these debug print commands might affect the system timing and move the problem to a different place. Another way to track down the problems in your program is to use a debugger. The debugger can show you where the program crashed and the current values of the variables.

You can set breakpoints at various places to monitor the program flow and data. Again, some of the system components rely on timing and are not suitable for debugger use. To debug the OpenRG system a remote debugging technique is used. The debugged program runs on the development board under a debugger agent and the host runs the debugger, which communicates with the debugger agent via a serial cable or an Ethernet cable. Some problems require the use of a hardware debugger. For example, trying to resolve an issue with the bootstrap of a board might be difficult without the use of a hardware debugger. Before delving
deeper into the specifics of OpenRG debugging utilities and commands, here are some general precepts that may serve you well, regardless of what you are attempting to debug:

- If the problematic portion of code with which you are dealing is limited in length, i.e. you merely added a few lines that caused a crash, or the output messages direct you to a specific area of code, comb this portion of code, the answer is likely to be found there. It may be a case of un-initialized variables, memory overrun, erroneous parameters, and so on. Even if you fail to locate the origin of the problem, the knowledge of the context can aid during the debugging session. For example, you can ascertain which variables should be inspected, and which lines should be executed step-by-step.

- Try and think of possible scenarios that may lead to the problem you are experiencing. Consider the debugging process of several applications that communicate with each other - a situation fraught with potential deadlocks, synchronization problems and shared resource conflicts. Such obstacles are better tackled when one is perched high, commanding a broad view yet intimately aware of the high-level sequence of events and entity relationships.

- If you experience a crash relating to registers and stack or memory dumps, it can be useful to disassemble the program into its distinctive components and search for the stack or memory function that may have caused the crash. In most cases the solution will be right under your nose, a wrong parameter that can easily be recreated from register values, freed variables that point to memory areas that containing garbage. Take in consideration that this kind of modus operandi requires moderate proficiency in low-level computer programming, and comprehensive knowledge of the schematics of your board.

- Be playful. Try and add variables that profile your code, count loop iterations, tally network packets, and so on. You can print the value of such variables when they reach a certain threshold or condition, or monitor their value during the debugging session. Being intelligently knowledgeable of the values of pertinent variables can lead to enlightening realizations on how to counter the setback.

20.2 GNU Debugger

GNU Debugger (GDB) is a debugger developed under the GNU project and provided with the OpenRG toolchain. It can be used in both console and window modes. This section concentrates on the console version, but all points are applicable to the window mode as well. It is recommended that you read Programming with GNU Software, specifically chapter 6 Debugging C and C++ Programs, which explains in detail how to use GDB.

The standard Linux distribution provides extremely helpful GDB documentation, which can be read by typing info gdb in your command prompt. The window modes are available with the DDD and GVD tools, which provide graphical user interface (GUI) to the GDB debugger. When activating these tools you need to direct them to use the GDB provided in the toolchain. You should use GDB to debug any new or problematic portion of code, be it the operating system or user applications in both OpenRG and vendor-specific modules. There are different terms that are used in context with GDB:
The GDB program runs on the host and allows running another program in a controlled environment. The GDB provides the user with commands to control the flow of the other program, using breakpoints and examining and modifying its data. The GDB commands can be grouped into the following categories:

**Control Commands** allow the user to run the program or continue its execution after a breakpoint. You could set or view the arguments for program execution with `set args` and `show args`. It is also possible to execute one source instruction at a time using the `next` command and to enter a function using the `step` command.

**Breakpoints** allow the user to break into the debugger at a certain line number or a function name. It is also possible to define a condition that should be met in order for the break to happen. You can view the list of breakpoints with `info breakpoints`. You can also delete, enable and disable a breakpoint providing its number. Once you stopped at a breakpoint you can see the call stack with `backtrace`.

**source and data** allows the user to view the source code with the `list` command and examine and modify both global and local program data with the `print` and `set` commands. You can also view the definition of data types and variables using the `whatis` and `ptype` commands.

The following table describes some of the useful GDB commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set args</td>
<td>set args</td>
<td>Set the arguments to pass to the debugged program</td>
</tr>
<tr>
<td>show args</td>
<td>show args</td>
<td>Show the arguments passed to the debugged program</td>
</tr>
<tr>
<td>run</td>
<td>r</td>
<td>Start executing the program</td>
</tr>
<tr>
<td>continue</td>
<td>c</td>
<td>Continue from the current breakpoint</td>
</tr>
<tr>
<td>next</td>
<td>n</td>
<td>Execute the current source line and stop before the next one</td>
</tr>
</tbody>
</table>
### Debugging

<table>
<thead>
<tr>
<th>step</th>
<th>s</th>
<th>Step into the function called on the current line and stop before executing it</th>
</tr>
</thead>
</table>

#### Breakpoint commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>b &lt;line&gt;</td>
<td>Break into GDB every time the control reaches the line number or enters the specified function</td>
<td></td>
</tr>
<tr>
<td>b &lt;where&gt; if &lt;condition&gt;</td>
<td>Break into GDB only if the condition is met</td>
<td></td>
</tr>
<tr>
<td>i b</td>
<td>Show all the current defined breakpoints, both active and inactive</td>
<td></td>
</tr>
<tr>
<td>d &lt;number&gt;</td>
<td>Remove the breakpoint &lt;number&gt;</td>
<td></td>
</tr>
<tr>
<td>ena &lt;number&gt;</td>
<td>Enable the breakpoint &lt;number&gt;</td>
<td></td>
</tr>
<tr>
<td>dis &lt;number&gt;</td>
<td>Disable the breakpoint &lt;number&gt;</td>
<td></td>
</tr>
<tr>
<td>bt</td>
<td>Display the current frames on the stack. Note that the further the frame is from the current one, the less likely the data there (which you can see using the following command) will be coherent</td>
<td></td>
</tr>
<tr>
<td>frame &lt;#&gt;</td>
<td>Jump to the designated frame number, &lt;#&gt;, in the stack. The further the frame is down the stack, the more likely you are to encounter garbage when viewing variables there</td>
<td></td>
</tr>
</tbody>
</table>

#### Source and Data commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
<td>Display several lines of code before and after the program counter's current position</td>
<td></td>
</tr>
<tr>
<td>p &lt;var&gt;</td>
<td>Print the value of a variable. Use C syntax, e.g. &quot;p <em>(socket</em>)s&quot; to de-reference a pointer. To dump part of an array use &quot;parray_var@5&quot; to see the first five elements of the array</td>
<td></td>
</tr>
<tr>
<td>p/x &lt;var&gt;</td>
<td>Print the value of the variable &lt;var&gt; in hexadecimal format. For example, use p/x $pc to print the current program counter on ARM</td>
<td></td>
</tr>
</tbody>
</table>
Register Information | i r
--- | ---
The compiler tends to optimize code, especially kernel code, which is compiled with -O2, so looking at the registers is very useful. When a function is called, in most architectures at most times, all passed parameters will be found in the registers. There are cases in which you print a variable and see a value in it that does not make sense—this could be since the variable is not used, due to optimizations, and a register is used instead.

**Miscellaneous commands**

<table>
<thead>
<tr>
<th>help</th>
<th>help</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display information regarding available commands</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Map Symbols</th>
<th>add-symbol-file \ <code>&lt;elfdebug version of executable&gt; &lt;address&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>This command maps executable symbols to a certain address. Make sure you know to where the program is loaded (where the text segment is), then use this command to map the GDB file created at compilation time to that address</td>
<td></td>
</tr>
</tbody>
</table>

**Table 20.1 Useful GDB Commands**

### 20.2.2 Remote Debugging

The GDB client runs on the host computer, and communicates with the GDB server running on the development board. The server on the development board is either *kgdb* for kernel debugging, or *gdbserver* for user-mode debugging. The communication channel between the development board and the host computer is either serial or Ethernet:

- Use a serial connection when using *kgdb*.
- Use a serial or Ethernet connection when using *gdbserver*.

![Figure 20.1 Remote Debugging Setup](image)
You can compile an OpenRG image with the "CONFIG_RG_GDBSERVER=y" flag. The `gdbserver` application will be added to OpenRG's file system (under the `bin/` directory), eliminating the need to fetch the application with protocols such as `tftp`. Instead, you will be able to run the application from a shell in OpenRG's CLI:

```bash
OpenRG> system shell
/ # gdbserver
```

## 20.3 Debugging a User-Mode Program

Debugging a user-mode program requires several tools installed on the host. Some are provided with the OpenRG release and others are available on the Linux distribution you are using.

- **gdb** is provided with the OpenRG release and is part of the toolchain under directory `/usr/local/openrg/<arch>/bin`, where `<arch>` is the relevant architecture of the target development board.

- **gdbserver** for the target development board is also provided with the OpenRG release and is under the tools directory.

- **console application** is used to connect via the serial cable to the target board console. It is available on your Linux distribution and from various other sources on the Internet. This document provides instructions for the **Minicom** console application.

- **TFTP client** is part of the OpenRG system and is used to transfer files from the host to the board and vice versa.

- **TFTP server** should be installed on the host and is available on your Linux distribution or through other Internet sources. The TFTP server repository holds the files transferred to and from the target.

- **window debuggers** are optional and available on Linux distribution and through other Internet sources. They should be installed on the host and directed to use the GDB provided in the toolchain.

### 20.3.1 Remote Connection Setup

In order to work with the serial port, it is necessary to configure the host and target to communicate at the same speed. You can accomplish this by running the following command on the host and target:

```bash
# stty -F /dev/ttyS1 speed 115200
```

The communication is established from the host side, however, the target should perform some actions beforehand.
20.3.2 Accessing the WBM via a PPPoS Connection

A Point-to-Point over Serial (PPPoS) connection, established between your board and a PC, enables you to access OpenRG's WBM via the serial port. This capability is useful, for example, in case you wish to deactivate OpenRG's LAN interface, and still maintain access to the WBM.

Note: The PPPoS connection does not allow you to access the Internet via the board's WAN device.

To establish a PPPoS connection to the board, perform the following:

1. Connect a PC to the board using a serial cable.

2. Compile a new OpenRG image with the following flag: CONFIG_RG_PPPOS_CLI=y. For example:

   ```
   make config DIST=MONTEJADE CONFIG_RG_PPPOS_CLI=y && make
   ```

3. Burn this image to the board:

   ```
   OpenRG> flash load -u tftp://192.168.1.10/openrg.img
   ```

4. After the new image is up and running, start the PPPoS service as follows:

   ```
   OpenRG> misc pppos_start /dev/ttyS1
   ```

5. Verify that the new device (ppp400) has been created on OpenRG:

   ```
   OpenRG> net ifconfig
   ```

6. Close the Minicom session.

7. Install a PPP client on your PC. For example, on a Debian Linux PC run:

   ```
   # apt-get install ppp
   ```

8. Verify that the `/etc/ppp/pap-secrets` file contains the `admin *` entry, and the `/etc/ppp/chap-secrets` file contains the `admin * admin *` entry.

9. Start the PPPoS connection:

   ```
   pppd /dev/ttyS0 debug nodetach noauth user admin local nocrtsets record ppp.log nobsdcomp nodeflate asyncmap 0xffffffff mru 1600
   ```

   After the connection is established, a new PPP device should appear on your PC:

   ```
   # ifconfig
   ppp0   Link encap:Point-to-Point Protocol
   inet addr:192.168.0.2  P-t-P:192.168.0.1  Mask:255.255.255.255
   ```

10. Open a browser and enter the PPPoS connection's IP: 192.168.0.1. OpenRG's WBM login screen will appear.
20.3.3 GDB Server Setup

The GDB Server is provided with the OpenRG release under `rg/pkg/gdb/gdb/gdbserver`. Run `make` to compile it. The first step before using `gdbserver` is to load it from the host to the target. The OpenRG system is equipped with a TFTP client that allows copying files from the host to the target and vice versa. The host should have a TFTP server installed and `gdbserver` should be copied to its repository. Then, on the target you can type the following commands in the Minicom to bring `gdbserver` to the target and be able to start using it:

```
# tftp -g 192.168.1.10 -r gdbserver
# chmod a+x gdbserver
```

In order to get to the shell prompt from OpenRG's CLI, use the CLI command `system shell`:

```
OpenRG> system shell
```

The `system shell` command will switch from OpenRG CLI to the BusyBox shell. To switch back to OpenRG CLI, use the `exit` command on the BusyBox shell. The TFTP client will get the `gdbserver` file from the host at IP address 192.168.1.10 and keep it in the `gdbserver` file on the target. The `gdbserver` is copied to the RAM disk. On some boards the RAM disk is small and a slightly different method should be used:

```
# mount -t ramfs ramfs /tmp
# cd /tmp
# tftp -g 192.168.1.10 -r gdbserver
# chmod a+x gdbserver
```

The first two commands create a RAM file system under the `/tmp` directory to where `gdbserver` is copied later. The RAM file system uses the target memory to store files and expands as needed. You should keep in mind that this memory file system reduces the memory available for the OpenRG system, hence you should limit the amount of data you store there when possible.

20.3.4 Start GDB Server

When working with `gdbserver`, you could communicate with GDB on the host either via the Ethernet connection or via the serial cable. There are two possible ways to start debugging, either attach the GDB server to a running process or start a new process under the GDB server. The examples below show the various ways to use `gdbserver` on the target:

1. The process is already running and according to the `ps` command its process id is 8. The GDB server should communicate with the GDB on the host via the Ethernet connection and use network port 1234:

   ````
   # gdbserver :1234 --attach 8
   ```

2. Starting a new process with `gdbserver` using the serial cable connected to the second serial port (/dev/ttyS1), is performed with the following command on the target:

   ```
   # gdbserver /dev/ttyS1 <process_name>
   ```

In order to restart OpenRG main program (openrg) you need to first stop the current running main program. You could do it from the OpenRG CLI with the `system exit_and_shell` command, which will leave you at the BusyBox shell prompt. There you can start `gdbserver`
on the current openrg program or bring a new openrg copy with TFTP and debug it. Another
option is to compile OpenRG with a special flag that does not start the OpenRG system but
rather the BusyBox shell. Then you could run the openrg program from the shell prompt. To do
that add the following definition to pkg/main/init.c file:

```c
#define OPENRG_DEBUG_EXEC_SH
```

For more information on BusyBox, refer to Chapter 22.

### 20.3.5 Start GDB on Host

Once the GDB server is up and running on the target development board, you can connect to
it from the GDB running on the host. You can start the GDB on the host with the following
commands:

```
$ cd <rg tree root>/pkg/main
$ /usr/local/openrg/<arch>/bin/<arch>_gdb openrg
(gdb)
```

Note that in the above case you may replace the openrg process with any other external
process you have.

The `<rg tree root>` is the directory in which you have installed the OpenRG source. The `<arch>`
is based on the architecture of your target board. After the GDB is up, you can connect to
the GDB server running on the target. If you use the Ethernet connection, use the following
command:

```
(gdb) target remote 192.168.1.1:1234
```

This command instructs the GDB to connect to the GDB server running on the target platform
using the Ethernet connection with the IP address 192.168.1.1 and port 1234. The port number
should be the same port number provided to the GDB server running on the target. After
establishing the connection, you can place breakpoints and start running the program:

```
(gdb) break my_function
Breakpoint 1 at <address>: file <my_file>.c, line <line>.
(gdb) continue
```

The break command instructs the GDB to stop the program execution on entry to the
`my_function` function. The GDB numbered this breakpoint number 1, which can be later
used to delete, disable or enable this breakpoint. In addition the GDB provides you with the
information of the address in the memory of this function, in what source file and at what line
number the function starts. When OpenRG is initialized, it runs several processes that initialize
different parts of the system. Each process that is started by OpenRG and exits when finished,
will send a signal to OpenRG to notify that it is finished. By default these signals stop the
GDB. A useful command is the `handle` command, which allows you to disable the break into
GDB on every SIGSTOP signal (caused by the exit of a child process):

```
(gdb) handle SIGSTOP nostop
```

### 20.3.6 Crash Dump

When the program you run crashes it will create a dump which shows the reason for the
crash, the registers at the time of crash and the stack status. When you run the program under
gdbserver, you can check where the crash was and have the debug information help you to
resolve the crash reason. If you have not used \texttt{gdbserver} to run the program, you can still use the information in the crash dump to try and understand what went wrong and where. To demonstrate the information obtained from a crash dump, the following (bad) code has been added to one of the WBM pages, which will definitely crash the main program when executed. In this example, the Watchdog tutorial code is used:

\begin{verbatim}
if (watchdog_conf.margin > 50)
{
    int *crash_ptr = 0;
    console_printf("Going to crash from User-space...");
    *crash_ptr = 1234;
}
\end{verbatim}

Once you activate the Watchdog page and set the margin to a value greater than 50, the code above will crash the OpenRG main program. Below is an example of a crash dump that was produced this way:

Going to crash...
openrg: unhandled page fault at pc=0x0009e4cc, lr=0x0009e4cc
   (bad address=0x00000000, code 7)
   pc : [<0009e4cc>]  lr : [<0009e4cc>]  Not tainted
   sp : bffffbc0  ip : 000004c8  fp : 00000000
   r10: 00000000  r9 : 00a3d8c  r8 : 00000000
   r7 : 001f6e38  r6 : 00166da0  r5 : 0020123c  r4 : 00000000
   r3 : 00000508  r2 : 00282268  r1 : 00282228  r0 : 00000012
Flags: Nczv  IRQs on  FIQs on  Mode USER_32  Segment user
Control: 39FF  Table: 0076C000  DAC: 00000015
User-mode stack:(0xbffffbc0 to 0xc0000000)
   fbc0: 002748f0 00000037 00000001 00166da0 bffffbc8 00000012 00155f48 00000000
   fbfc0: 00000000 00000000 0016d418 bffffbc4 0000003c 0016d428 00000000 00000000
   f0c0: 00000000 0016d430 bffffbc0 4000001a 00181ee0 00000000 00000000 00000000
   ...  
   fcc0: 00000000 00000000 00000000 00007635 62002f62 696e2f6f 70656e72 6700484f
   ffe0: 4d453d2f 00544552 4d3d6c69 6e757800 2f62696e 2f6f7065 6e726700 00000000
Kernel panic: Attempted to kill init!
\end{verbatim}

Right after the output of ‘Going to crash...’ generated by the \texttt{console_printf} function, the main program crashed due to 'unhandled page fault' that was caused by the code at hexadecimal address 0009e4cc, as pointed by the pc (program counter). In order to figure out what command caused the crash, you could use the \texttt{objdump} tool, which is part of the OpenRG toolchain, to disassemble the OpenRG main program with the source embedded. The command you would use is:

\begin{verbatim}
/usr/local/openrg/<arch>/bin/<arch>_objdump -xDSl openrg
>openrg.objdump
\end{verbatim}

To decode and analyze a crash dump in an easier way, use the Crash Analyze tool described in Section 20.9.

You should look for the crash address (9e4cc) in the contents of the \texttt{objdump} output file, in this case \texttt{openrg.objdump}. The following extract reflects what the file would look like around the crash address:

```
... 
9e4b8:       e59d3004        ldr     r3, [sp, #4] 
9e4bc:       e59f0084        ldr     r0, [pc, #132] ; 9e548 <s_scr_watchdog+0x178> 
9e4c0:       e3530032        cmp     r3, #50 ; 0x32 
9e4c4:       da000001        ble  9e4d0  <s_scr_watchdog+0x100> 
/.../rg/pkg/web_mng/cgi/watchdog_page.c:52 
    |    console_printf("Going to crash...
    | 9e4c8:     ebfc0e41        bl  f854 <_start-0x1810> 
/.../rg/pkg/web_mng/cgi/watchdog_page.c:53 
    *crash_ptr = 0;
```
You can see that the crash was caused by the `strb` command that tried to update the memory pointed to by `r4`. As you can see in the registers dump, the value of `r4` is 0, which is what caused the page fault.

## 20.4 Kernel Debugging

In some cases you would encounter a crash that happened in the kernel. Remote debugging of kernel code uses the same concept of user-mode debugging. You run the GDB on the host and a GDB agent on the target. One difference is that the GDB agent, `kgdb`, is embedded within the kernel. Another difference is that only serial communication is available between the GDB on the host and GDB agent on the target.

### 20.4.1 General Strategy: Swim Upstream

When a kernel crash occurs, it is often the result of a chain of events that is visible. The general strategy to solve bugs would be to start 'swimming' upstream from where a bug can be reliably detected:

- At the point where something goes wrong, discover what exactly is wrong, e.g., a bad pointer.
- Insert `printk()` statements to discover where at the earliest time something went wrong.
- If the problem was caused by something else that went wrong, keep following the chain of cause and effect backwards (using `printk`) until you find the place where the first thing that went wrong has not gone wrong yet.
- Keep closing in until you have narrowed it down to a single line of code.
- Remember that all you are interested in is the earliest point during execution at which memory becomes corrupted. A good way to focus on this is to have the process halt, or do an 'oops' as soon as it detects the corruption.

### 20.4.2 KGDB Server Setup

Working with `kgdb` requires that you activate it prior to the debugging session. This is done by using the "`CONFIG_RG_KGDB=y`" flag at compilation time.

> **Warning:** `kgdb` is not supported on all platforms, and therefore may not be available.

If you are developing for an ARM-based platform and use Minicom for serial communication, you can break into `kgdb` by pressing Ctrl+A, followed by F and D in the Minicom session.
Connect to the agent from the host. Run `gdb` and in its prompt invoke the 'target remote' command. If using a serial connection use the following command:

```bash
(gdb) target remote /dev/ttyS1
```

## 20.4.3 Running Two Serial Channels on a Single Port

Most board platforms feature a single serial port. This poses a difficulty when debugging the kernel, where two serial channels are required—one for the serial console and the other for the debug process. The following script enables you to use two serial channels on a single serial port.

Download and extract the `kgdb demux` script:

```bash
$ wget http://linux.junsun.net/porting-howto/src/kdmx-1.02.tar.gz
$ tar xzvf kdmx-1.02.tar.gz
$ cd kdmx-1.02
$ g kgdb_demux.pl
```

Comment out the 'use blib' line:

```bash
@@ -42,7 +42,7 @@
-    $Z0,c00d61b4,4#3a
-    hel$c#63lo   { "hello" -> console, "$c#63" -> debug }
+    # use blib;
+    use IO::Pty;
    use IO::Select;
```

Run the script:

```bash
$ sudo su
# kgdb_demux.pl
```

You can now use both console and debug ports separately. To exit, press Ctrl+C.

```bash
# minicom --pty=/dev/pts/6
```

Since you cannot send the BREAK char from this Minicom console, open another regular Minicom window (/dev/ttyS0), send the break, and close it immediately afterwards.

```bash
# minicom
<Press Ctrl+A,F,D>
<Exit Minicom>
```

Open `jgdb` to use the debug port:

```bash
$ cdr
# jgdb build/os/linux/vmlinux
(gdb) target remote /dev/pts/8
```

Optionally, you can use the following settings:

```bash
(gdb) set remotebaud 38400
(gdb) set debug remote 1
```

## 20.4.4 "Oops"

When the kernel falls, an "oops" is issued:
Unable to handle kernel paging request at virtual address 00000000,
epc == 8018083c, ra == 80180874
Oops: 0000
$0 : 00000000 80230000 0000002e 00000029
$4 : 0000003b 0000003b 81204000 1000fc01
$8 : 00006e00 ffff00ff 00000000 bf000928
$12: 00000600 81205c90 000000a0 0000000d
$16: 00000000 81fbb994 000fffff 817202c0
$20: 81fbb920 81401c00 00000000 00000000
$24: 81204000 81205ed6 7fffff98 80180874
epc : 8018083c
Status: 1000fc03
Cause : 00800008
Process dhclient (pid: 290, stackpage=81204000)
Stack: 802096f0 80209744 00000000 00000000 802b9400 817202c0 ffffff81
81205ea8 801791f0 ffffff7c 801755bc 817202c0 802b9400 817202c0
00000003 ..
Call Trace: [<802096f0>] [<80209744>] [<801791f0>] [<801755bc>]
[<801b4864>] [<801b4774>] [<80176348>] [<801729cc>]
[<00275a1c>] [<80173284>] [<801252b0>] [<801251a4>]
[<c0275a1c>] [<80173f70>] [<80110180>] [<8010c6c4>]
Code: 12110013 3c12000f 3652ffff <8e060000> 0246102b 10400005
0004 0242102b
Aiee, killing interrupt handler

This will show us where the 'oops' happened as well as show us the call stack and the state of
the registers. In the example above the 'oops' happened at the address in the program counter,
"epc".

### 20.4.5 System Map

The `linux/System.map` file will show you what function caused the 'oops' if it occurred in
statically-linked code. Look up the epc in this file. If you cannot find anything close, the 'oops'
may have occurred in a loaded module (refer to Section 20.4.10).

### 20.4.6 Object Dump

Use the object dump utility for your architecture, e.g. for the MIPS architecture:

```
$ /usr/local/openrg/mips/bin/mips-linux-objdump
   -Sdg rg.100/linux/net/core/core.o > core.src
```

This will give you disassembled code for a module. If it was compiled with debug info (-g)
then you will get C source along with the assembler. To simplify things you can do this on the
total kernel:

- Ensure that `linux/arch/mips/boot/Makefile` is modified to save the kernel image before
  symbols are stripped:

  ```
  cp $(PROD_KERNEL_IMAGE) $(PROD_KERNEL_IMAGE).symbols
  $(STRIP) $(PROD_KERNEL_IMAGE)
  ```

- If you want to see all source lines of code in the dump, add the following to `linux/Makefile`:

  ```
  CFLAGS += -g
  ```

- Once the kernel image is built, get the `objdump`:

  ```
  $ /usr/local/openrg/mips/bin/mips-linux-objdump -DSl vmlinux.symbols >vmdump
  ```
• Look up addresses from `oops/show_trace()` in `vmdump`. You will see exactly what machine instruction was executing as well as what registers it was using. Look at the 'oops' printout to examine the contents of the registers—this will give you an idea what data may have been corrupted.

20.4.7 KSYMS

KSYMS shows you where modules are loaded in memory:

```
801fac0c strtok
801facb4 memcmp
801fadc0 simple_strtoul
801fb614 sprintf
801fb640 disable_irq
801fb6ec enable_irq
801fb998 request_irq
```

On your development board use the following command:

```
~# cat /proc/ksyms > ksyms.out
```

You could send the file to the host with the TFTP client using the following command:

```
~# tftp put 192.168.1.10:ksyms.out ksyms.out
```

20.4.8 SHOW_TRACE

In `linux/arch/mips/kernel/traps.c` there is a function called `show_trace()` that uses `printk()` to print out a trace of the call stack. You need to give it the stack pointer as an argument—try the address of the first local variable in the routine. You can call this from any kernel-mode code to get a printout of the call stack. There are other interesting functions there like `show_stack()` and `show_code()`.

20.4.9 Logging Console Output

If you use Minicom to open a console on the box, first press Ctrl-A L—this will open a log file `minicom.cap` in the Minicom current working directory. Press Ctrl-A L to close it when whatever you are waiting for to happen happens. You now have a log of all console output.

20.4.10 Finding an Address in a Loaded Module

• Target an address to lookup from the 'oops' (e.g. pc).

• Look in `ksyms` to find the closest address that is not greater than the target address—this will give a routine name.

• Calculate the offset from the start of the routine: offset = target_address - routine_address.
• Find the module in which the routine is: go to where your tags file is and type gvim -t <routine_name>—the directory in which the source file is, is the name of the module and is also where the module is located. Note that you may have to go through a number of hits—use the :ts command in gvim to see all possibilities. Pay close attention to the architecture-dependent modules.

• Create a disassembled listing for the module using objdump.

• Find the routine in the objdump listing—it will have an address relative to the start of the module. Add the offset you calculated above to that address to get to the instruction of the target address; it may or may not be in the same routine.

• Now that you have an idea of where things are going wrong, you can start inserting printk() in strategic places.

• To try and figure out from where a function is called, use show_trace().

## 20.5 Debugging Kernel Modules

On boot, OpenRG prints all the insmod commands that are used when loading modules. For each module the addresses of where the module code and data are displayed. Later, you can use these addresses, when working with gdb, to locate the source code and variables of the module. For example, if you require to debug the Watchdog module, simply locate its insmod command:

```
insmod: add-symbol-file PATH/watchdog_mod.o 0xc5931060
    -s .data 0xc5931400 -s .bss 0xc5931458
```

After connecting gdb to OpenRG, invoke the following command:

```
add-symbol-file pkg/samples/tutorial/module/watchdog_mod.o 0xc5931060 -s .data 0xc5931400 -s .bss 0xc5931458
```

This will add all the symbols for these segments to gdb. Note that the first number is the text segment.

## 20.6 Finding an Address

In case you experience a crash dump, you can find the source of your problem by looking at the numbers on the stack, visible from the stack dump and register output. For each target, there is a difference in kernel/module addresses and user-mode addresses. For example, for on ARM-based boards, user-mode application are loaded from address 0x400, and kernel runs from 0xc0000000.

Thus, you can determine which mode caused the problem. Use insmod output to determine a module and a function for kernel-mode problems. For example, the Watchdog tutorial is used to create a crash when the margin is set to more than 40 seconds using the following code in pkg/samples/tutorial/module/watchdog.c:

```
static int do_ioctl(kos_chardev_t *context, unsigned int cmd, unsigned long data)
```
Once you set the margin to more than 40, the kernel module will crash. You could for example get the following crash dump on the console log:

```
Set new margin
Unable to handle kernel NULL pointer dereference at virtual address 00000000
pgd = c0260000
pc : (<c5931184>)[00000000] lr : [<c5931178>] Not tainted
sp : c0269d70 ip : 00000000 fp : 00000000
r10: 40170308 r9 : c0268000 r8 : bffffaf8
r7 : ffffffff r6 : c3575a60 r5 : c3653d80 r4 : bffffaf8
r3 : 00000000 r2 : 00000000 r1 : bffffaf8 r0 : 00000000
Flags: nzCv IRQs on FIQs on Mode SVC_32 Segment user
Control: 39FF Table: 00260000 DAC: 00000015
Process watchdog (pid: 54, stack limit = 0xc0268368)
Stack: (0xc0269d70 to 0xc026a000)
9d60: 00000000 c3ae97c0 00000000 00000000
9d80: c00baeb0 c0269d98 c01bffa8 00000000 c00bad2c c3ae97c0 00000000 00000000
9da0: 08000000 c399229c c0196298 c0196298 c0269d94 00000000 c0196298 c018c1c4
... 9fc0: 00000003 4004c002 0000003 00000001 bfffd00 00000000 40170308 00000000
9fe0: 40170558 bfffa0 40161918 401618e4 20000010 00000003 e58da56c e5999c00
Backtrace: not available
Code: ebf7ffbd e59d3000 e3530028 c3a02000 (c5c22000)
Breaking into KGDB
```

The crash address in `pc` is 0xc5931184 and according to the log messages of the `insmod` you can see that the module is the `watchdog_mod.o` that was loaded to address 0xc5931060. You can disassemble the module to find out the exact place. The `adjust-vma` option may be used to give the corrected mapping of the symbols according to the load address of the module. You could also calculate the offset by subtracting the module code start address from the crash program counter. In our example the offset would be:

```
crash offset = 0xc5931184 - 0xc5931060 = 0x124
```

The `objdump` command you would use is:

```
objdump -Dsxl watchdog_mod.o
```

And the area around the crash (offset 0x124) should look like this:

```
../../../rg/pkg/samples/tutorial/module/watchdog.c:51
if (margin > 40)
  200: e59df3000 1d , r3, [sp]  
  202: c3530028  cmp  r3, #40 ; 0x28
  202: 0x28
  202: e3a02000  movgt  r2, #0 ; 0x0
  202: c5c22000  strgtb  r2, [r2]  
  202: ea00005e  b  180  <do_ioctl+0x124>
  202: 128: R_ARM_PC24 .text
```
You can see from the registers in the crash that the value of \texttt{r2} is 0 and the instruction at offset 0x124 is trying to store the value to the address that created the 'kernel NULL pointer dereference'.

## 20.7 Configuring GDB

The GDB debugger is very powerful and has lots of features and configuration options. A few of its important features are listed here.

### 20.7.1 .gdbinit

Once started, \texttt{gdb} searches for a file named \texttt{.gdbinit} in the current directory, and if it is not found, it will look for this file in your home directory. You can write simple procedures there and invoke them from the GDB prompt. For example, if your \texttt{.gdbinit} file were to contain:

```c
define xs
  b openrg_start
  c
end
```

You could type this from the GDB prompt when it is running:

```
(gdb) xs
```

This will cause \texttt{gdb} to put a break-point at the \texttt{openrg_start()} function and continue execution.

### 20.7.2 Dynamic Link

In case your system is configured to use a dynamic link, you will have problems identifying where the dynamically linked libraries are mapped in memory. To overcome this you could use the \texttt{solib} options as listed below:

```
(gdb) set solib-absolute-prefix /dev/null
(gdb) set solib-search-path /.../rg/pkg/ulibc/lib:/.../rg/pkg/lib:
    /.../rg/pkg/freeswan/lib:/.../rg/pkg/openssl/crypto
```

## 20.8 Cross-GDB Compilation

Cross-GDB compilation is not a trivial issue, and you may find it necessary to debug often. The following are instructions for building a cross-debugging GDB for x86 and a GDB server for a MIPS target. The following procedure is performed on a x86 host PC.

1. Build the GDB.
2. Build the GDB server:
20.9 Crash Analyze Tool

20.9.1 Tool Description

The crash analyze script, `crash_analyze.pl`, can be used to determine the function call stack of OpenRG from the serial console output. The script can be found in the development tree under the `rg/pkg/tools` subdirectory. Using the crash analyze script:

1. Start capturing the serial output and boot the OpenRG board. The capturing can be done using a terminal program (e.g. Minicom) connected to the OpenRG serial console. It should contain all the text, which is printed by OpenRG to the console, starting from boot, and especially the `insmod trace` log messages.

2. Cause OpenRG to crash. If you have an application that crashes the board, run the following command in the shell, just before you run the application:

   ```
   > export LD_LIBRARY_PATH=TRACE_AND_RUN
   ```

   This will produce log messages, which will enable you to trace crashes inside shared objects that your application uses.

3. Enter the OpenRG tree root directory, from which the crashing image was built.

4. Run:

   ```
   ./pkg/tools/crash_analyze.pl < crash.log
   ```

   Or, if you ran an application that crashed, run:

   ```
   ./pkg/tools/crash_analyze.pl -p MyApplication < crash.log
   ```

   This will display the function call stack. `MyApplication` is the application that crashed.

   **Note:** In order for the function call stack to contain function names in addition to memory addresses, OpenRG and all the other modules and programs should be compiled with the compilation flags:
• -g—includes debug symbols
• -fno-omit-frame-pointer—enables reconstruction of the call stack.

Alternatively, you may run 'make config CONFIG_RG_DEV=y DIST=<your_dist>
<your_other_configs>' in the OpenRG source tree configuration stage, in order to set
these flags.

The **analyze_crash.pl** script can be run with the following options:

- **-h** – Help
- **-d** – Dump symbol table. Do not print the stack trace
- **-a** – Print all symbols, even the ones that are outside the objects
- **-m** – Ignore missing modules
- **-p prog** – Load and analyze dynamic libraries of the 'prog' program
- **-i** – Target instruction-set specific support (currently includes ARM only)

### 20.9.2  Crash Example

The following is an extract from a crash log as captured by Minicom:

```
Deliberately crashing in pkg/main/openrg_main.c
openrg: unhandled page fault at pc=0x4003ff40, lr=0x40040818 (bad address=0x00000001, code 243)
pc : [<4003ff40>]    lr : [<40040818>]    Not tainted
sp : bffffd18  ip : 4003ff10  fp : 00000002
r10: 40062c64  r9 : 00000200  r8 : 00000000
r7 : 4006475b  r6 : 001deae0  r5 : 4006438b  r4 : 00000200
r3 : 00000001  r2 : 7f800000  r1 : 00376038  r0 : 00000000
Flags: nZCv  IRQs on  FIQs on  Mode USER_32  Segment user
Control: 39FF  Table: 00CD4000  DAC: 00000015
User-mode stack:(0xbffffd18 to 0xc0000000)
```

---

© 1998-2012 Jungo Ltd.
Kernel panic: Attempted to kill init!

The following command produced the result.txt output file:

```
rg$ ./pkg/tools/crash_analyze.pl < crash.txt > result.txt
```

Examine the result.txt output file:

```
Loading vmlinux .text=0x0 (build/debug/vmlinux)
Loading openrg .text=0x0 (build/debug/openrg)
Loading libjutil.so .text=0x4000a000 (build/debug/libjutil.so)
Loading libssl.so .text=0x400a6000 (build/debug/libssl.so)
Loading libc.so.0 .text=0x401a6000 (build/pkg/ulibc/lib/libc.so.0)
```

```
nm: build/pkg/ulibc/lib/libc.so.0: no symbols
```

```
Loading ld-uClibc.so.0 .text=0x40000000 (build/pkg/ulibc/lib/ld-uClibc.so.0)
```

```
nm: build/pkg/ulibc/lib/ld-uClibc.so.0: no symbols
```

Function stack which led to the crash:

```
pc: rg_error_reboot               (0x4003ff40: libjutil.so + 0x15f40)
lr:     _rg_vperror                   (0x40040818: libjutil.so + 0x16818)
        _rg_vperror                   (0x40040818: libjutil.so + 0x16818)
        main                          (0x1df78: openrg + 0x1df78)
        rg_error                      (0x40040874: libjutil.so + 0x16874)
        main                          (0x1dfc0: openrg + 0x1dfc0)
        main                          (0x1df78: openrg + 0x1df78)
```

The uppermost function is always the one that crashed the board. In this case it is `rg_error_reboot()`. The offset within the module is displayed (libjutil.so + 0x15f40). It can be debugged by disassembling `libjutil.so`.

## 20.10 User Mode Memory Debugging

Debugging the memory requires a memory leaks detection tool, which will help you locate the piece of faulty code causing retention of the memory blocks. Use Valgrind for this purpose. Valgrind is a set of tools for memory leaks detection and profiling. You should use Valgrind for general OpenRG memory diagnostics and also after creating a new WBM module, in order to test it for causing memory leaks. Valgrind compiles in `rg/pkg/valgrind`. To use the tool, you need to compile it with OpenRG’s UML by performing the following:

1. Open a shell prompt and change to the relevant tree root.

2. Run the following command:

   ```
   make config DIST=UML_VALGRIND
   ```

After the tree configuration process is complete, enter the `make` command to compile the UML. When the UML compilation is finished, enter the following commands:

```
jrguml burn
jrguml start -s
```

After OpenRG's UML is up and running, explore the Valgrind commands as follows:

```
OpenRG> help valgrind
mark_set   Set a new valgrind mark
mark_show  Show current valgrind mark
exit       Exit sub menu
```
Debugging

To check OpenRG's UML for memory leaks, perform the following:

1. In the shell prompt where you started OpenRG's UML, change to Valgrind and enter the leak_check command as follows:

```bash
OpenRG> valgrind
valgrind> leak_check
```

The memory leaks report is printed as in the following example:

```
==7== searching for pointers to 34 not-freed blocks.
==7== checked 955,600 bytes.
==7== 11 bytes in 1 blocks are still reachable in loss record 1 of 6
==7== at 0x40166CE: _vgrZU_libcZdsoZa_malloc (vg_replace_malloc.c:149)
==7== by 0x429BCEB: strdup (in /mnt/cramfs/lib/libc.so.6)
==7== by 0x826411E: cmd_substitute (cmd.c:218)
==7== by 0x826443E: do_command (cmd.c:352)
==7== by 0x82647FB: cmd_exec (cmd.c:453)
==7== by 0x8177937: cli_readline_do (cli.c:819)
==7== by 0x8177657: cli_readline (cli.c:743)
==7== by 0x8177B7D: cli_read_cb (cli.c:903)
==7== by 0x40500FD: estream_call_read (estream.c:557)
==7== by 0x404EC87: estream_read_e fd (estream.c:1002)
==7== by 0x404EC87: estream_call_read (estream.c:557)
==7== by 0x404EC87: estream_read_e fd (estream.c:1002)
==7== by 0x4040A73: event_loop (event.c:855)
==7== by 0x8062096: main (openrg_main.c:21)
==7== LEAK SUMMARY:
==7== definitely lost: 0 bytes in 0 blocks.
==7== possibly lost: 0 bytes in 0 blocks.
==7== still reachable: 737,518 bytes in 35,195 blocks.
==7== suppressed: 0 bytes in 0 blocks.
```

The number beginning each line (in this example it is 7) is the PID of the process. The third line, containing the phrase 'still reachable', means that while the report was generated, the memory block was not yet freed, but still the program can reach the block and free it. Lines #18 display the stack status at the time of the memory block allocation. If the memory block cannot be freed, it can be seen in the 'Leak Summary' section of the report. Line 20 will display the amount of bytes that were lost in a certain amount of blocks during the report generation. To start checking for memory leaks from a certain point in time, enter the following command:

```bash
valgrind> mark_set
```

The 'mark_set' command creates an index for the point in time, at which the command was run.

```
Note: By default, the time at which the UML was started is marked as 0.
```

To show the index of the current mark, enter the following command:

```bash
valgrind> mark_show
```

If you run the leak_check command after setting the mark, a report of the allocated memory will be printed from the last time mark till now. For example, assuming that the current mark is 6, and you want to check for the memory leaks from mark 2 till 5, enter the following command:

```bash
valgrind> leak_check 2 5
```

The report will show if there were memory leaks between marks 2 and 5.
2. If memory leaks occurred, start checking the code of each of the files in a stack. Valgrind facilitates your search. As it is seen from the example report, each file name is followed by a number of the code line from which the next function is called.

## 20.11 Kernel Mode Memory Debugging

You can compile the development tree with the KMALLOC flag, which assists with finding memory leaks in the kernel code. This feature is part of the kernel code (2.4 and 2.6). If you suspect that the kernel allocates memory without freeing it, you can use this debug option in order to find the line that allocated the unfreed memory.

### 20.11.1 Setup

1. When performing archconfig on the tree, add CONFIG_DEBUG_KMALLOC=y.

2. Burn the new image.

3. Access OpenRG’s shell.

4. `# cat /proc/slabdebug`

5. Perform the steps to reduce the memory.

6. `# cat /proc/slabdebug`

You can compare the two slabdebug reports to see if some slabs were not freed. This method will also alert when trying to free a wrong pointer (one that was not allocated), or in case of trying to free the same pointer twice.

### 20.11.2 The CONFIG_RG_DEBUG_KMALLOC Method

By using CONFIG_DEBUG_KMALLOC you can detect memory leaks but not memory overruns. In order to detect memory overruns, use CONFIG_RG_DEBUG_KMALLOC (currently only in kernel version 2.4). This debug method is activated with CONFIG_RG_DEBUG_KMALLOC. For each allocation, a header is added before the object, and several bytes are added after the object. The \texttt{kfree} function can detect the following:

- kfree without kmalloc (if header is incorrect)
- A second kfree to an object (was\_released flag)
- Buffer overrun (if the additional bytes are incorrect)
- Attempting to free a NULL object
There are also some `procfs` files for retrieving additional data:

```
cat /proc/kmalloc_debug_label   - get the current label (0-9)
echo label > /proc/kmalloc_debug_label  - set the current label (0-9)
cat /proc/kmalloc_report   - Displays a report of the allocations
    with the current label.
echo label > /proc/kmalloc_report - Displays a report of the allocations
    with 'label' label.
    - use * to see all labels

echo test > /proc/kmalloc_debug_test - Executes a memory test:
echo 1 > kmalloc_debug_test     - Check buffer overrun
echo 2 > kmalloc_debug_test     - Check freeing an invalid pointer
echo 3 > kmalloc_debug_test     - Check freeing an already free object
```

This debug method was successfully tested on a JIWIS board. It was also tested using UML 2.4 on a computer running kernel 2.6.

```
<i>The CONFIG_RG_DEBUG_KMALLOC and CONFIG_DEBUG_KMALLOC methods cannot be used together.</i>
```

### 20.11.3 Finding a Memory Leak

In order to find a memory leak with CONFIG_RG_DEBUG_KMALLOC, you must "capture" the state before and after the leak. In order to do so, use labels before and after the sequence you suspect is causing the leak.

1. Access OpenRG's shell.
2. `# echo 1 > /proc/kmalloc_debug_label`
3. Perform the steps to reduce the memory.
4. `# echo 2 > /proc/kmalloc_debug_label`
5. `# echo 1 > /proc/kmalloc_report`

The first two echoes are the labels for before and after the memory leak. The third echo tells the system that you want a report from label "1" up to the next label. You can make more than just two labels and then produce a report between each two successive labels. This debug method alerts if trying to free an incorrect pointer or the same pointer twice. It will also alert after freeing a pointer that was overrun.
20.12 Debugging Tips and Tricks

20.12.1 Running Debugging Applications with NFS

When debugging OpenRG, you can use a variety of debugging applications. Instead of fetching these applications from a remote host using protocols such as `tftp`, you can use the Network File System (NFS) protocol to mount a remote directory on the board and run an application or process from that directory, directly on OpenRG. Using this method, you can also run the `openrg` process itself from the mounted directory, saving you the need to load the image to the board. This is useful when using many images, which is common when debugging.

When compiling an OpenRG image, use the "CONFIG_NFS_FS=y" flag to add the NFS protocol to OpenRG. For example:

```
make config DIST=<your distribution> CONFIG_NFS_FS=y && make
```

After the compilation completes successfully, burn the image to the board. In OpenRG's CLI, execute:

```
OpenRG> shell
/ # mkdir /mnt/nfs
```

To mount the NFS directory you have created, use the following command:

```
/ # mount -t nfs -o nolock <host IP>:<path to directory on host> /mnt/nfs
```

You can now run the desired application from a shell in OpenRG's CLI:

```
/ # <application name>
```

20.12.2 Debugging the WBM with Firebug

The Mozilla Firefox™ browser features the Firebug extension. This add-on allows debugging, editing, and modifying a website's CSS, HTML, DOM, or JavaScript, and provides other web development tools. This utility is highly recommended when debugging OpenRG's WBM and Jungo.net pages. Refer to Firebug's official website for the complete guide.

You can install Firebug from within the Firefox browser—click 'Tools' and then 'Extensions'. If the new window does not contain Firebug, click 'Get More Extensions'. Search for the application, download and install.

An additional helpful debugging tool available from Mozilla is the iMacro add-on, which enables you to record the steps taken in the browser and then replay them.

20.13 Using the Crash Logger

OpenRG's serial console output is issued from the moment the board is switched on. This log is constantly written to the volatile RAM, which is deleted in case of a power cycle. In case of a crash, the crash dump code (used to analyze the crash) is also written to the volatile RAM.
To avoid losing this information after rebooting, OpenRG automatically saves the console log from the RAM to the flash, enabling you to review it when the board restarts. In addition, you can actively save the console output to the flash at anytime, or send it via email to an address defined in rg_conf, using CLI commands.

The OpenRG image must be compiled with the following flags to include this feature:

- CONFIG_RG_KLOG
- CONFIG_RG_KLOG_RAM_BE
- CONFIG_RG_KLOG_EMAIL
- CONFIG_RG_KLOG_RAMSIZE

**Warning:** This feature is only available on specific platforms. Loading an image containing the feature's flags to an incompatible platform may render the board inoperable.

When the console output is saved, the **crash** category of CLI commands becomes usable, enabling you to save and view serial logs and crashes. This category includes the following commands:

- **status**  Indicate if a new crash log is available.
- **timestamps**  Print a list of previous serial and crash logs timestamps (up to four logs are saved).
- **log <log num>**  View the log of a previous crash. Indicate the number of the desired log (0-3) – 0 - latest log, 3 - oldest log.
- **save_buffer_now**  Save the current serial log to the flash.
- **send_mail_now**  Send the current serial log by email, to the address defined in rg_conf. Refer to the explanation below to set up this address.
- **print_current_buffer**  Show the current serial log on the screen.
- **erase_buffer**  Erase the current serial log buffer from the RAM.

To set up the email address in rg_conf, use the following CLI commands:

```bash
conf set /klog/email/enabled 1
conf set /klog/email/smtp/server <email server>
conf set /klog/email/smtp/port 25
conf set /klog/email/smtp/to <email address>
conf set /klog/email/smtp/from <email address>
```

If you are using authentication:

```bash
conf set /klog/email/smtp/auth 1
conf set /klog/email/smtp/username <username>
conf set /klog/email/smtp/password <password>
```

**Note:** Your email server must be properly configured to accept these parameters. No notifications of errors or undeliverable messages are provided.
Part IV  Feature Knowledge Base
# Table of Contents

21. Version Management ................................................................. 251
22. BusyBox ................................................................................... 255
23. KOS/KNET ............................................................................... 257
24. Remote Upgrade ........................................................................ 263
25. Network Bridging ...................................................................... 267
26. Extending the SNMP Agent ....................................................... 282
27. Command Line Interface ........................................................... 300
28. Web-based Management ............................................................ 322
29. OpenRG Over UML ................................................................. 343
30. Firewall .................................................................................... 348
31. Quality of Service in a Nutshell .................................................. 385
32. Dynamic DNS ............................................................................ 391
33. Virtual Private Network (VPN) ................................................. 397
34. Adding a Proxy ........................................................................... 400
35. Voice over IP (VoIP) ................................................................. 402
36. Secure Socket Layer (SSL) VPN ............................................... 419
37. TR-069 and Extensions ............................................................... 423
38. TR-064 .................................................................................... 463
39. Network Acceleration ............................................................... 483
40. Asynchronous JavaScript and XML (AJAX) ............................... 490
41. Testing OpenRG's Networking Performance ............................... 497
42. Supporting Separate Networks with Routing Groups ................. 521
Version Management

OpenRG supports two version counters concurrently—an internal version, set by Jungo, and an external version, for use by the customer. The version management module is responsible for controlling the version number and setting the configuration file (rg_conf) values accordingly whenever OpenRG's image is updated or changed, resulting in full backwards compatibility between versions.

21.1 Backward Compatibility

Backward compatibility is required when upgrading OpenRG and wishing to keep its current configuration. The new (or changed) version's rg_conf entries are added by designated compatibility code. This code is for upgrading only. If a downgrade is performed, the only possible action is restoring the factory defaults.

The version management module runs whenever OpenRG reboots, comparing the image's (possibly new) internal and external versions to the ones stored in rg_conf. The possible scenarios are:

- **New version number < stored version number** – The new version is older than the stored version. In this case, factory defaults are restored, to avoid any errors.

- **New version number == stored version number** – The new version is the same as the stored version. In this case, no action is taken.

- **New version number > stored version number** – The new version is newer than the stored version. In this case compatibility code is run. This code resides in `rg/pkg/main/mt_rg_conf_compat.c`. 
Each version type is handled by a different function. OpenRG's internal version type is set of integers separated by dots, for example "4.5.6". The external version is user defined, and its default type is an integer.

Note: `rg_version_compare()` can also be used for external version comparison if its type is "x.y.z".

- **Internal version comparison** is performed by the following code:
  ```c
  if (rg_version_compare(saved_version, "4.5.0") < 0)
  ```

  The comparison function, which resides in `pkg/util/rg_version.c`, is:
  ```c
  int rg_version_compare(char *v1, char *v2);
  ```

  Its parameters are:
  - **v1** The current version.
  - **v2** The new version.

  Its possible return values are:
  - **-1** If v1 < v2.
  - **0** If v1 == v2.
  - **1** If v1 > v2.

- **External version comparison** is performed by the following code:
  ```c
  if (mt_rmt_upd_version_cmp(saved_ext_version, "A") < 0)
  ```

  The comparison function, which resides in `pkg/main/mt_rmt_upd.c`, is:
  ```c
  int mt_rmt_upd_version_cmp(char *v1, char *v2);
  ```

  Its parameters and possible return values are identical to those of the internal version comparison (as described above).

  The function's default behavior is:
  ```c
  return atoi(v1) - atoi(v2);
  ```

  You may want to add your own callback function in order to handle your method of version comparison. To do so, register your external version comparison function in `pkg/main/mt_rmt_upd.c`:
  ```c
  typedef int (*version_cmp_cb_t)(char *v1, char *v2);
  void mt_rmt_upd_register_version_cmp_func(
      version_cmp_cb_t func);
  ```

  Note: When an external version comparison function is registered, it is used not only for compatibility purposes, but also for checking whether a remote update of the software is required.
21.2 An Example

The following example demonstrates a version change on OpenRG and what can be achieved through this process—in this example you will add Watchdog configuration parameters as part of an external version update. The required steps are:

- Declare a new external version:

  **mgt/ext_version.h**
  ```c
  void ext_version_init(void);
  ```

  **pkg/main/mt_init.c**
  ```c
  void mt_init(void)
  {
    ...
    ext_version_init();
  }
  ```

  **pkg/include/external_version.h**
  ```
  #define EXTERNAL_VERSION "B"
  ```

- Register your own external version comparison function:

  **mgt/ext_version.c**
  ```c
  #include <main/mt_rmt_upd_common.h>
  int ext_version_compare(char *v1, char *v2)
  {
    if (!v1 && !v2)
      return 0;
    if (!v2)
      return 1;
    if (!v1)
      return -1;
    return strcmp(v1, v2);
  }
  void ext_version_init(void)
  {
    mt_rmt_upd_register_version_cmp_func(
      ext_version_compare);
  }
  ```

- Call compatibility code for the older version:

  **pkg/main/mt_rg_conf_compat.c**
  ```c
  void mt_rg_conf_compat(void)
  {
    ...
    if (mt_rmt_upd_version_cmp(saved_ext_version, "A") < 0)
      watchdog_conf_values_add();
    ...
  }
  ```

- Add Watchdog rg_conf parameters:

  **pkg/main/mt_rg_conf_compat.c**
  ```c
  #include <watchdog/mgt/watchdog_mgt.h>
  ...
  static void watchdog_conf_values_add(void)
  ```
Note: As depicted in this example, the comparison function registration must be called before the compatibility code is run.
BusyBox

BusyBox is a software application that provides many standard Unix tools. OpenRG utilizes BusyBox for its shell. To enter the shell, type the following at the CLI prompt:

```
OpenRG> system shell
BusyBox v1.01 (2005.09.07-07:38+0000) Built-in shell (lash)
Enter 'help' for a list of built-in commands.
```

To switch back to OpenRG’s CLI, use the `exit` command in the BusyBox shell.

22.1 Adding Applets

Some BusyBox commands are not included in OpenRG’s shell by default. In order to add an existing Busybox command (applet) to OpenRG’s shell, you must update the `pkg/busybox/.config` and `pkg/busybox/include/config.h` files and then recompile the OpenRG image.

For example, the command `vconfig`, used for creating and removing virtual Ethernet devices, is not included in OpenRG’s shell by default. To add this command, perform the following:

1. In `pkg/busybox/.config`, locate the line
   ```
   # CONFIG_VCONFIG is not set
   
   and change it to:
   ```
   ```
   CONFIG_VCONFIG=y
   ```

2. In `pkg/busybox/include/config.h`, locate the line
   ```
   #undef CONFIG_VCONFIG
   
   and change it to:
   ```
   ```
   #define CONFIG_VCONFIG 1
   ```
3. Compile and burn the image:

$ make
$ cp build/openrg.img /tftpboot/
OpenRG> flash load -u tftp://192.168.1.10/openrg.img
OpenRG> system reboot

After reboot, the applet is available in OpenRG's shell:

OpenRG> system shell
BusyBox v1.01 (2005.09.07-07:38+0000) Built-in shell (lash)
Enter 'help' for a list of built-in commands.

/ # vconfig
Usage: vconfig COMMAND [OPTIONS] ...

vconfig lets you create and remove virtual ethernet devices.

Options:
  add  [interface-name] [vlan_id]
  rem  [vlan-name]
  set_flag  [interface-name] [flag-num]  [0 | 1]
  set_egress_map  [vlan-name]  [skb_priority]  [vlan_qos]
  set_ingress_map  [vlan-name]  [skb_priority]  [vlan_qos]
  set_name_type  [name-type]

/ #
23

KOS/KNET

The KOS/KNET layer is an OS abstraction layer, allowing the kernel code to be reused with multiple operating systems. KOS/KNET exports OS-independent functions for use by OpenRG components. The implementation of those functions in KOS/KNET depends on the underlying operating system. In many cases, a KOS/KNET function is mapped directly to an existing Linux/VxWorks kernel function. KOS/KNET functions resides in \texttt{rg/pkg}.

23.1 KOS

KOS implements basic operating system functions. The functions are used in the following areas:

- Memory allocations
- Timers
- Interrupt control
- Locks, mutex
- Work scheduling
- Queues
- Char device implementation
23.2 KNET

KNET implements networking-related functions. KNET functions are used in the following areas:

- Packet management
- Networking constants
- Protocols helper structures
- Networking devices management
- Network hooks
- Routing
- Support for OpenRG components

23.2.1 Knet Hooks

The Kernel-mode Network Hooks (knet hooks) are building blocks of a generic, cross-platform packet processing mechanism. This mechanism enables you to register functions for processing network packets, either after the packets are received from a network device driver, or before they are sent to it. Packet processing functions may serve various purposes, such as:

- Network traffic filtering
- Packet statistics calculations
- Packet integrity checks
- Packet defragmentation
- Processing of special applications (such as DHCP, RTP)
- Network bridging

The Rx knet hooks are called after a network device driver prepares the sk_buff data structure holding the packet information—a step before the upper network layers (for example, the IP Stack) process this packet.

The Tx knet hooks are called when the network stack delivers the packet to a network device driver—a step before a packet is queued for transmission on a network device.

On Linux, the knet hook calls can be found in rg/os/linux/net/core/dev.c. The Rx knet hooks are called by the netif_receive_skb() function, just before it calls net_rx_lower(). The Tx knet hooks are called by the dev_queue_xmit() function.
Knet hooks are implemented on each device as a prioritized ordered list, in which they are organized according to the hook types. When the generic knet packet handler invokes all registered hooks, the hooks are called one after another in the pre-defined order. This is important, since a hook may act differently in case its position has been changed. Every hook may change the packet content, or even drop the packet before it is passed to the next hook. The hook’s decision whether to handle the packet by itself or to pass it forwards is determined by its return value.

If all registered hooks have been called, but none of them has handled the packet, processing of the packet continues according to the operating system's normal flow—the incoming packets are transferred to the network stack, and the outgoing ones move on to the device driver.

### 23.2.1.1 The Knet Hook API

The following is the knet hook (packet handler routine) API.

```c
typedef knet_hook_ret_type_t (*knet_hook_handler_t)(knet_buf_info_t *binfo);
```

**knet_hook_ret_type_t** The returning hook's code, which determines whether the hook has handled the packet (no further processing of the packet), or the packet processing continues. The type is defined in `pkg/kos/knet.h`:

```c
/* Return values of knet Rx/Tx hook handlers. */
* Important: Be very careful with the return value of a handler because it
* directly affects the flow of packets in the network stack. */
typedef enum {
    KNET_PKT_HANDLED = 0, /* Packet handled/dropped - no further handling. */
    KNET_PKT_NOT_HANDLED = 1, /* Packet not handled - all further handling
* allowed. */
} knet_hook_ret_type_t;
```

**Important**: If a hook decides to handle a packet, it should free the network buffer, using `knet_buf_free(*binfo->kb)`.

**binfo** A handler function receives a single network buffer info parameter (`binfo`), which encapsulates a network buffer (packet), supplying the common supplementary fields. The `knet_buf_info_t` type is defined in `pkg/kos/knet.h`. Consider a short piece of this structure:

```c
typedef struct {
    knet_link_proto_t link_proto;
    u8 *link_header; /* Pointer to knet_eth_hdr_t / ppp stuff, etc */
    knet_payload_proto_t payload_proto; /* KNET_ETH_TYPE_IP / ..._ARP / etc. */
    u16 link_header_len; /* length of link layer */
    u8 *payload; /* Pointer to data - i.e. IP header. */
    u8 *payload; /* OpenRG extension of net_device structure */
    knet_netdev_t *dev; /* OpenRG extension of net_device structure */
    u8 *payload; /* knet_buf->knet_data also points to the payload. */
} knet_buf_info_t;
```

**devx** Each device has an extension—a structure holding additional data that OpenRG components need to keep on a device. Each `devx` points to its `net_device` structure. Each kernel's `net_device` points to its `devx` structure. Consider a short piece of the `devx` structure:

```c
struct knet_if_ext {
    struct knet_stats_t stats;
    struct knet_if_ext *next;
```
Pay close attention to the pointer from net_device to devx. Usually, this pointer is implemented by a field added to struct net_device in the Linux kernel. But in case you have only the device driver's binaries, you cannot add fields to the net_device structure, because a device driver will know nothing about them. Instead, use a list of devx structures (knet_if_exts located in pkg/kos/knet_if_exts.c). When handling each packet, the correct devx is chosen from the list according to the net_device structure.

### 23.2.1.2 Hook Registration

The following is the knet hook registration function:

```c
void knet_buf_hook_devs_register(knet_hook_type_t type, knet_hook_dir_t dir,  
knet_hook_handler_t handler, knet_netdev_t **devs, int devs_count);
```

**type** A hook type that should be registered. The type determines a hook's priority. The following are examples of existing types defined in pkg/kos/knet_hooks.h:

```c
typedef enum {
    /* 0 is used by Dummy head */
    KNET_HOOK_FASTPATH, /**< Fast path */
    KNET_HOOK_FIREWALL, /**< Firewall */
    KNET_HOOK_BRIDGE, /**< Bridge */
    KNET_HOOK_DHCP, /**< DHCP override */
    KNET_HOOK_PPPOE, /**< PPPoE driver */
    KNET_HOOK_MAC_CACHE, /**< IP to MAC cache */
    KNET_HOOK_RTP_RECV, /**< RTP received packets */
    KNET_HOOK_QOS_INGRESS, /**< QoS ingress */
} knet_hook_type_t;
```

The type prioritization of the Rx/Tx hooks is determined by the arrays `knet_rx_hook_order[]` and `knet_tx_hook_order[]` respectively. A full array is defined in pkg/kos/knet_hooks.c:

```c
static knet_hook_type_t knet_rx_hook_order[] = {
    KNET_HOOK_FASTPATH,
    KNET_HOOK_FIREWALL,
    KNET_HOOK_BRIDGE,
    KNET_HOOK_DHCP,
    KNET_HOOK_PPPOE,
};
```

For example, assume there are only 3 registered Rx hooks, of type KNET_HOOK_FIREWALL, KNET_HOOK_BRIDGE and KNET_HOOK_PPPOE. The order of hook invocation will be: KNET_HOOK_FIREWALL, then KNET_HOOK_BRIDGE, and finally KNET_HOOK_PPPOE.
Note: Only one handler can be associated with a given type at any given time. This means that it is impossible to register several different handler functions of the same type simultaneously.

**dir** Represents the hook transmission direction, and may be one of the following:

```c
typedef enum {
    KNET_TX_HOOK = 0, /**< Transmit. */
    KNET_RX_HOOK = 1, /**< Receive. */
} knet_hook_dir_t;
```

**handler** The hook packet handler routine.

**devs, devs_count** If devs_count is 0 (zero), the hook will be activated for all network devices. Otherwise, the hook is activated only for devices listed in the devs array.

### 23.2.1.3 Hook Un-Registration

The following is the knet hook un-registration function:

```c
/** Deletes the list of devices specified in 'devs' from the hook identified * by 'type', 'dir' and 'handler'. If 'devs_count' == 0, or all the active * devices have been deleted from the hook, the hook will be deleted too. * Blocks interrupts during execution. * @param type - the type of the hook from which we want to delete devices. * @param dir - the transmission direction of the hook. * @param handler - the hook callback function to be executed on the packet. * @param devs - array of (knet_netdev_t *) to be deleted from the hook active * devices list. * @param devs_count - the size of the devs array. */
void knet_buf_hook_devs_unregister(knet_hook_type_t type, knet_hook_dir_t dir,
        knet_hook_handler_t handler, knet_netdev_t **devs, int devs_count);
```

### 23.2.1.4 Adding a New Knet Hook

To add a new knet hook, perform the following:

1. Define a new hook type for `knet_hook_type_t` enum in `pkg/kos/knet_hooks.h`. For example, add the following line:

   ```c
   KNET_HOOK_TEST_EXAMPLE = 18,
   ```

2. Set the priority of your hook’s type. This is an important step in a hook design, since the positioning of a hook defines its operation. For example, depending on its position, a hook can or cannot change a packet, the associated network device, or even decide whether to drop the packet before it reaches the next hook. Once a hook’s location is determined, you should add a new entry to the `knet_rx_hooks/knet_tx_hooks` array, in a proper position. For example, if our hook applies only to the incoming (Rx) packets, and you want to set it as the first activated hook, add the following entry to the `knet_rx_hooks` array (`pkg/kos/knet_hooks.c`), just before the `KNET_HOOK_STATISTICS` entry:

   ```c
   { NULL, KNET_HOOK_TEST_EXAMPLE , 0, NULL, NULL },
   ```

3. Implement the hook in your module. The following is a short example:

   ```c
   static knet_hook_ret_type_t test_example_rx_hook(knet_buf_info_t *binfo)
   {
       knet_eth_hdr_t *eh;
   ```
KOS/KNET

```c
knet_icmp_hdr_t *icmph;

/* small example: we are interested in swallowing Ethernet ICMP
 * protocol unreachable messages, if their ethernet destination address is a
 * broadcast/multicast address */
if (binfo->link_proto != KNET_LINK_ETH ||
    binfo->payload_proto != htons(KNET_ETH_TYPE_IP) ||
    ((knet_ip_hdr_t *)binfo->payload)->ip_p != IPPROTO_ICMP)
{
    /* if its not an ethernet frame, an IP datagram, or an ICMP
    * packet, return, continue packet processing. */
    return KNET_PKT_NOT_HANDLED;
}

/* skip non bcast/mcast packets */
    eh = (knet_eth_hdr_t *)(binfo->link_header);
    if (!KNET_ETHER_IS_MULTICAST(eh->ether_dhost))
        return KNET_PKT_NOT_HANDLED;

/* skip icmp msgs other than protocol unreachable */
icmph = binfo->payload + ((knet_ip_hdr_t *)binfo->payload)->ip_hl << 2;
    if (icmph->icmp_type != ICMP_TYPE_DST_UNREACH ||
        icmph->icmp_code != ICMP_CODE_PROTO_UNREACH)
    {
        return KNET_PKT_NOT_HANDLED;
    }

/* found our message... free the network buffer and return "handled", so
 * processing for this packet is performed */
knet_buf_free(*binfo->kb);
return KNET_PKT_HANDLED;
```

4. Register the hook in your module. You should either place the hook's registration in the module's initialization code, or register it by running a ioctl implemented in your module.

```c
knet_buf_hook_devs_register(KNET_HOOK_TEST_EXAMPLE, KNET_RX_HOOK,
    test_example_rx_hook, NULL, 0);
```

It is important to unregister your hook afterwards (upon a module uninit, or as part of a ioctl implementation).
24

Remote Upgrade

24.1 Overview

The **openrg.rmt** is an OpenRG image, loadable via the 'OpenRG Firmware Upgrade' screen of the WBM. It is an image with the addition of a header, which is checked before the upgrade to verify that the image can be used for upgrading OpenRG. The following is an example of an **openrg.rmt** header:

```
rg_hw: UML
dist: UML
prod_version: 4.8.3
version: 40030
```

For more information on using the 'OpenRG Firmware Upgrade' screen, refer to the 'Firmware Upgrade' section of the OpenRG Administrator Manual.

24.2 Remote Upgrade Behavior

When the remote upgrade Main Task is started using `mt_rmt_upd_open`, a timer event that will invoke `mt_rmt_upd_reconf` is set to be called after two minutes. This period of time should enable the WAN connection to be ready. Upon the first call to `mt_rmt_upd_reconf`, the function `mt_rmt_upd_periodic_start` is called.

The `mt_rmt_upd_periodic_start` function begins the remote upgrade process by calling `mt_rmt_upd_start`, and sets a periodic timer that will invoke the `mt_rmt_upd_start` function every 24 hours. This implies that if there is no user interference, then OpenRG tries to perform a remote upgrade after two minutes from reboot, and then every 24 hours.

If a user tries to force a remote upgrade before the 24 hours are over, then `openrg_new_ver_update` is called. All pending timer events are removed, a timer event
for *mt_rmt_upd_periodic_start* is set to be called in 24 hours, and a remote upgrade process starts immediately.

## 24.3 The Remote Upgrade File Utility

The *upd_file_util* application is created during the build process of the OpenRG tree. This application makes modifications or additions to *.rmt* files according to the scenario with which you would like to work. The *upd_file_util* application is located under the *build/pkg/rmt-update* subdirectory of your OpenRG development tree.

### 24.3.1 Creating a New Upgrade File

Call *upd_file_util* with `-C` to create a new *.rmt* upgrade file, using the following format:

```
$ upd_file_util -C -f <upgrade file> -i <input datafile>
```

This creates a new upgrade file (or overwrites an existing one), with no product headers.

### 24.3.2 Adding a Product Header

Call *upd_file_util* with `-A` to add a product header to a *.rmt* upgrade file, using the following format:

```
$ upd_file_util -A -f <upgrade file> -h <hardware> [-d <dist>] 
-v <version> [-e <external version>] [-u <url>]
```

This adds a product header to an existing image at the end of the product's headers.

### 24.3.3 Listing Products in the Upgrade File

Call *upd_file_util* with `-L` to list products in a *.rmt* upgrade file, using the following format:

```
$ upd_file_util -L -f <upgrade file>
```

This lists the image size as written in the generic header, and prints all the products.

## 24.4 Backwards Compatibility

When upgrading your OpenRG to a newer version, sometimes it is necessary to go over the *rg_conf* of the older version and convert it into a new *rg_conf* according to the current version. It is preferable to make modifications in *rg_conf* rather than to restore the default settings, as it may be important to preserve your customer's settings.

The code you must add is located in the *mt_rg_conf_compat.c* file residing under *rg/pkg/main*. The function to which you must add the code is *mt_rg_conf_compat()*. This function goes over the *rg_conf* of the older version and converts it into a new *rg_conf* according to the current version. The following is an extract from the *mt_rg_conf_compat()* function:
if (rg_version_compare(saved_version, "4.1.7") < 0)
{
    convert_tz_name();
    ipsec_temp_type_set();
    ipsec_range_type_set();
    tod_convert();
    bridge_convert();
}

In this example, the code checks whether the previous version of OpenRG is smaller than 4.1.7. If it is, the changes that must be made for this version are applied by calling the listed functions. In the same manner, you may add (in the appropriate places within the `mt_rg_conf_compat()` function) the functions that must be called when performing an upgrade.

Note: `rg_conf` is not changed when upgrading.

You may have a distribution that is equivalent to an existing distribution. OpenRG holds a list of aliases, and can recognize whether your distribution is equivalent to another and perform the upgrade seamlessly.

## 24.5 External Version Description

OpenRG has an external version mechanism that enables you to define your own version on top of the original one. This version is compared in the remote upgrade procedure, and upgrade actions are performed according to the external version changes as well. The external version string is defined in `rg/pkg/include/external_version_data.h`, in the EXTERNAL_VERSION variable. This string can be a simple number that is added to the OpenRG version (i.e OpenRG 4.1.7 external version 16), or a character string describing your version.

The `rg/pkg/main/mt_rmt_upd.c` file contains a mechanism for handling this external version when it is enabled. In order to define a vendor-specific subversion number, use OpenRG’s external version feature. All you have to do is modify the EXTERNAL_VERSION macro in the `pkg/include/rg_version.h` file. This macro is currently defined as:

```c
#define EXTERNAL_VERSION ""
```

You just need to change it to:

```c
#define EXTERNAL_VERSION "1"
```

When character strings are stored in the external version, a comparing function should be registered using the `mt_rmt_upd_register_version_cmp_func` function. This registration should be done in the `mt_rmt_upd_start` function, before the `mt_rmt_upd_start_do` call. Consider the following example of the comparing function:

```c
#include <main/mt_rmt_upd_common.h>

int ext_version_compare(char *v1, char *v2)
{
    if (!v1 && !v2)
        return 0;
    if (!v2)
        return 1;
    if (!v1)
        return -1;
    return strcmp(v1, v2);
}
The comparing function is registered as in the following example:

```c
void ext_version_init(void)
{
    mt_rmt_upd_register_version_cmp_func(
        ext_version_compare);
}
```

After creating this registration function, add a call to it in the `mt_init` function, located in the `pkg/main/mt_init.c` file.
Network Bridging

A bridge is a device that connects several LANs into one by relaying Ethernet frames. The bridge functionality is independent of any higher layer information relayed, such as IP, TCP, or SNMP. Effective relaying is done when all frames are transferred to their destination, with minimum unneeded traffic. To do that the bridge relaying mechanism learns MAC addresses of frames received, caches them, and uses this cache to know where each MAC is connected. OpenRG is an OS-independent bridge that supports VLANs and STP. The bridge is fully configurable via rg_conf or the WBM, and can be used to bridge any OpenRG Ethernet interface (EthoATM, wireless Ethernet, USB and Ethernet).

25.1 Transparent Bridging

OpenRG uses transparent bridging in order to create address lookup tables. Transparent bridging allows OpenRG to learn everything it needs to know about the location of nodes on the network, eliminating the need for a network administrator's intervention. Transparent bridging is consisted of five steps:

- Learning
- Flooding
- Filtering
- Forwarding
- Aging

The underlying logic is illustrated in the following diagram. The depicted segments should be plugged into OpenRG's ports.
Scenario #1:

1. A computer (Node 1) on the first segment (Segment A) sends data to a computer (Node 3) on another segment (Segment B).

2. OpenRG receives the first frame of data from Node 1. It reads the source MAC address and saves it to the lookup table for Segment A. OpenRG now knows where to find Node 1 any time a frame is addressed to it. This process is called **Learning**.

3. Since OpenRG does not know where Node 3 is, it sends the frame to all of the segments except the one on which it arrived (Segment A). When OpenRG sends a frame out to all segments to find a specific node, this is called **Flooding**.

4. Node 3 gets the frame and sends a frame back (through OpenRG) to Node A in acknowledgement.

5. The frame from Node 3 arrives. Now OpenRG can add the MAC address of Node 3 to the lookup table for Segment B. Since OpenRG already knows the address of Node A, it sends the frame directly to it. Because Node 1 is on a different segment than Node 3, OpenRG must connect the two segments to send the frame. This is called **Forwarding**.

6. The next frame from Node 1 to Node 3 arrives. OpenRG now has the address of Node 3, so it forwards the frame directly to it.

Scenario #2:
1. Node 2 sends information to OpenRG for Node 1. OpenRG looks at the MAC address for Node 2 and adds it to the lookup table for Segment A. OpenRG already has the address for Node 1 and determines that both nodes are on the same segment, so it does not need to connect Segment A to another segment for the data to travel from Node 2 to Node 1. Therefore, OpenRG will ignore frames traveling between nodes on the same segment. This is called **Filtering**.

2. **Learning** and **Flooding** continue as OpenRG adds nodes to the lookup tables. Most ordinary switches have plenty of memory for maintaining the lookup tables; but to optimize the use of this memory, they still remove older information so that there is little time wasted in searching through stale addresses. To perform this, OpenRG uses a technique called **Aging**. When an entry is added to the lookup table for a node, it is given a timestamp. Each time a frame is received from a node, the timestamp is updated. OpenRG has a user-configurable timer that erases the entry after a certain amount of time with no activity from that node. This frees up valuable memory resources for other entries.

### 25.2 Flow Description

- **Bridge creation**
  1. A Bridge rg_conf branch is added to rg_conf.
  2. OpenRG reconfigures, and asks dev_if_bridge to reconfigure as well.
  3. dev_if_bridge sends a BRDGCREATE ioctl to the kernel asking to add bridge bridge module adds bridge context.
  4. dev_if_bridge sends BRDGADD ioctls to the kernel asking to add an interface (for each enslaved device).
  5. The bridge module adds device context to the bridge.
  6. The enslaved device is set to promiscuous mode
  7. dev_if_bridge sends BRDGVLANSET ioctls to the kernel, asking to add VLANs.

- **Frame path (bridged)**
  1. The Device driver raises an interrupt for an incoming frame, and queues the frame for processing on the receiving device rx queue. This is performed inside the IP stack.
  2. The Bottom Half pulls the frame out of the device queue.
  3. A bridge hook is called to start bridge handling for incoming frames.
  4. The frame destination is determined by its MAC and by MAC cache.
  5. The frame is queued by the bridge to target the interfaces’ tx queue.
6. The Device driver pulls frames from the tx queue and sends them to network.

- Frame path (routed)
  1. The Device driver raises an interrupt for incoming frames and queues the frame for processing on the receiving device rx queue.
  2. The Bottom Half pulls the frame out of the device queue.
  3. The bridge hook is called to start bridge handling for incoming frame.
  4. The frame destination is determined to be local by MAC.
  5. The frame is queued on the bridge device rx queue.
  6. The Bottom Half pulls the frame out of the bridge device and forwards it to the appropriate processing module.

- Frame path (transmitted by bridge)
  1. The network module queues a frame on bridge device tx queue.
  2. The Bottom Half pulls the frame out of the queue and calls bridge tx function.
  3. The frame destination is determined by its MAC and by MAC cache.
  4. The frame is queued by the bridge to target interfaces’ tx queue.
  5. Device driver pulls frames from the queue and sends them to network.

### 25.3 Virtual LAN (VLAN)

As networks continue to grow in size and complexity, there is an increasing demand for virtual local area networks (VLANs) to provide some way of structuring this growth logically. Basically, a VLAN is a collection of nodes that are grouped together in a single broadcast domain that is based on something other than physical location.

#### 25.3.1 VLAN Advantages

Here are some common reasons why you may consider VLANs:

**Security**  Separating systems that have sensitive data from the rest of the network decreases the chances that people will gain access to information they are not authorized to see.

**Projects/Special applications**  Managing a project or working with a specialized application can be simplified by the use of a VLAN that brings all of the required nodes together.
Performance/Bandwidth  Careful monitoring of network use allows the network administrator to create VLANs that reduce the number of router hops and increase the apparent bandwidth for network users.

Broadcasts/Traffic flow  Since a principle element of a VLAN is the fact that it does not pass broadcast traffic to nodes that are not part of the VLAN, it automatically reduces broadcasts. Access lists provide the network administrator with a way to control who sees what network traffic. An access list is a table the network administrator creates that lists which addresses have access to that network.

Departments/Specific job types  Companies may want VLANs set up for departments that are heavy network users (such as multimedia or engineering), or a VLAN across departments that is dedicated to specific types of employees (such as managers or sales people). Even though not all networking devices comply with the VLAN extension, there is a way to handle both:

- Every section of the network that consists of only VLAN-unaware (untagged) devices is logically assumed to participate in only one VLAN.

- All the VLAN-aware (tagged) devices can be configured to participate in any VLAN or several VLANs as well.

- Connecting a tagged section to an untagged section will always be done with a tagged device that assumes the untagged section belongs to a certain VLAN. All the traffic from this section into the tagged network will be assigned a default VLAN, and all the traffic in the tagged network that carries this VLAN (the default one) will be forwarded into the untagged section -- not before it is stripped from VLAN tags.

25.3.2 Demonstrating VLANs

Let's take a real-life example to make things clearer. Jungo's R&D team and administration group are all connected to the same physical network. However, the R&D workstations need to be connected to dedicated storage and DHCP servers.
The solution comes by using a VLAN switch that can handle both tagged/untagged traffic and connect all the workstations to it using the following port configuration:

- P1 - untagged, default VLAN 14.
- P2 - untagged, default VLAN 15.
- P6 - untagged, default VLAN 15.
- P7 - tagged, allows VLANs 14,15.

Using this configuration, VLAN 14 is shared between the administration and file server. VLAN 15 is shared between the R&D workstation, DHCP and file server. The advantage of managing the configuration this way, is that any workstation can be moved from network to network without rewiring. It is interesting to note that the only hardware that needs to be aware of the VLAN is the DHCP server and VLAN switch. The rest of the stations are unaware of the existence of VLANs in the system at all. VLANs are implemented by adding a VLAN tag to the Ethernet header that specify the VLAN and priority of this frame. VLAN ID (VID) values range between 1 to 4094; priorities range from 0 to 7.
25.4 Configuration File Entries & Examples

A bridge is represented in the configuration file as a device entry: /dev/<device name>. This entry represents both the bridge and the bridge device. Apart from general fields like 'DHCPC' or 'enabled', the bridge has a unique entry called 'enslaved' that defines all the devices enslaved to this bridge along with their enslaved behavior. Additionally, the bridge will always appear as an enslaved device of itself, as a way to state that the bridge device is also one of the bridge ports.

25.4.1 Example 1: A Typical Ethernet-USB Bridge

The following configuration file entry illustrates how a typical Ethernet-USB reference looks like.

```
(dev
  (br0
    (description(LAN Bridge))
    (enslaved
      (ixp0
        (stp(1))
      )
      (usb0
        (stp(1))
      )
      (br0)
    )
    (static
      (ip(192.168.1.1))
      (netmask(255.255.255.0))
    )
    {... other device fields }
    {... other devices}
  )
)
```

25.4.2 Example 2: Ethernet--USB Bridge with VLAN

Extending the previous example, the following configuration file entry illustrates how a typical Ethernet-USB reference looks like with VLAN support.

```
(dev
  (br0
    (description(LAN Bridge))
    (enslaved
      (ixp0
        (stp(1))
        (tagged(0))
        (def_vlan(-1))
      )
      (usb0
        (stp(1))
        (tagged(0))
      )
    )
    (static
      (ip(192.168.1.1))
      (netmask(255.255.255.0))
    )
    {... other device fields }
    {... other devices}
  )
)
```
25.4.3 Example 3: Detailed VLAN Example

Like the previous example, this implementation supports VLANs. This time VLANs are configured in the system in the following way: The bridge device is an untagged device participating VLAN 112. The Ethernet device is a tagged device participating VLANs 112, 113, 114. The USB device is a tagged device participating VLANs 113, 114.

25.5 Operating System Implications

Bridge kernel handling in OpenRG is all done via Kos/Knet abstraction, currently covering Linux 2.2/4 and VxWorks. Please keep in mind that Linux and VxWorks are dissimilar in network implementation, and a comprehensive understanding of the differences (such as skb Vs. Mbuff) is recommended when writing a cross-OS code, even when using KOS abstraction.
25.6 Security Implications

Using a LAN/WAN bridge is safer than using a HUB, or other device, since the bridge does not transfer frames that have an identical source and destination. However, the bridge does expose the LAN topology to the WAN, making this a configuration that is relatively insecure. In order to protect the LAN, OpenRG's Firewall should be applied on the WAN device directly, protecting the LAN from harmful frames arriving from the WAN. Applying the Firewall on the bridge device will not protect the LAN, because it is only triggered for frames that go 'up' towards OpenRG. Moreover, it will make OpenRG's management inaccessible, and prevent further reconfiguration.

Figure 25.3 Applying the Firewall on the Bridge Device

Figure 25.4 Applying the Firewall on the WAN Device
25.7 Command Line Interface Bridge Commands

**bridge info**  Displays a list of devices enslaved to the bridge, their flags and a dump of the MAC cache for each device.

**net ifconfig**  Though not a dedicated bridge command, it can be used to obtain information about the bridge device. Enslaved bridge devices appear as entries in 'depend_on_list'.

```
OpenRG> bridge info
    Bridge br0 1c:c7:88:99:f5:af
         ixp1 30:4b:94:34:3f:b5 flags:0x2f
          00:ff:1d:d2:34:4c
          00:0a:cd:03:f0:85
          00:00:41:2e:d2:66
          00:50:fc:e4:44:32
          00:c1:26:00:95:bb
          00:fd:ee:ff:fc:11
          00:4d:f4:70:0e:82
    usb0 18:d1:fb:dc:f1:7b flags:0x2f
   18:d1:fb:dc:f1:7b
         ixp0 1e:c1:86:58:93:69 flags:0x2f
          00:11:d4:ce:56:01
          le:cl:86:58:93:69
```

```
OpenRG> net ifconfig br0
    depend_on_list=ixp0(0x216b90), usb0(0x216c38), ixp1(0x216990)
    next=ipsec1(0x1d74a0)
    mac=1c:c7:88:99:f5:af
    ip=192.168.91.139,    netmask=255.255.255.0
   Aliases:
    Returned 0
```

25.8 Relationship with other Modules

The bridge module is mainly initiated by hooks. When the module initializes, the bridge registers as a handler for knet_rx_hooks. When a new bridge is created, the new bridge context is added to the list of available bridges. It is important to note that every arriving frame goes over the hook list. When a frame gets to the bridge knet_hook, an initial test is done to determine whether this device is bridged at all. If it is bridged, the frame is passed to the bridge module for bridging. Otherwise, the knet_hook is ignored and the frame goes on to the next hook and eventually to the receiving device rx queue. As far as other User-Mode application are concerned, the bridge interface represents an Ethernet device with an IP assigned to it. There is no special behavior or addressing that should be used.
25.9 Performance Implications

The following conditions may affect network performance and considered normal behavior.

25.9.1 Frame RX Sequence

During the frame rx sequence, the bridge is registered on the hook list. The hook is called for every rx frame, and performs a search to find out if the rx device is enslaved to a bridge. This process causes a slight decrease in network performance.

25.9.2 Bridge Forwarding Mechanism

All bridged traffic is handled by the bridge forwarding mechanism. Even if the data path is from an enslaved to an un-enslaved device, the bridge forwarding mechanism works for every passing frame. All of the connections that go in some way through the bridge, may suffer a decrease in performance. A reasonable decrease is measured in up to 10% and will usually be noticed only in small frames (512 bytes or less).

25.9.3 VLAN Tagging/Untagging

When a bridge is configured to function as a VLAN bridge, it will usually avoid any redundant tagging or untagging. Yet, every stream of frames from an untagged device to a tagged one will force the bridge to tag the frames. Since tagging and untagging actions are usually negligible unless during a tagging operation, a frame does not have enough headroom for the VLAN header. In that case, the bridge will be forced to reallocate new space for the frame. This becomes significant if all the frames in the stream need to be reallocated. Normally network drivers allocate enough headroom for tagging operations -- the only place where it expected to take place is when a local OpenRG application is sending a stream of frames to a tagged
interface. Since OpenRG's default headroom allocation for Ethernet devices is always 16 bytes, every frame will have to be reallocated on its way out.

25.10 Troubleshooting Bridge Connectivity Problems

Since the bridge is a layer-2 connectivity device, malfunctions that are related to it will generally first be classified as 'loss of connectivity'. The following examples show typical connectivity problems, and how to recognize them.

25.10.1 Bridge STP Learning

When the bridge loads it will spend some time, about 30 seconds, in "STP Learning" mode. The bridge uses the spanning tree protocol (STP) to discover the breadth of the network and identify potential loops. During this time, the bridge will block all traffic except frames to and from the bridge device itself. This leaves OpenRG's management accessible even when the bridge does not pass frames between devices. This behavior is commonly mistaken as a bridge failure, and should be taken into account when writing automatic tests. The following scenario may serve as a common example:

1. OpenRG act as a bridge between Ethernet and WLAN devices.
2. A user accesses OpenRG's Web-based management from a PC station connected to a WLAN interface.
3. The user then changes the bridge's IP address from x.x.1.1 to x.x.2.1.
4. The user tries to use the FTP protocol to transfer a file from a server connected to the Ethernet interface of the bridge, but FTP server cannot be reached.

The solution would be to simply wait for 30 seconds before accessing FTP server, and wait for the STP learning process to end.

25.10.2 Bridge MAC Changes

The OpenRG bridge has utilizes a specific logic to determine its MAC address. Changes to the bridge configuration can cause OpenRG to change its MAC address. If a LAN station was in use to communicate with the bridge for a while, and then the bridge configuration changed, the LAN station will still have an ARP entry for the old MAC of the bridge and will try to use this MAC to access the bridge. The bridge will ignore those frames because they do not match its MAC. This behavior will continue until the ARP entry in the station expires, and the station "learns" the new MAC address. The following scenario may serve as a common example:

1. OpenRG is bridging between USB, Ethernet and WLAN device.
2. A user accesses OpenRG from a station connected to the WLAN interface.

3. Using OpenRG’s Web-based management, the user removes the USB device from the bridge.

4. After confirming the change the Web-based management stops responding.

The solution would be to wait for the ARP entries to expire, usually after about 30 seconds. Alternatively, you can flush the ARP table after committing the change by using the following command:\(^1\):

\[
\text{arp \(-d\) *}
\]

### 25.10.3 VLAN Definitions

When changing the bridge's VLAN configuration, you should take into account two important factors:

1. Any interface in a bridge is always either tagged or untagged, but never both. So if you're accessing OpenRG via an untagged device do not set it as "tagged" before you know whether your workstation can produce tagged traffic. The following scenario may serve as a common example:

   1. OpenRG bridges two Ethernet devices, ixp0 and ixp1, via br0.

   2. A user accesses OpenRG via ixp0, and sets up the following VLAN configuration:

      - **ixp0** tagged, participating VLANs 2001, 2002 and 2003.
      - **ixp1** untagged, participating VLAN 2001.
      - **br0** untagged, participating VLAN 2002.

   3. After confirming these parameters, OpenRG becomes inaccessible.

Since the user is connected to a tagged interface (ixp0), it must be able to produce tagged frames. This can be done using Linux OS tools such as vconfig, or by using another VLAN device between the user's workstation and OpenRG.

2. The bridge device is always untagged. This means that at any point in time every bridge device 'sees' only one VLAN, rendering the Web-based management accessible only via this VLAN. Bear in mind that OpenRG is often only accessible from very few devices. Therefore, before confirming changes, make sure that you are still able to access the Web-based management after the changes take effect. The following scenario may serve as a common example:

   1. OpenRG bridges two Ethernet devices, ixp0 and ixp1, via br0.

\(^1\) This command applies to the Windows command line only.
2. A user accesses OpenRG via ixp0, and sets up the following VLAN configuration:

   - **ixp0** tagged, participating VLANs 2001, 2002 and 2003.
   - **ixp1** untagged, participating VLANs 2001 and 2004.
   - **br0** untagged, participating VLAN 2002.

3. The user configures his workstation to use VLAN 2001.

4. OpenRG becomes inaccessible.

   Although the user is using a VLAN that exist in the bridge (2001) this is not the VLAN the bridge device is sharing. This leaves OpenRG's management out of reach. Setting the user's workstation to use VLAN 2002 will solve the problem.

### 25.11 Terminology

This chapter makes use of the following networking terms and concepts:

**Network** A network is a group of computers connected together in a way that allows information to be exchanged between the computers.

**Node** A node is anything that is connected to the network. While a node is typically a computer, it can also be something like a printer or network attached storage.

**Segment** A segment is any portion of a network that is separated, by a switch, bridge or router, from other parts of the network. on the computer's mother board.

**Unicast** A unicast is a transmission from one node addressed specifically to another node.

**Multicast** In a multicast, a node sends a frame addressed to a special group address. Devices that are interested in this group register to receive frames addressed to the group.

**Broadcast** In a broadcast, a node sends out a frame that is intended for transmission to all other nodes on the network.

**Port** A port in a bridge means any connection (either physical or virtual) that is bridged by the bridge relaying mechanism. This should not be confused with a TCP/UDP port number.

**VLAN tag** An Ethernet header extension that enlarges the header from 14 to 18 bytes. The VLAN tag contains the VLAN ID and priority.

**VLAN tagged/untagged** A device or device port that receives and transfers only frames with/without VLAN tag respectively.

**VLAN tagging/untagging** Adding/stripping a VLAN tag to/from an Ethernet frame.

**VID/VLAN ID** Ranging from 1 to 4094, VID represents the VLAN that a certain frame/port belong to.
**Enslaved interface** A device that is bridged with all incoming traffic for it intercepted by the bridge hook.

**TH - Top half** The part of interrupt processing that takes place at interrupt time.

**BH - Bottom half** The part of interrupt processing that takes place out of interrupt time, or on soft interrupt time, while BH handler is de-queueing.

**RX** Receive.

**TX** Transmit.

**Bridge device** Unlike bridge module, the bridge device is the Ethernet device that represents the bridge to OpenRG’s user-mode. The bridge device name will always be formatted as BRn (n - bridge number, starting with 0).

**KOS/KNET** An operating system abstraction layer.

**STP** Spanning Tree Protocol, a mechanism applied by many bridges to locate loops in the network topology and disconnect them.
26

Extending the SNMP Agent

OpenRG includes a Simple Network Management Protocol (SNMP) Agent that allows a Network Management Station (NMS) to remotely configure the gateway’s features using SNMP messages. OpenRG’s SNMP Agent supports the v1, v2c and v3 SNMP protocols and, it implements standard Management Information Bases (MIBs) such as MIB-II and SNMPv3 MIB modules.

26.1 Architecture

The SNMP Agent is one of Main Task’s applications. It consists of an SNMP engine and a command responder module. The SNMP engine is responsible for message processing, message security application, access control, and dispatching the message to the command responder. The command responder module is responsible for providing access to managed MIBs and executing management commands. The SNMP Agent is also capable of generating SNMP notifications (traps and informs).
26.1.1 The Agent's Execution Flow

Figure 26.1 The SNMP Agent Execution Flow
As shown in Figure 26.1, the entire process of SNMP message processing is completely synchronous. This means that none of the described steps should be blocking, most importantly, the MIB access step.

### 26.1.2 Management Information Base Data

Management Information Base (MIB) object data can be stored in various places. For example, persistent values can be stored as entries in the configuration database, where volatile values can be stored internally in the MIB module, or even fetched from kernel modules using IOCTLs. The important guide regarding MIB data access is that getting or setting values to MIB objects must be a non-blocking operation, since it is executed as part of the synchronous flow of SNMP message processing.

### 26.2 General Configuration of the SNMP Agent

General properties of OpenRG’s SNMP Agent can be controlled by SNMP entries in the configuration database. These properties include:

- Enabling or disabling the agent
- Read-only and read-write community strings
- NMS IP Address (‘Trusted Peer’)
- Enable/disable outgoing SNMP Traps
- Version of Trap messages
- Trap receiver IP address
- Trap messages community string

These properties are controlled by configuration file entries found under `/snmp`. For more information, refer to Section 18.1.

### 26.3 Supported MIBs

The following MIBs are currently supported by OpenRG’s SNMP Agent. Their definition files are located under the `rg/pkg/ucd_snmp/mibs` directory of the development tree.

Note: Parent MIBs may have more children than specified herein. Only the supported child MIBs are listed.

- `snmpTargetMIB`
Extending the SNMP Agent

- snmpTargetAddrEntry
- snmpTargetParamsEntry
- snmpTargetMIB

- mib-2
  - system
  - sysORTable
  - interfaces
  - snmp
    - tcp
    - icmp
  - ip
  - udp
    - snmpVacmMIB
    - snmpSet

- snmpNotificationMIB
  - snmpNotifyTable
  - snmpNotifyFilterTable
  - snmpNotifyFilterProfileTable

- snmpModules
  - snmpCommunityMIB

- jungoMib
  - jOpenrgDevObjects
  - jOpenrgDevPPPObects
  - jOpenrgSecObjects
  - jOpenrgSecFwObjects
26.4 Implementing New MIBs

The following section describes the process of extending OpenRG's SNMP Agent to support new MIB modules. The section starts with an explanation about the MIB definition document, which is the technical requirement description of new MIBs to be implemented. Then, it covers the three parts of MIB module implementation: module definition and initialization, one (or more) variable handling routine (FindVarMethod), and several write routines (WriteMethods). The Watchdog example described in Chapter 5 is used to demonstrate the process of implementing a MIB module.

26.4.1 The MIB Definition Document

Implementing a new MIB modules starts with a MIB definition file. This is the ASN document describing the MIB (its objects, groups, compliance statements, notifications, etc.). This definition file provides all of the required information regarding the MIB objects to be implemented. This includes the most basic properties of the MIB, such as the MIB's root OID, the object's OIDs, the MIB objects types, value ranges, default values, MIB object access rights, and advanced properties such as a detailed description of each MIB object's behavior. The following is the MIB document describing the Watchdog MIB:

WATCHDOG-MIB DEFINITIONS ::= BEGIN
As you can see, this simple MIB contains three objects which control the functionality of the Watchdog:

- **watchDogEnabled** A TruthValue object that controls whether the Watchdog feature is enabled or not.

- **watchDogMargin** An Integer object specifying the Watchdog’s time-frame in seconds.

- **watchDogEndMessage** A DisplayString object specifying the message to be displayed when the Watchdog reboots the system.

### 26.4.2 Module Definition and Initialization

The initialization routine is called when the agent task is opened (at boot time). The initialization routine of a MIB module should register the objects of this MIB to the agent. The registration process simply associates OIDs with variable handling routines, and it serves two purposes: specifying the variables implemented (OIDs added to the overall tree of OIDs.
manageable by the agent), and assigning handling routines to variables (routines that will be called whenever SNMP operations are performed on the registered OIDs).

Let's review the initialization code for our Watchdog MIB:

```c
/*
 * watchDogMib_variables_oid:
 * this is the top level oid that we want to register under. This
 * is essentially a prefix, with the suffix appearing in the
 * variable below.
 */
oid watchDogMib_variables_oid[] = {1,3,6,1,4,1,10799,9999};

/*
 * variable2 watchDogMib_variables:
 * this variable defines function callbacks and type return information
 * for the watchDogMib mib section
 */
struct variable2 watchDogMib_variables[] = {
    /* magic number, variable type, ro/rw, callback func, L, oidsuffix
     * (L = length of the oidsuffix)
     */
    #define WATCHDOGENABLED 1
    {WATCHDOGENABLED, ASN_INTEGER, RWRITE, var_watchDogMib, 2, {1,1}},
    #define WATCHDOGMARGIN 2
    {WATCHDOGMARGIN, ASN_INTEGER, RWRITE, var_watchDogMib, 2, {1,2}},
    #define WATCHDOGENDMESSAGE 3
    {WATCHDOGENDMESSAGE, ASN_OCTET_STR, RWRITE, var_watchDogMib, 2, {1,3}},
};

/* init_watchDogMib():
 * Initialization routine. This is called when the agent starts up.
 * At a minimum, registration of your variables should take place here.
 */
void init_watchDogMib(void)
{
    /* register ourselves with the agent to handle our mib tree */
    REGISTER_MIB("watchDogMib", watchDogMib_variables, variable2,
        watchDogMib_variables_oid);
}
```

### 26.4.2.1 MIB Root OID

`watchDogMib_variables_oid` (line 7) is an OID array defining the root OID of our MIB. It is set to the OID of `watchDogMib`, which is defined in the MIB definitions document as `jungoMib 9999`, i.e. 1.3.6.1.4.1.10799.9999. Note that `watchDogMib_variables_oid` is given as a parameter to `REGISTER_MIB` (line 35) in the MIB initialization routine.

### 26.4.2.2 MIB Variables Array

`watchDogMib_variables` is an array describing all objects implemented. Each entry in this array corresponds to one object in the MIB tree (or one column in the case of table entries), and these should be listed in increasing OID order. A single entry consists of six fields:

- A magic number
- A type indicator (from the values listed in `<snmp/lib/snmp_impl.h>`)  
- An access indicator (essentially RWRITE or RONLY)  
- The name of the routine used to handle this entry
• The length of the OID suffix used

• An array of integers specifying the OID suffix

A magic number is used to easily identify a MIB variable. The magic number is a positive integer unique for each object in the registered objects array (more precisely, magic values should be distinct within a single variable handling routine). The most common practice is to define the magic number of an object as the final sub-identifier of the MIB object's OID. The type for a single entry in the variables array is \texttt{struct variableN} where \( N \) is the length of the longest OID suffix in the array.

In the current example, \texttt{struct variable2} is used (line 14), since the OID suffix of each of our three variables never exceeds 2 sub-ids. Note that in practice, only certain sizes of the structure \texttt{variableN} are defined (listed in \texttt{<agent/var_struct.h>}), being sufficient to meet the common requirements. If your particular module needs a non-supported value, the easiest thing is simply to use the next largest value that is supported. Let's closely examine the following line:

```c
#define WATCHDOGENDMESSAGE 3
    {WATCHDOGENDMESSAGE, ASN_OCTET_STR, RWRITE, var_watchDogMib, 2, {1,3}},
```

This array element defines the properties of the \textit{watchDogEndMessage} MIB object. The magic is defined as WATCHDOGENDMESSAGE (whose value is 3, as the last sub-id of \textit{watchDogEndMessage}). The type is ASN\_OCTET\_STR which is the built-in ASN.1 type of \textit{watchDogEndMessage}. The variable has RWRITE access (corresponding with the MIB object's MAX-ACCESS). The handling routine for this variable is \texttt{var\_watchDogMib}. The OID suffix of this variable is 1.3, corresponding to the OID defined in the MIB (\textit{watchDogObjects 3}).

### 26.4.2.3 MIB Registration

\texttt{init\_watchDogMib} is the MIB's initialization routine. This routine is called when agent starts up, and it should perform any once-only initializations needed for the MIB module. For example, it can initialize any internal data structures needed by the MIB module. At minimum, the initialization routine should register the MIB's variables. Variable registration is done using the \texttt{REGISTER\_MIB} macro. In the current example, the following lines perform the registration:

```c
REGISTER\_MIB("watchDogMib", watchDogMib\_variables, variable2, watchDogMib\_variables\_oid);
```

Where "\texttt{watchDogMib}" is a string used for identification purposes, \texttt{watchDogMib\_variables} is the variables array to register, \texttt{variable2} is the type of the elements in the given variables array, and \texttt{watchDogMib\_variables\_oid} is the root OID of our MIB, under which the variables are located. The initialization function should be called from \texttt{cust\_mibs\_init()}, found in the following file:

```
pkg/vendor/user_mode/cust_mibs_init.c:
```
26.4.3 The Variable Handling Routine (FindVarMethod)

This obligatory routine is that which actually handles a request for a particular variable instance. This is the routine that appeared in the variableN structure, so while the name is not fixed, it should be the same as was used there. This routine has six parameters, which will be described in turn. Four of these parameters are used for passing in information about the request, these being:

```
struct variable *vp, /* The entry in the variableN array, for the
  * object under consideration.
  * Note that the name field of this structure
  * has been completed into a fully qualified OID,
  * by prepending the prefix common to the whole
  * array.
  */
oid *name, /* The OID from the request */
int *length, /* The length of this OID */
int exact, /* A flag to indicate whether this is an exact
  * request (GET/SET) or an 'inexact' one (GETNEXT).
  */
```

Four of the parameters are used to return information about the answer. The function also returns a pointer to the actual data (object's value) for the variable requested (or NULL if this data is not available for any reason). The other result parameters are:

```
oid *name; /* The OID being returned */
int *length; /* The length of this OID */
WriteMethod **write_method; /* A pointer to the SET function
  * (WriteMethod) for this variable.
  */
```

Note that two of the parameters (name and length) serve a dual purpose, being used for both input and output. The first thing that this routine needs to do is to validate the request, to ensure that it does indeed lie in the range implemented by this particular module. This is done in slightly different ways, depending on the style of the module, so this will be discussed in more detail later. At the same time, it is common to retrieve some of the information needed for answering the query. Then the routine uses the Magic Number field from the vp parameter to determine which of the possible variables being implemented is being requested. This is done using a switch statement, which should have as many cases as there are entries in the variableN array (or more precisely, as many as specify this routine as their handler), plus an additional default case to handle an erroneous call.

Each branch of the switch statement needs to ensure that the return parameters are filled in correctly, set up a static return variable with the correct data, and then return a pointer to this value. These can be done separately for each branch, or once at the start, being overridden in particular branches if necessary. In fact, the default validation routines make the assumption that the variable is both read-only, and of integer type (which includes the COUNTER and GAUGE types among others), and set the return parameters write_method and var_len appropriately.

These settings can then be corrected for those cases when either or both of these assumptions are wrong. Note that because the routine returns a pointer to a static result, a suitable variable must be declared somewhere for this. Two global variables are provided for this purpose — `long_return` (for integer results) and `return_buf` (for other types). This latter is a
generic array (of type u_char) that can contain up to 256 bytes of data. Alternatively, static variables can be declared, either within the code file, or local to this particular variable routine. Let's review the FindVarMethod of our Watchdog MIB:

```c
/* var_watchDogMib(): */
/* This function is called every time the agent gets a request for */
/* a scalar variable that might be found within your mib section */
/* registered above. It is up to you to do the right thing and */
/* return the correct value. */
/* You should also correct the value of "var_len" if necessary. */
unsigned char *var_watchDogMib(struct variable *vp, oid *name, size_t *length, int exact, size_t *var_len, WriteMethod **write_method)
{
    set_t **wd_set;
    static s32 s32_ret;
    char *boot_msg;
    if (header_generic(vp, name, length, exact, var_len, write_method) == MATCH_FAILED)
    {
        return NULL;
    }
    /* this is where we do the value assignments for the mib */
    /* results. */
    wd_set = set_get(rg_conf, Swatchdog);
    switch (vp->magic)
    {
    case WATCHDOGENABLED:
        *write_method = write_watchDogEnabled;
        s32_ret = set_get_path_flag(wd_set, Senabled);
        *var_len = sizeof(s32_ret);
        return (unsigned char *)&s32_ret;
    case WATCHDOGMARGIN:
        *write_method = write_watchDogMargin;
        s32_ret = set_get_path_int(wd_set, Smargin);
        *var_len = sizeof(s32_ret);
        return (unsigned char *)&s32_ret;
    case WATCHDOGENDMESSAGE:
        *write_method = write_watchDogEndMessage;
        boot_msg = set_get_path_strz(wd_set, Sboot_msg);
        *var_len = strlen(boot_msg);
        return (unsigned char *)boot_msg;
    default:
        break;
    }
    return NULL;
}
```

### 26.4.3.1 Request Validation

As previously noted, a FindVarMethod should start with validation of the request. This ensures that the requested OID is found within the scope of the variable handling routine. For non-table MIB modules (i.e. for a FindVarMethod associated with a bunch of scalar MIB objects), request validation is performed using the `header_generic()` utility call. Its parameters are the exact same parameters of the FindVarMethod. Its return value is an integer indicating whether the validation succeeded (MATCH_FAILED or MATCH_SUCCEEDED).

If validation fails, then the variable handling routine should return NULL, indicating the object "does not exist" (i.e. not found within the scope of variables handled by this routine). In the current Watchdog MIB example, request validation is performed in lines 17-21. Note that `header_generic` does one more thing besides validating the request: in case validation
succeeded, header_generic sets the OID of the result variable. For an exact (GET) request, this is purely the same OID requested; However for non-exact (GETNEXT) request, the name and length parameters are updated to the OID of the matched variable.

### 26.4.3.2 Data Retrieval

The other main job of the request handling routine is to retrieve any necessary data, and return the appropriate answer to the original request. As has been indicated earlier, the different cases are handled using a switch statement, with the Magic Number field of the vp parameter being used to distinguish between them. The data necessary for answering the request can be retrieved for each variable individually in the relevant case statement, or using a common block of data before processing the switch. The data itself can be read from various of sources, depending where it is actually stored: from internal data structures of the MIB module, from the kernel using ioctls, from some external API calls, or from OpenRG's rg_conf.

An important guide already stated—no matter the source of the data, data retrieval must not be a blocking operation, since the variable handling routine is executed during the execution flow of the agent task handling an SNMP message (i.e. part of Main Task's execution flow). The variable handling routine should return a pointer to the result value. The result data should be treated as a generic buffer of data, meaning the pointer should be casted to the pseudo-generic u_char* type. Also note that the pointer returned should be valid outside the scope of the FindVarMethod; for example, pointers to static variables or pointers to dynamically allocated data may be used. In our Watchdog MIB example, the data is retrieved by reading the appropriate rg_conf fields.

The Integer typed values (watchDogEnabled and watchDogMargin) are stored in the static variable s32_ret (lines 30 and 35), and the length of the returned result (*var_len) is set to the byte size of returned result (lines 36, 41). Finally, a pointer to the s32_ret static variable is returned (lines 37, 42). The Octet-String value (watchDogEndMessage) is simply extracted from rg_conf (set_get_path_strz returns a pointer to the allocated value of the rg_conf field). The length of the returned result is set to the string length of the retrieved rg_conf value, and the pointer to the rg_conf value is returned (lines 40-42).

### 26.4.4 The Write Routine (WriteMethod)

#### 26.4.4.1 SET-Request Overview

The design of SNMP calls for all variables in a SET request to be done "as if simultaneously" - i.e. they should all succeed or all fail. However, in practice, the variables are handled in succession. Thus, if one fails, it must be possible to "undo" any changes made to the other variables in the request. This is a well understood requirement in the database world, and is usually implemented using a "multi-stage commit". This is certainly the mechanism expected within the SNMP community (and has been made explicit in the work of the AgentX extensibility group). In other words, the routine to handle setting a variable will be called more than once, and the routine must be able to perform the appropriate actions depending on how far through the process we currently are. This is determined by the value of the action parameter.
26.4.4.2 A SET-Request in OpenRG’s Agent

The agent's variable-set mechanism traverses the list of OIDs (variables) of the given SET-PDU four times, in four different stages (meaning it doesn't simply set a variable and proceed to the next). The four stages are called: RESERVE1, RESERVE2, ACTION, and COMMIT. As mentioned, the mechanism traverses the whole list of "variables-need-to-be-set" before proceeding from one stage to the next. There are 2 additional special stages that perform error handling, which are called: FREE and UNDO. The flow between stages is as follows:

Figure 26.2 Stages of SET-Request processing

As noted in the overview, if a SET Request has several objects to set, and failure arises when processing a single object, then all objects are not supposed to be set. For example, suppose we received a SET pdu with 4 variables need to be set. If variable 3 failed the RESERVE2 stage, then the FREE stage is applied to the whole list of variables. The implementor must remember that processing of a single stage is applied to the entire list of objects, sequentially. Proceeding from a stage to another occurs only if the stage succeeded for all objects in the list. If one object failed a stage, then the algorithm switches to the appropriate error-handling stage (FREE or UNDO), and applies that stage to the entire list of objects.
26.4.4.3 The Roles of SET-Request Stages

The following is a suggested role for each of the SET-Request stages. Note this is only a general suggestion and it is not adequate for all situations.

**RESERVE1** is used to check the syntax of all the variables provided (verify that the new value is of the correct type, has the correct value and length, etc.)

**RESERVE2** may be used for resource allocation, and for setting the new value in some temporary storage.

**FREE** undo any changes made by the RESERVE1 and RESERVE2 stages (freeing resources acquired and undoing any actual change made)

**ACTION** is used to perform any additional validations of the new values. This is mostly used when needed to test relationships among values of several variables altogether (in contrast to the RESERVE1 validations, which simply check value correctness of a single variable). In this stage we perform the SET command, changing the variable's value, in a reversible manner. In some cases, an additional action needs to be performed due to the value change—doing it in this stage is recommended.

**UNDO** resets the variable to its pre-ACTION value, and free any resources acquired in previous stages.

**COMMIT** discards the saved information, make the value change permanent. Free any resources used. This stage should be error free (should not fail).

As noted, this is only a general suggestion. Different scenarios arise when handling different kind of variables, depending on storage type (variable's value is stored internally in the MIB module, in `rg_conf`, etc.), whether additional action needs to be performed due to the value change, whether relationships between variables needed to be tested, etc. As long as you understand the design and flow sequence of the various stages, you can use it according to your specific needs.

In case you need to validate the value against the values of other variables ("cross-validations", that test relationships between variables), the validations must be executed after the new values for all the objects are set. Since the mechanism traverses all PDU objects before proceeding to the next stage, writing new values must be carried out in a stage that precedes the validation stage.

26.4.4.4 The WriteMethod

The heart of SET handling is the `write_method` parameter from the variable handling routine. This is a pointer to the relevant routine for setting the variable in question. This routine should be declared as follows:

```c
int write_variable(  
    int action, /* stage indicator. can be one of the following:  
    * SNMP_STATE_RESERVE1, SNMP_STATE_RESERVE2,  
    * SNMP_STATE_FREE, SNMP_STATE_ACTION, SNMP_STATE_UNDO,  
    * SNMP_STATE_COMMIT */  
)  
```
The return value of the routine is simply an indication of whether the current stage of the SET was successful or not. Note that it is the responsibility of this routine to check that the OID and value provided are appropriate for the variable being implemented. This includes (but is not limited to) checking:

- The OID is recognized as one this routine can handle (this should be true if the routine only handles the one variable, and there are no errors in the main variable routine or driving code, but it does no harm to check).
- The value requested is the correct type expected for this OID.
- The value requested is appropriate for this OID (within particular ranges, suitable length, etc).

Example for return codes of this function are SNMP_ERR_NOERROR, SNMP_ERR_WRONGTYPE, SNMP_ERR_WRONGLENGTH, SNMP_ERR_WRONGVALUE, SNMP_ERR_INCONSISTENTVALUE, etc.

26.4.4.5 WriteMethod Examples

The following are examples of write-methods for two of the Watchdog MIB objects, watchDogMargin and watchDogEndMessage. These examples are of course very trivial, since no cross-validations are needed, nor any special storage allocations, nor special actions needed to be performed. The examples are simple, since setting one of these MIB objects simply involves writing to an rg_conf field.

```c
int write_watchDogMargin(int action, u_char *var_val,
                        u_char var_val_type, size_t var_val_len, u_char *statP,
                        oid *name, size_t name_len)
{
    snmpa_t *t = &snmpa_task;
    switch (action)
    {
    case SNMP_STATE_RESERVE1:
        return snmp_int_object_validate(var_val, var_val_type,
                                         var_val_len, 30, 60);
    case SNMP_STATE_RESERVE2:
    case SNMP_STATE_FREE:
    case SNMP_STATE_ACTION:
    case SNMP_STATE_UNDO:
        break;
    case SNMP_STATE_COMMIT:
        set_set_path_int(rg_conf, Swatchdog"/"Smargin, *(s32*)var_val);
        if (t->cb.reconf_request_set)
            t->cb.reconf_request_set(0);
        break;
    }
    return SNMP_ERR_NOERROR;
}
```
int write_watchDogEndMessage(int action, u_char *var_val,
    u_char var_val_type, size_t var_val_len, u_char *statP, oid *name,
    size_t name_len)
{
    snmpa_t *t = &snmpa_task;
    char string[SNMP_DISPLAY_STRING_MAX_LEN+1];

    switch (action)
    {
    case SNMP_STATE_RESERVE1:
        return snmp_tc_display_string_validate(var_val, var_val_type,
            var_val_len);
        case SNMP_STATE_RESERVE2:
        case SNMP_STATE_FREE:
        case SNMP_STATE_ACTION:
        case SNMP_STATE_UNDO:
            break;
    case SNMP_STATE_COMMIT:
        buf2str(string, var_val, var_val_len);
        set_set_path_str(rg_conf, Swatchdog"/"Sboot_msg, string);
        if (t->cb.reconf_request_set)
            t->cb.reconf_request_set(0);
        break;
    }
    return SNMP_ERR_NOERROR;
}

As you can see, in the RESERVE1 stage, type and value validations are performed (lines 10 and 37). These validations check that the type of the given new value is the correct ASN.1 type of the MIB object, that the length of the new value is adequate, and that the value itself lies within the allowed range. The functions used for these validations (snmp_int_object_validate and snmp_tc_display_string_validate) are part of API functions for validating common SNMP types and textual-conventions of MIB objects. These API functions are found at <snmplib/snmp_var_valid.h>. In these examples, the stages besides COMMIT and RESERVE1 do not perform any operations, due to the simplicity of the examples. In the COMMIT stage, the variable's new value is extracted (lines 18 and 45), and written to the proper rg_conf field (lines 18, 46) using the set_set_path_XXX API.

Observe another important thing: lines 20 and 48 perform a call to a special callback of the snmp-agent task—the reconf_request_set callback. As you probably noted, this callback is called right after rg_conf has been altered. This callback signals the agent to call openrg_reconf() after successfully setting all variables in this PDU. This is needed in order to notify all appropriate Main Task tasks of the changes made to rg_conf.

### 26.5 Generating SNMP Traps

SNMP Traps are asynchronous messages sent by OpenRG's SNMP Agent to some Network Management System (NMS), in order to notify about the occurrence of events. The agent supports some standard SNMP traps such as coldStart, authenticationFailure, linkDown and linkUp.

#### 26.5.1 Trap Receiver Configuration

In order for the agent to be capable of sending SNMP Traps, a trap destination must be configured. Configuring a trap receiver is possible in one of the following ways:
• Via the agent's general configuration entries in `rg_conf`. These are: `snmp/traps_enabled`, `snmp/trap_version`, `snmp/trap_destination` and `snmp/trap_community`. These define a single trap receiver with common properties. For more information about these fields, refer to the Configuration Entries Guide.

• Using SNMP MIBs. By populating the `snmpNotifyTable`, `snmpTargetAddrTable`, `snmpTargetParamsTable` and `snmpCommunityTable` according to the rules of RFCs 3413 and 3584, it is possible to remotely configure trap destinations in the most versatile way.

• Using the given API for constructing entries into the above tables: `notify_entry_construct()`, `target_addr_table_add()`, `tgt_param_entry_construct()` and `comm_table_add()`.

26.5.2 Generating Standard Traps

Standard (generic) traps, such as coldStart, authenticationFailure, linkDown and linkUp, are generated by OpenRG's agent automatically, when the events defined for these notifications arise. For example, when one of OpenRG's network interfaces goes down, a linkDown trap will be sent (as long as there are trap receivers configured); when the SNMP agent receives a protocol message not properly authenticated, an authenticationFailure trap will be sent. In order to send the SNMP V1 "Enterprise-Specific" trap, you may use the following API:

```
send_easy_trap(int trap, int specific)
```

where `trap` should be `SNMP_TRAP_ENTERPRISESPECIFIC`, and `specific` should be the "specific-trap" code of your enterprise-specific trap.

26.6 SNMP Commands

The following section provides an example of managing OpenRG with SNMP, based on the Watchdog example (refer to Section 5.4).

SNMP commands enable you to configure the Watchdog module remotely and view its settings. When a command is executed, an SNMP application scans the Watchdog MIB definitions file (`WATCHDOG-MIB.mib`) for relevant MIB objects. You will find the `WATCHDOG-MIB.mib` file at `rg/pkg/samples/tutorial/snmp`. Some of the configuration data located in this file is imported from other MIB files, which are located at `pkg/ucd-snmp/mibs`. This is the standard location of all MIB files needed for configuring various OpenRG modules. Depending on the type of the executed command, you can retrieve values or assign them to the module's managed objects. The Watchdog MIB objects are:

- `watchDogEnabled`  A TruthValue object that controls whether the Watchdog feature is enabled or not. Its value can be 1 or 0.

- `watchDogMargin`  An Integer object specifying the Watchdog's timeframe (in seconds), during which it waits for the keep-alive signal. Its value range is 30–60.

- `watchDogEndMessage`  A DisplayString object specifying the message to be displayed when Watchdog reboots the system. Its value can be any letter string.
Three SNMP commands are used for sending Remote Procedure Calls (RPCs) to Watchdog:

**snmpset**  Sends a SET request to a network entity to assign a value to a certain parameter. The following is the template of the **snmpset** command executed on Watchdog:

```
snmpset -Os -v2c -m WATCHDOG-MIB -M <path to WATCHDOG-MIB.mib>:pkg/ucd-snmp/mibs/ -c private <OpenRG IP> <MIB object>.0 i (for Integer) or s (for DisplayString) <value>
```

Consider the command options:

- **-Os** Deletes all but the last symbolic part of the object ID. This option is used to display only the relevant part of the module's response.

- **-v2c** Specifies the protocol version (in this case, it is SNMPv2c).

- **-m** Specifies the MIB files that will be loaded for this application.

- **-M** Specifies the directories in which the MIB files are located.

- **-c** Specifies an SNMP community (in this case, it is `private`, enabling you to set values for the module parameters).

For example, let's assume that you left the `WATCHDOG-MIB.mib` file in its original location, that is at `rg/pkg/samples/tutorial/snmp`. If so, consider the syntax of the following **snmpset** command, which targets the **watchDogMargin** object.

```
rg$ snmpset -Os -v2c -m WATCHDOG-MIB -M pkg/samples/tutorial/snmp/:pkg/ucd-snmp/mibs/ -c private 192.168.1.1 watchDogMargin.0 i 31
```

By executing this command, you set the Watchdog margin to 31 seconds.

**snmpget**  Sends a GET request to a network entity to retrieve the current value of a certain parameter. The following is the template of the **snmpget** command executed on Watchdog:

```
snmpget -Os -v2c -m WATCHDOG-MIB -M <path to WATCHDOG-MIB.mib>:pkg/ucd-snmp/mibs/ -c private <OpenRG IP> <MIB object>.0
```

For example, consider the **snmpget** command that targets the **watchDogEndMessage** object:

```
rg$ snmpget -Os -v2c -m WATCHDOG-MIB -M pkg/samples/tutorial/snmp/:pkg/ucd-snmp/mibs/ -c private 192.168.1.1 watchDogEndMessage.0
```

After executing this command, the shell displays the message that Watchdog prints out on the screen before it reboots OpenRG.

**snmpwalk**  Sends GET NEXT requests to a network entity to retrieve the values of all configurable module parameters. The following is the template of the **snmpwalk** command executed on Watchdog:

```
snmpwalk -Os -v2c -m WATCHDOG-MIB -M <path to WATCHDOG-MIB.mib>:pkg/ucd-snmp/mibs/ -c private <OpenRG IP> watchDogMib
```

```
Note: The watchDogMib entry is the name of the Watchdog MIB objects group. This entry is defined in the WATCHDOG-MIB.mib file.
```

For example, consider the **snmpwalk** command that targets all Watchdog MIB objects:
After executing this command, the shell displays the values of all Watchdog managed objects.
27

Command Line Interface

The Command Line Interface (CLI) provides the means to start tasks and call functions from an interactive 'shell-like' prompt. It is used to enable on-the-fly access to a Main Task function using either a serial console or network connection, enabling the developer to:

- Display/set configuration parameters
- Start tasks and call functions
- Restore configuration defaults
- Load and burn an OpenRG image
- Configure network interfaces
- Reboot the system
- Access the Flash memory
- For a full list of the command categories, log into OpenRG's CLI and type help.

The CLI can be used to change internal OpenRG configuration settings. This functionality groups the CLI together with other management tasks such as Web-based management, Universal Plug & Play, and SNMP. Multiple CLI instances can exist simultaneously. A comprehensive list of CLI commands is provided in Section 27.2.

The CLI uses estream for input and output. Input and output do not occur while a command is being performed, since during that time the estream belongs to the command. All characters typed into the CLI during a command execution will be read and parsed after the command returns. The CLI should always return the returned value of the performed command, or an error message and "Return 1" on parsing/usage error.
27.1 Accessing the CLI

The CLI can be accessed in two ways:

1. Serial console
   a. Connect a serial cable between your PC and gateway.
   b. Establish communication with OpenRG using a serial console application. Refer to your Installation Guide for hardware specific communication parameters.

2. Network connection
   a. Connect your PC to the gateway.
   b. Establish communication with OpenRG using Telnet. The login name and password are the same as in the Web-based management—the defaults are both "". For example:

   ```
   $ telnet 192.168.1.1
   Trying 192.168.1.1...
   Connected to openrg (192.168.1.1).
   Username: 
   Password: *****
   OpenRG>
   ```

27.2 CLI Commands

The templates in which commands are written in this document conform with the following standard Linux syntax conventions:

• Diamond brackets (<> ) – Contain a description of a mandatory parameter to be written in its place.

• Brackets ([ ] ) – Indicate that their content is optional.

• Braces ( { } ) – Delimit multiple options, which are separated by vertical bars (|).

All CLI commands are divided into categories, which can be viewed by typing `help` in the CLI prompt. Use the `help` command as follows:

- **help all** Display all available commands.

- **help [category [subcategory]...] <category>** Display a description of the category along with all its commands. For example, type: "help conf factory" in the root shell. This is equivalent to entering the 'factory' category and typing "help all".

- **help [category [subcategory]...] <command>** Display detailed help for a specific command—both its description and usage. For example, type: "help conf factory print".

- **help -s <string>** Search for all commands and categories containing the string, along with their descriptions (the search goes from the current level down).
In addition to the **help** command, the following commands can be used in any shell level—the root as well as any category.

**exit**  Exit from the current session. In the root, this command will terminate the CLI session. In a category, it will go up one level.

**hide**  Conceal commands that are defined as hidden.

**unhide**  Make hidden commands available.

The following is an alphabetical list and specification of the CLI commands, organized according to the CLI command categories available on OpenRG.

### Important Notes:

- Commands representing their respective CLI categories appear in titles of the following sections. In OpenRG's CLI prompt, all commands appear in lower case, and they are case-sensitive.

- Additional commands may be available in the CLI that do not appear in this manual. These commands are for internal debugging purposes only, and therefore they have not been documented here.

#### 27.2.1 adm6996

The following commands are used for hardware switch debugging on the *Danube* platform.

- **vlan_print**  Display the switch VLAN table information.
- **igmp_print**  Display the switch IGMP table information.

#### 27.2.2 bridge

The following commands are used for configuring OpenRG's bridge.

- **config** `{addif | delif} <bridge name> <device> [device]...`  Configure bridge: add or delete device interface(s).
- **connection add** `<device> [device]... | delete <bridge name>`  Create or delete an Ethernet bridge over one or more devices.
- **info**  Display bridge information.
- **mac** `{add | delete} <bridge> <mac> [dev]`  Add or delete MAC mapping.

#### 27.2.3 conf

The following commands are used for reading and writing OpenRG's configuration data (refer to Chapter 18).
compress Compress and encode in base64 configuration.

uncompress <string> Restore configuration from compressed and encoded in base64 string.

del <path> Delete a sub-tree from OpenRG’s configuration, starting at the indicated path.

print <path> | / Print OpenRG’s configuration either from the path indicated or from the root (the entire configuration).

priv_set <path> <value> Set private configuration – set path to value.

priv_print <path> Print private configuration. Use rg_conf_priv_print to print entire configuration.

priv_del <path> Delete subtree (starting at <path>) from private configuration.

ram_print <path> | / Print OpenRG’s dynamic configuration either from the path indicated or from the root (the entire configuration).

ram_set <path> <value> Set OpenRG’s dynamic configuration path to a value.

ram_del <path> Delete OpenRG’s dynamic configuration starting at <path>.

reconf <flash delay> Reconfigure the system according to latest component changes and store this configuration in the Flash. The time of storage is determined by the Flash delay flag —an integer in the range of 1 (Now) to 4 (Low).

set <path> <value> Set OpenRG’s configuration path to a value. For settings to take effect you must issue the reconf command. An example for disabling the rtl0 device:

```
OpenRG> conf
cnf> set dev/rtl0/enabled 0
Returned 1074241648
cnf> print dev/rtl0/enabled
(enabled(0))
Returned 0
```

set_merge <path> <set_str> Merge configuration set to path.

set_obscure <path> <value> Set OpenRG’s configuration path to an obscured value.

set_permission <path> <permission> <value> Set permission configuration path value.

print_permission <path> Print permission configuration for path.

27.2.3.1 factory

The following commands are used for controlling OpenRG’s factory settings.

close Load factory settings back to the factory configuration sector.

del [<path>] Delete a sub-tree from OpenRG’s factory configuration, starting at the indicated path.

open Load the settings from the factory configuration sector.
**print** `<path> | /` Print the factory configuration either from the path indicated or from the root.

**restore** [-d] [-a] Restore the default factory configuration settings. The -d option is for restoring defaults without rebooting the system. The -a option is for removing persistent entries.

**set** `<path> <value>` Set the factory configuration path to value.

**set_escaped** `<path> <value>` Set the factory configuration path to a value that is given as a string with encoded special characters.

### 27.2.4 config

The following commands are used for configuring devices.

**config l2tps enable** [from `<from ip> to `<to ip>`] | disable Configure L2TP server: enable within an IP range or disable.

**config pptps enable** [from `<from ip> to `<to ip>`] | disable Configure PPTP server: enable within an IP range or disable.

### 27.2.5 connection

The following commands are used for managing general connections.

**l2tp_vpn add** `<server> <login name> <password> <shared secret> | delete `<connection name>` Create or delete a VPN WAN connection to an L2TP server.

**pppoe add** `<login name> <password> dev `<underlying device>` Configure a PPPoE interface.

**pptp_vpn add** `<server> <login name> <password> | delete `<connection name>` Create or delete a VPN WAN connection to a PPTP server.

**vlan add** `<underlying device> <vlan id> | delete `<vlan device>` Create or delete a VLAN device.

### 27.2.6 crash

The following commands are used for saving and viewing serial logs and crashes.

**status** Indicate if a new crash log is available.

**timestamps** Print a list of previous serial and crash logs timestamps (up to four logs are saved).

**log** `<log num>` View the log of a previous crash. Indicate the number of the desired log (0-3) – 0 - latest log, 3 - oldest log.
save_buffer_now  Save the current serial log to the flash.

send_mail_now  Send the current serial log by email, to the address defined in rg_conf. Refer to the explanation below to set up this address.

print_current_buffer  Show the current serial log on the screen.

erase_buffer  Erase the current serial log buffer from the RAM.

### 27.2.7 cwmp

The following commands are used for CWMP session control.

get_params <parameter name>  Display the CWMP parameter value.

session_start  Start CWMP session to ACS.

session_stop  Stop CWMP session.

set_params <parameter name> <value>  Set a CWMP parameter value. Indicates success or failure (and reason).

status [-v]  Display CWMP status. The -v option is for an additional detailed display of pending requests and responses.

### 27.2.8 dev

ioctl_print <dev_name> DEV_IF_<ioctl name>  Print dev_if ioctl result.

mii_reg_get <dev_name> <register>  Display the Ethernet MII register value. Enter the device name and MII register number.

mii_reg_set <dev_name> <register> <value>  Set the Ethernet MII register value. Enter the device name, MII register number, and the value that you want to set.

mii_phy_reg_get <dev_name> <phy_id> <register>  Display the Ethernet MII register value. Enter the device name, the physical hardware number, and the MII register number.

mii_phy_reg_set <dev_name> <phy_id> <register> <value>  Set the Ethernet MII register value. Enter the device name, the physical hardware number, the MII register number, and the value that you want to set.

stats <dev_name>  Print device statistic.

### 27.2.8.1 ar8316

The following are ar8316 hardware switch debugging commands.

reg_get <dev_name> <register>  Display AR8316 register value.

reg_set <dev_name> <register> <value>  Set AR8316 register value.
vlan_dump <dev_name>  Display AR8316 VLAN table.

arl_dump <dev_name>  Display AR8316 arl table.

regs_dump <dev_name>  Display AR8316 registers.

mirror <dev_name> <mirror_port> <port_to_mirror>  
<mode:0=off,1=ingress,2=egress,3=all>  Mirror AR8316 port.

27.2.9 docsis

The following are DOCSIS related commands.

certificate  Displays the Docsis certificate.

cli  Display the Docsis subsystem (by TI) that has its own CLI.

cm_freq_plan_get  Display the cable modem frequency plan.

cm_freq_plan_set <us/euro/japan/hybrid>  Configure the cable modem frequency plan.

full_scan_restarted_notify  Notify that a cable downstream frequency full scan was finished without achieving synchronization on any of the frequencies.

golden_freq_get  Display the list of golden frequencies.

golden_freq_set <freq_1> <freq_2> ... <freq_n>  Configure the list of golden frequencies (in Hz) that the cable modem will try to synchronize before scanning all the frequencies.

golden_list_scan_ended_notify  Notify that the golden list scan has ended.

golden_list_scan_started_notify  Notify that the golden list scan has started.

info  Provides Docsis information.

pacm_config_get  Display PACM configuration.

pacm_config_set <mode> <prov_flows> <file_hash_bypass>  Configure PACM configuration: mode – us/euro, prov_flows (provision flows) – secure/hybrid/basic, file_hash_bypass – 0/1.

reset_mta  Resets MTA.

reset_cable_modem  Resets the cable modem.

27.2.10 factory

The following are manufacturing factory related commands.

generate [<base MAC> <serial number>]  Generate factory settings from a random MAC address or as specified in a parameter.
**print_label**  Print manufacturing label.

**print_configuration**  Print configuration parameters.

### 27.2.11 fastpath

The following are Fastpath (PSE) related commands.

**verbose**  Enable/disable fastpath verbose debugging.

**info**  Display Fastpath information.

### 27.2.12 firewall

The following commands are used for controlling and displaying firewall and NAT data (refer to Chapter 30).

```
dump [-v] [-ah|-ai] [-c <chain> | -h <address> | -i <index>] [-pr|-ps|-ph|-pm|-pd|-pp]
```

Display information on current state of the firewall. If no options are passed, all available information is displayed. Command options are:

- **-v** Displays the connection virtual machine.
- **-a** Displays advanced information. Command options are:
  - **-ah** Displays memory address of each entry.
  - **-ai** Displays index of each entry.
- **-c** Displays the opcodes from chain number `chain`.
- **-h** Displays the opcodes from address `address`.
- **-i** Displays the opcodes from index `index`.
- **-p** Displays partial information only. Command options are:
  - **-pr** Displays the virtual machine opcodes.
  - **-ps** Displays the list of active connections.
  - **-pn** Displays the numbers of active connections.
  - **-ph** Displays the list of host mappings.
  - **-pm** Displays the list of LAN MAC addresses mappings.
  - **-pd** Displays the list of device on which the firewall hooks are active.
- **pp** Displays the list of firewall reserved ports.

**fastpath**  Turn the firewall Fastpath feature on/off (default is on).

**filter <mode>**  Mode 0 turns firewall packet inspection off, leaving NAT only. Mode 1 turns on both NAT and firewall.

**mac_cache_dump**  Dump the firewall's cached data of MAC/IP association.

**restart**  Stop and start the firewall and NAT. Equivalent to `firewall stop` and `firewall start`.

**start**  Activate the firewall and NAT.

**stop**  Deactivate the firewall and NAT.

**trace <mode>**  Changes the firewall tracing mode. Available modes are:

- **0** – Off
- **1** – Full
- **2** – Partial

**variable [-a] [-i <index> | -n <name>]**  Displays a list of available rule variables and their hit counters. Command options are:

- **-a** Display active variables only.
- **-i** Display only the variable with index `<index>`.
- **-n** Display all variables that `<name>` is a substring of their names.

### 27.2.13 firmware_update

The following commands are used for remotely upgrading OpenRG.

**start -u <url> [-c][-i]**  Remotely upgrade OpenRG from a URL (both tftp:// and http:// are supported). Command options are:

- **-c** Check for availability only—do not burn to the Flash.
- **-i** Ignore version number when deciding whether to burn the image.

**cancel**  Terminate a remote upgrade process.

### 27.2.14 flash

The following commands are used for manipulation of the flash and loader (refer to Chapter 6).
**active_image_name**  Print the name of the active image.

**boot [-g] [-s <section> | -r <address>]**  Boot the system. Command options are:

- `-g` Boot with kgdb.
- `-s` The Flash section from which to boot.
- `-r` The Flash address from which to boot.

**bset**  Configure the bootloader. This command must be issued during boot sequence.

**commit**  Save the current configuration to permanent storage.

**cramfs_location**  Print the cramfs (mainfs) location.

**dump [-s <section> | -r <address>] [-l <length>] [-1|-2|-4]**  Dump the Flash content. Command options are:

- `-s` Begin dump from the stated section (default is 0).
- `-r` Begin dump from the stated address (default is 0).
- `-l` The length of the content to dump (in bytes).
- `-1` Display dump in bytes.
- `-2` Display dump in words.
- `-4` Display dump in double words.

**erase [-d] <section>**  Delete the contents of the flash section number stated. The `-d` option is for deleting section data as well, and initializing it with 0xFF.

**layout**  Display the current layout of the Flash.

**load -u <url> [-s <section> | -r <address>]**  Load an image from a URL (both tftp:// and http:// are supported) to the board's flash. The default location is the oldest section of type "IMAGE". Alternatively, you can specify a location, using the following command options:

- `-s` Load the image to the specified section of the flash (recommended).
- `-r` Load the image to the specified physical address in the Flash, disregarding section boundaries.

Caution: Loading an image to the beginning of the flash with the "load -u <url> -r 0" command overwrites the original bootloader. You must therefore ensure that the new image contains a bootloader to avoid loss of booting capability.

**lock <address> <length>**  Lock the MTD region from the specified address and in the specified length.
unlock <address> <length>  Unlock the MTD region.

27.2.15 inet_connection

The following commands are used for managing Internet connections.

ether dynamic_ip [dev <underlying device>]  Configure an Ethernet Internet connection.

l2tp <server> <login name> <password> [dev <underlying device>]  Create a VPN WAN connection to an L2TP server.

pppoe <login name> <password>  Configure a PPPoE Internet connection.

pptp <server> <login name> <password> [dev <underlying device>]  Create a VPN WAN connection to a PPTP server.

27.2.16 jnet

The following commands are used for managing OpenRG's Jnet client:

status  Display the Jnet client's status (active/inactive).

start  Run the Jnet client.

stop  Stop the Jnet client. Note that OpenRG will disconnect from JRMS.

restart  Restart the Jnet client.

enable_interceptions <1 | 0>  Start/stop intercepting the gateway's disconnections from JRMS (enter 1 for enabling the interception and 0—for disabling).

conn_req_url  Display the Jnet connection request URL.

rmt_upd_start  Stop Jnet client and start remote upgrade.

27.2.17 kernel

The following commands are used for kernel and CPU related actions.

cpu_load_avg  Display the board's average CPU load.

cpu_load_on <number of seconds>  Display the board's CPU usage in percentage, refreshing every number of seconds specified.

cpu_load_off  Stop the board's CPU usage display.

skb_cache_stat  Display SKB cache statistics (Freescale mpc8349itx platform only).

meminfo  Display memory information.
sys_ioctl <kos character device type> <command> <argument> Issue an OpenRG IOCTL. The list of available device types is located in pkg/kos/kos_chardev_id.h.

top Display the board’s top CPU-consuming processes and their respective virtual memory size and data usage details.

27.2.18 log

The following commands are used for controlling OpenRG’s logging behavior.

print [varlog | fw] [component1.severity1][component2.severity2]... Display all contents of log file or firewall log. Command options are:

- varlog Display complete system log (this is the default when typing "log print").
- fw Display firewall log.
- [component1.severity1][component2.severity2]... Filter specific lines from the log file. For example, "log print varlog http.err" will display messages for the http entity, with severity err (error) and up.

clear <varlog | fw> Erase the specified buffer.

filter {set | print | reset} Perform filtering actions on the real-time log output. Command options are:

- set <component1.severity1> {[component2.severity2]}... Filter log output, equal to or higher than the severity level, to the CLI. For example, "log filter set *.debug http.err" will display messages for all components with severity level of debug and up, except for http, for which messages with level err and up will be displayed.
- print {[all]} Print the currently active filters. Using the option "all" prints all components and their current severity level.
- reset Deletes all defined filters.

statistics [all | <category> | clear |clear <category>] Manage statistics – select whether to view all or a specific category.

27.2.19 misc

The following are miscellaneous commands for controlling various functions in OpenRG.

crash [kernet | openrg | panic] Crash the system (to test debugging-tools).

diskplug Disk plug in/out emulation (UML only).

disks Display all connected disks and file shares.

eknet_hooks_dump Dump to the console which knet_hooks run on each device.
leak [kernet | openrg]  Create a deliberate memory leak.

lpd_spool_info  Display printer spool diagnostics.

lpq  Display print server status.

pppos_start <serial device name>  Initiate a PPP over Serial (PPPoS) connection to the device specified.

pppos_close <network device name>  Close the specified PPPoS connection.

print_ram  Print RAM consumption for each process.

radius_stats  Display RADIUS packet statistics.

ralink_set <dev_name> <string_name> <integer_value>  RALink RT2880 private set ioctl. For example: ralink_set ra0/ra1 BasicRate 3.

vlan_add <device name> <vlan id>  Add a VLAN interface by entering its name and ID.

top <{start | stop}>  Profiling over event loop and estream. Use Start/Stop to control profiling. The Stop option also prints the output. If you would like to receive a more detailed output than provided by default, compile OpenRG with the CONFIG_RG_MT_PROFILING_FULL_INFO flag enabled.

vlan_add <device name> <vlan id>  Add a VLAN interface by entering its name and ID.

27.2.19.1  serial

mute <reboot>  Mute the serial interface. Use reboot if mute status was actually changed.

unmute <reboot>  Un-mute the serial interface. Use reboot if mute status was actually changed.

27.2.19.2  telnet

allow <mode>  Allow telnet access from the LAN. Mode: 0 – Apply until next reboot only. 1 – Apply and save.

27.2.20  net

The following commands are used for configuring different network parameters.

dns_route  The following commands are used for dynamic routing according to DNS replies.

Note: The dynamic routes are automatically created only if OpenRG's 'Domain Routing' feature is enabled. For more information, refer to the 'Routing Protocols' section of the OpenRG User Manual, and to the 'Domain Routing' section of the Configuration Entries Guide.
• **print** Print dynamic routes.

• **del** Delete dynamic routes.

**igmp** The following commands are used for handling OpenRG's IGMP subscription.

• **status** Print the LAN device's IGMP subscription status.

• **reset** Clear all IGMP subscriptions of the LAN device. Running this command causes OpenRG to issue an IGMP query to the LAN and rebuild its IGMP database.

**host <name>** Resolve the IP addresses of a host by its name.

**ifconfig [-v] [dev] [ip]** Configure network interfaces. Command options are:

• **-v** Present more details.

• **dev** The device name.

• **ip** The address given to the device.

**intercept_state** Prints the interception state.

**ipsec_dbg_conn <initiate | wait>** Initiate an IPSec connection or wait for incoming connections. This command displays debugging information on the console.

**main_wan** Print the name of the current main WAN device.

**mii_reg_get <dev_name> <register>** Get an Ethernet MII register value.

**ping <ip>** Test network connectivity.

**rg_ifconfig <details level>** List OpenRG's network devices.

**route** Print the route table.

**subnet_set <dev_name> <0-255 - subnet>** Set device's subnet to 192.168.X.0.

**toggle_lan_traffic <1 | 0> <timeout in seconds>** Enable/Disable all LAN traffic.

### 27.2.20.1 protected_setup

**push_button** Start protected setup registration by push button.

**pin_code <pin_code (8 digits)>** Start protected setup registration by pin code.

### 27.2.21 psb6973

The following commands are used for hardware switch debugging on the Twinpass platform.
vlan_print  Display the switch VLAN table information.

reg_get <dev_name> <register>  Display the register value. Enter the device name and the register number.

reg_set <dev_name> <register> <value>  Set the register value. Enter the device name, the register number, and the value that you want to set.

27.2.22  pvc

The following commands are used for performing PVC scans.

scan_restart  Stop and restart PVC scan.

scan_status  Display PVC scan status.

27.2.23  qos

utilization  Display connection utilization information.

27.2.24  scr

27.2.24.1  log

print  Print the SCR log.

clear  Clear the SCR log.

27.2.25  service

27.2.25.1  ssh

The following commands are used for Secure Shells (SSH).

start  Activate SSH daemon.

status  Print status of SSH daemon.

stop  Stop SSH daemon.

27.2.26  switch

The following commands are used for hardware switch.
info  Print switch and bridge information.

reset_vlan <device name>  Clear all the VLANs.

mac_info <device name> <mac>  Get the port used for connecting to this MAC, from the MAC table.

27.2.26.1 stp

status  Display the status of STP and ports.

start <device name>  Starts the given STP.

stop <device name>  Stops the given STP.

port_state <device name> <port id>  Get stp port state.

27.2.26.2 port

info <device name> <port id>  Gives information of the port of given device.

enable <device name> <port id> <enable[0|1]>  Enable / Disable a specific port.

add <addif | delif> <bridge name> <device> [device]...  Add/Delete a specific port to/from bridge.

status <device name> <port id>  Gives status of a port of given device.

conf <device name> <port> <duplex> <speed> <enable[0|1]>  Configure port.

set_pvid <device name> <port> <pvid>  Set default VLAN ID.

set_vid <device name> <port> <vid> <egress policy [0|1]>  Assign VLAN ID to specific port.

ports_isolation <device name> <port 1> <port 2> ... <port n>  Enable the ports isolation.

27.2.27 system

The following commands perform system-related tasks.

cat <file>  Print file contents to console.

date  Print the current UTC and local time.

delayed_reboot  Reboot the system asynchronously.

die <exit status>  Exit from OpenRG returning the indicated exit status.

echo  Echo arguments to console.
entity_close <entity pointer>  Close an entity by providing a pointer to one. The pointer to a task can be retrieved by executing the `ps` command.

etask_list_dump  Display back trace of all etasks.

exec <command> [ <arguments...> ]  Execute a program with optional arguments.

exit_and_shell  Exit from OpenRG and open a shell on the serial console.

http_intercept_status  Displays the HTTP intercept status.

ps  Display detailed task information.

print_config [ [-s] CONFIG ]  Print compilation configuration. Search for option if specified. Examples:
print_config – Print full configuration.
print_config DDNS – Print configuration values containing DDNS as a substring.
print_config -s CONFIG_RG_DDNS – Print configuration value of CONFIG_RG_DDNS only.

reboot  Reboot the system.

restore_factory_settings [ -d ] [ -a ] [ -f ]  Restore the default factory configuration settings. The -d option is for restoring defaults without rebooting the system. The -a option is for removing persistent entries. The -f option is for deleting the configuration sections.

shell  Spawn a BusyBox shell in foreground. Type 'exit' to return from the shell to the CLI.

ver  Display version information.

### 27.2.27.1 todc

The following commands are used to update the todc task from the Docsis serial console.

shift_time  Shift the system time. Input: shift in seconds.

notify_failure  Notify OpenRG that the docsis todc failed.

### 27.2.28 upnp

The following commands are used for UPnP device handling.

status  Display UPnP device model details.

tree_rebuild  Build and advertize the UPnP device model.

### 27.2.28.1 av

status  Display the UPnP AV status.
**rescan** `<root_dir>` `<rebuild>`  Rescan the root directory. For `root_dir` enter the partition name, e.g. "A", and for `rebuild` enter 0 for a regular scan or 1 for a full rebuild.

### 27.2.28.2 igd

**status**  Display the IGD status.

**tree_rebuild**  Rebuild and advertize the IGD device tree.

### 27.2.28.3 sniffer

The following commands are used for UPnP environment debugging, and only available when configuring OpenRG with `CONFIG_RG_UPNP_SNIFTER`.

**status**  Display the UPnP sniffer status.

**rescan**  Perform a rescan on all devices.

**reset**  Clear detected devices on all devices.

### 27.2.29 wbm

**border_set** `<{0 | 1}>`  Stop and start WBM border mode.

**debug_set** `<0 | 1>`  Stop and start WBM debug mode, by setting 0 or 1.

**session_release_all**  Release all existing WBM sessions.

**themes**  Print a list of available WBM themes.

### 27.2.30 wireless

The following commands are used for handling the wireless connection.

**channels** `<device name>`  Show channels supported by a given wireless device.

**get_main**  Display the main (i.e. physical) wireless device name.

**get_default_password**  Display the default global wireless password (unobscured).

**global_password_supported**  Indicate whether the board supports a global wireless password (0=no, 1=yes).

### 27.2.30.1 security

**status**  Check if wireless device security is enabled.

**set** `<on | off>`  Set wireless device security on/off.
change_password <password>  Change the password of the secured wireless network.

27.2.30.2 ssid

get  Display the SSID of the main wireless device.

set <SSID>  Set the SSID of the main wireless device.

27.2.30.3 captive

check  Check if a wireless captive portal is available.

create <SSID> <URL>  Create a wireless captive portal with a specific SSID. Enter the SSID and the portal's URL.

destroy <SSID>  Destroy a wireless captive portal with a specific SSID.

auth <SSID> <MAC> <1-add 0-remove>  Add or remove authorization for a wireless client by MAC. Enter the SSID, the MAC address, and 1 or 0 to add or remove, respectively.

qos <SSID> <downstream> <upstream>  Configure QoS for the wireless portal. Enter the SSID and downstream and upstream values (in Kbps, -1 for unlimited).

27.3 Adding a New Command Line Function

The CLI receives a list of commands from Main Task when the CLI is opened. The list of commands is dynamic, and commands can be added/removed from the list on-the-fly. In order to add a command you need to perform the following steps:

1. Write functions that the CLI will call.

2. Define a cmd_item_t array with the commands, or add your commands to an existing array.

3. Define a category_t for your commands (or use an existing category).

4. Register the command.

The following sections cover these steps one by one, and then give an example of adding two commands to the CLI.

27.3.1 Writing Functions

Since a command is translated by the CLI to a function call, you must first write the functions that you would like to be accessible from the CLI.
27.3.2 Defining a Command Array

This is the prototype for a single CLI command:

```c
typedef struct {
    char *name; /* Name of the CLI command - this is what the user
                 * types in order to execute the function. */
    char *desc; /* One line description. "\n" is not allowed */
    char *desc_long; /* Detailed description. "\n" is allowed. Must
                      * be terminated by "\n" */
    void *function; /* Typecast anyway */
    cmd_param_t param_types[CMD_MAX_PARAM_COUNT]; /* parameters types,
                      * ended by
        * CMD_PARAM_END */
    u32 flags; /* One of CMD_ITEM_... */
} cmd_item_t;
```

The flags for the commands:

```c
#define CMD_ITEM_ASYNC (1<<0) /* Command is asynchronous.
                       * Note: async commands are always
                       * treated as void! */
#define CMD_ITEM_HIDDEN (1<<1) /* Command is hidden - will not be shown
                           * by help or tab unless in "unhide" */
#define CMD_ITEM_NO_STDIN (1<<2) /* On commands that have std_out,
                                * std_err and std_in (basically only
                                * the exec/shell commands) set
                                * std_in to /dev/null */
#define CMD_ITEM_RET_TYPE_VOID (1<<3) /* Command doesn't return
                                   * a value */
```

The argument types for the commands:

```c
typedef enum {
    CMD_PARAM_END = 0, /* Must be zero - don't change */
    CMD_PARAM_STR = 1,
    CMD_PARAM_ARGV = 2,
    CMD_PARAM_ARGC = 3,
    CMD_PARAM_INT = 4,
    CMD_PARAM_ESTREAM = 5,
    CMD_PARAM_CMD = 6,
} cmd_param_t;
```

27.3.3 Defining a Category

All CLI commands are part of categories. You can either define a new category or use an existing one. This is the category structure:

```c
/* Category for a group of commands. Examples can be Firewall Category:
   * .prefix = "firewall", .help = "Firewall related commands" */
typedef struct {
    char *prefix; /* Category name */
    char *desc; /* Category description - one line, no "\n" */
    cmd_active_t active_cb; /* if NULL all commands are assumed
                               * active */
} category_t;
```

27.3.4 Registering Commands

Registering a command array is done using a cli.h API function:

```c
/* Register commands to a sub-menu:
   * head - List where the sub-menu should be (it is added in case it doesn't exist).
   * category - The category of new sub-menu.
   * commands - The commands to be added to the sub-menu.
   * return - The sub-menu's list, which can be used to create sub-sub-menus.
```
The category represents a sub-menu in the CLI under the "head" menu. If there is no such sub-menu, one will be created by the function. The commands are added to the sub-menu. The return value is the lower level list, and can be used to add commands to a sub-sub-menu.

For example, in order to maintain the existing structure:

```
conf sub-menu
  <conf commands...>
  factory sub-sub-menu
    <factory commands...>
```

perform:

```
conf_list = multilevel_cli_add_commands(&openrg_commands, &conf_category, conf_commands);
```

This will add the `conf` sub-menu to the top level of the CLI, and the `conf_commands` to the `conf` sub-menu.

Next, perform:

```
multilevel_cli_add_commands(conf_list, &factory_category, factory_commands);
```

This will add the factory sub-sub-menu to the `conf` sub-menu, and the `factory_commands` to the factory sub-sub-menu. If you would like to add commands to the top level, use the `cmd_register` function as described in `pkg/util/cmd_t.h`.

### 27.3.5 Removing Commands

To unregister a command, perform the following:

```
/* Unregister commands from a sub-menu: */
* head - List where the sub-menu should be found.
* category - The category of the sub-menu.
* commands - The commands to be removed from the sub-menu.
*/
void multilevel_cli_remove_commands(cmd_list_t **head, category_t *category, cmd_item_t *commands);
```

The multi-level removal function removes the specified commands. Should all commands be removed from a level, this function will also unregister the level itself.

### 27.3.6 Example – Adding Commands

Let's assume that you have added a VoIP module to OpenRG, and wish to be able to enable or disable verbose output, and to add a dump function for the module data. Basically, you would like to enable CLI access to two functions:

```
void voip_verbose(int enable);
int voip_dump(estream_t *estream, int argc, char *argv[]);
```

You will need to perform the following three steps:

1. Define the `cmd_item_t` array:

```
static cmd_item_t voip_commands[] = {
```
2. Define a category (or use an existing one):

```c
static category_t voip_cmd_category =
{
    .prefix = "voip",
    .desc = "voip commands",
};
```

3. Register the commands. When your task/module is loaded, call the following function in order to add your commands to the CLI:

```c
multilevel_cli_add_commands(&openrg_commands, &voip_cmd_category,
   voip_commands);
```

The outcome will be a new "voip" sub-menu.

### 27.3.7 Notes

1. If the CLI function uses `estream` for input and output, and is synchronous, then `CMD_PARAM_ESTREAM` must be its first argument.

2. Asynchronous Commands:

   - Must use `CMD_PARAM_CMD` as their first argument, which means that their associated function must receive a `cmd_t *` as the first argument.
   - Should use `cmd_retval_set()` in order to return values to the CLI.
   - Must call `cmd_return()` when they terminate.
   - May issue push on cmd estream, in order to receive notifications when cmd->estream is closed.

3. An example of a full implementation of CLI commands is available in the Development Tutorial, Chapter 5.
28

Web-based Management

28.1 Overview

OpenRG includes a graphically intuitive Web-based Management (WBM) that provides comprehensive controlling and monitoring over the gateway’s features. The interactive graphical network map illustrates the current network topology and firewall setting, with LAN device and connection configuration screens easily accessible by clicking on their descriptive icons.

Figure 28.1 The Network Map

28.1.1 High Usability

The WBM is designed to assist the novice user when confronted with complex networking tasks such as setting up a VPN connection. The wizard guides the user through a series of
selection choices, collects all the necessary information, and creates the network connection. The wizard minimizes the steps involved in creating network connections by intelligently applying values to connection parameters that would otherwise require a networking proficient user. The VPN client wizard is such an example.

![VPN Client or Point-To-Point Wizard](image)

### 28.1.2 Customization

The WBM is designed to allow easy adjusting to specific requirements. For example, replacing corporate logos and icons is a simple task.

![Web-based Management Logo Customization](image)

Another example, is the WBM’s multilingual interface. OpenRG’s text is stored independently from the source code, enabling straightforward interface internationalization. Changing the Graphic User Interface (GUI) language can be performed in run-time. For more information, refer to Section 28.6.
28.1.3  Flexibility

OpenRG’s flexibility of dynamic content is achieved through the use of Common Gateway Interface (CGI) technology. Using CGI, the gateway generates HTML pages in run-time (unlike static HTML-based technology). The CGI technology offers maximum flexibility in terms of dynamic content, meaning that the processing is performed on the server side (OpenRG). Another advantage is the reduction in dependency on Javascript code, which is browser-dependent.

28.2  Directory Structure

Figure Figure 28.5 depicts OpenRG’s WBM directory structure.
The main WBM directories are located in `pkg/web_mng`. This directory holds the following subdirectories:

- **lib** - The WBM library, which is the WBM engine. This library can be used not only for OpenRG, but for other Web Server based products.

- **cgi** - OpenRG's general pages. These are component-less pages, such as 'Quick Setup' and 'Installation Wizard' pages. Unlike the 'lib' directory, the 'cgi' directory is specific to OpenRG.

- **images** - holds all WBM images (gif, jpeg, png).

Additional WBM pages reside in subdirectories across the source code tree, in their respective component directory. For example, print server WBM pages reside in `pkg/print_server/wbm`.

### 28.3 General Structure and Flow

This section will familiarize you with the Web-based management operation, depicting the HTTP session that is initiated by the user's browser and the main functions involved. In addition, you will learn about the WBM's initialization process.

#### 28.3.1 The HTTP Connection Life Cycle

*Figure 28.6* depicts the WBM's design:
Figure 28.6 Web-based Management Design

When the user initially browses to the WBM, the browser sends HTTP GET to OpenRG. When the user submits a page by clicking one of its buttons, the HTML objects in the page (such as drop-down menus, checkboxes, textboxes etc.) are posted over HTTP to OpenRG as HTTP POST data. OpenRG's HTTP server listens to incoming connections on port 80, and when it detects such an incoming connection, it invokes the WBM CGI handler. At this point, the scan stage starts, and the posted data is parsed by the WBM CGI.

Assuming no errors are detected, the scan function writes the input data to \texttt{rg\_conf}, and then reconfigures all relevant entities so that the changes will take effect. As the next step, the scan function decides what is the next page to be displayed, and sends back HTTP 302 code (Redirect) to the browser. In response, the browser sends a GET request to OpenRG. At this stage, the corresponding print function is called. The print function generates the new HTML page and writes it back to the HTTP server connection with the browser.

The new HTML page may reflect the last updates in OpenRG's configuration. OpenRG works according to the "POST-Redirect-GET" method. According to this method, POSTS requests coming from the browser should not be replied by HTTP 200 code with the result page, rather should only be processed and then a Redirect is sent back to the browser. The browser, in turn, sends a GET request to OpenRG, and only then OpenRG should reply by HTTP 200 code with the result page.

Figure 28.7 Request and Session Flow

**Request** A combination of a browser's POST/GET and OpenRG's response is also referred as a "Request". Each request is created when data arrives from the HTTP server and disappears right after the response is sent back to the client.

**Session** A series of requests. Begins when the user initially browses to OpenRG (login page), and ends when the the user logs out or if an idle timeout occurs. Each user that browses to the WBM is engaged in a separate session. The WBM is designed to store up to one hundred simultaneous sessions. This number is also configurable.
Scan  The process of extracting the posted variable values, validating and processing them, and storing the data in the configuration file.

Print  The process of generating an output HTML response, using updated data from the configuration file.

## 28.3.2 Main Function

When an incoming connection arrives to one of OpenRG's management ports, the HTTP server calls `wbm_cgi()`. `wbm_cgi()` is the main dispatcher. It creates the request data structure `g_request`, authenticates the session, analyzes some of the posted data (such as clicked buttons) and either calls a specific page scan function or invokes a generic button handler.

`wbm_cgi()` is actually the WBM CGI entry point—it is registered as a callback in the HTTP server upon the WBM initialization stage. Here are the main stages that occur in `wbm_cgi()`:

1. A request data structure `request_t` is created and initialized. `g_request` is a global variable that reflects the request that is currently being processed. Since only one request is handled at a given moment, there is no need to hold more than one global variable. Upon clicking one of the buttons in the page displayed on the browser side, `g_request` is created. After handling the POST request in OpenRG, a Redirect is returned to the browser, so `g_request` is freed. But when the browser sends OpenRG a GET request in response, `g_request` is created once again, and freed after the result page is sent back to the browser.

   Although `g_request` is created twice because of the POST-Redirect-GET method, it can be thought of as one request, as the previous diagram shows. `g_request` holds a pointer to an associated session. When a request is created, it is associated with one of the stored sessions, according to a session ID that is passed as a cookie (if the user initially browses to OpenRG, then there is no such ID in the HTTP GET, so a new session is created and associated with `g_request`, and the new session ID cookie is stored in the browser). The initial state of the new session is unauthenticated. This means that most of the WBM pages will be unaccessible unless the user logs in, thus authenticating the session.

2. Some of the posted data is processed, using the `request_entries_analyze()` function. This function processes data that exists in every page, such as the current page identifier and the clicked button. The extracted information is stored in `g_request` to allow simple access later.

3. If one of the posted variables has a special handling routine, called a variable handler (var_handler), then this routine is invoked. This is useful for situations in which you want the WBM CGI to behave differently. For example, if you concatenate "?is_waiting_page=1" to the URL of OpenRG's WBM in the browser, then instead of displaying a regular WBM page, the WBM only answers by 0 or 1.

4. If the session associated with the request is unauthenticated, then the print function of the login page is invoked (thus, access to most of the pages is blocked for unauthenticated sessions).
5. The `s_page()` function is invoked. This function handles global buttons and links. Every page has such buttons except from its specific buttons. For example, every page has topbar buttons that link to other pages. If one of the global buttons is clicked, then there is no need to invoke the page's scan function, and the button handler routine is invoked from `s_page()`, followed by a call to a page print function.

6. If no global button was clicked, then the current page's scan function is invoked. Every page scan function ends with a call to the next page print function (`p_page()` call).

7. `p_page()` does not print the result page yet. It sends back HTTP 302 code (Redirect) request to the browser.

8. After the browser responds to the HTTP 302 code (Redirect) with a subsequent HTTP GET request, `p_page()` is invoked again, generating the result page, and then the result page is sent back to the browser. `wbm_cgi()` ends the request (`g_request` memory is freed).

### 28.3.3 Print Page Function

General form:

```c
void p_scr_<page_name>(html_t **handle)
{
    frame = p_page_header(handle, g_request->active_page, NULL, ICON_DEFAULT);
    ...
    p_buttons(NULL, NO_BTN_DATA, btn_submit, ..., 0);
}
```

Most page print functions need to be able to do the following:

1. Call the `p_page_header()` function. This call prints the title of the page with its icon. The returned value of this function is a frame into which the page elements should be printed. Items that do not use the return value of `p_page_header()` will be printed outside of the page form. It is best to pass NULL as the title parameter and ICON_DEFAULT as the icon. This will place in the page both the icon and title that were declared for the page upon its registration.

2. Print HTML objects. HTML objects are printed into the frame handle, and sub-objects are printed into the objects handles. For example, if the page contains a table that has one row, the row has one cell, and the cell contains the text "Hello World", then you need to nest HTML elements one in another. First, the table will be printed into the frame:

   ```c
   table = p_table(frame, ...);
   ```
   The next step will be to create a table row, so the table will be passed as a handle to `p_row()`:

   ```c
   row = p_row(table);
   ```
   The next step will be to create a cell inside the row, so the row will be passed as a handle to `p_cell()`:

   ```c
   cell = p_cell(row, ...);
   ```
   The next step will be to create text inside the cell, so the cell will be passed as a handle to `p_text()`:
3. Call the `p_buttons()` function. This function prints buttons such as "OK" and "Cancel" at the bottom of the page.

### 28.3.4 Scan Page Function

**General form:**

```c
void s_scr_<page_name>(void)
{
    extract_data_t extract[] = {
        ...
    }
    int btn = button_num();
    if (btn == btn_onchange)
        goto Exit;
    else if (btn == btn_submit)
    {
        extract_data(extract);
        if (g_request->errors)
            goto Exit;
        set_set_path_str(rg_conf, ...);
        openrg_reconf(FLASH_DELAYED_TOP_PRIORITY);
        active_page_set(navigator_pop());
    }
}

Exit:
    p_page(NULL);
```

Most page scan functions need to be able to do the following:

1. **Handle buttons.** The first step that the scan function needs to take is to detect what button was clicked, and then apply the appropriate routine. One of the most important buttons is `btn_submit` (the "OK" button). When the user clicks this button, the form is submitted. The page scan function needs to write the posted data to `rg_conf`.

   Another important button that is handled in many page scan functions is `btn_onchange`. There are some objects in certain pages that changing their state causes re-submission of the form, due to an event-driven Javascript code invoked on the browser. Such a re-submission event is equivalent to clicking `btn_onchange`. If a page scan function detects that `btn_onchange` was clicked, then it should only refresh the page by calling `p_page()` (no extraction or writing to `rg_conf` is needed).

   As aforesaid, general buttons such as the sidebar links are intercepted by `s_page()` and thus the page scan function is not even called. Other general buttons that are intercepted by `s_page()` are `btn_nav_back` (the "Close" button) and `btn_cancel` (the "Cancel" button), since there is a fixed and page-independent routine for them.

2. **Call the `extract_data()` function.** In order to extract the user's input into local variables, you should create a small array that will define what should be extracted and of what type it is. A call to `extract_data()` with this array will populate the designated variables in the array with the values input by the user, after validating that these values are consistent with the types declared (for more information about input validation, refer...
to Section 28.3.5). Any errors will be automatically added to the g_request->errors list.

3. Handle input errors.

• Except from the errors that can be found by extract_data() call, more logical errors can be found by the page scan function itself, and added to g_request->errors using p_error() call. Typically, these types of errors will involve aspects that cannot be checked in extract_data(), like cross-dependencies between two or more variables.

• At the end of the scan, errors and warnings should be checked. If g_request->errors have a value other than NULL, extraction fails. In that case no values should be used or written to the permanent storage (rg_conf). After the user clicks the "Close" button in the error page, the page print function is automatically called again, giving the user the ability to fix his errors.

• If there is a need to warn the user or confirm an action, the p_warning() function can be used. At the end of warnings addition, just like for errors, g_request->warnings should be checked. After confirmation, the same scan function is automatically called again, but this time all the p_warning() calls are ignored, and g_request->warnings is NULL.

• If errors or warnings were found, the next page will be an error or warning report even if the active page is manipulated.

• Errors precede warnings, so if there are both errors and warnings, then an error page is printed as long as the errors are not fixed. Only when the errors are fixed then the user may proceed to a warning page.

4. Write to rg_conf.

• Data needs to be saved to rg_conf either directly, by calling set_set_*(), or indirectly, by using management library calls.

• openrg_reconf() must be called in order for the changes to take place.

5. Navigate to other pages.

• active_page_set() should be invoked with the next page as an argument.

• In order to go to the "parent" page (the page which redirected to the current page), you should use:

```c
active_page_set(navigator_pop());
```

6. Printing the new page.

• In order the make sure that the new active page is printed, any page scan function should call the general printing function p_page(). In many cases it is under an Exit label.
In most cases `p_page()` is called with NULL as argument. This argument is used for conveying information from the scan function to the print function. This will be discussed in detail in Section 10.3.

### 28.3.5 Input Validation

All the values that the user enters while configuring OpenRG's various modules should be validated before they are written to `rg_conf`. Input validation is performed when the `extract_data` function is called. The `extract_data` function uses an array that defines what should be extracted and of what type it is. A call to `extract_data` will populate the designated variables in the array with the values entered by the user, after validating that these values are consistent with the types you have declared.

When adding an editable field to a WBM page, you must decide on the way the field's input validation mechanism will be written. Your decision should be based on whether you are going to perform validation of the same input type more than once, throughout the WBM pages. If a specific input type is to be checked in one WBM page only, it is recommended that you add the validation mechanism locally—right after the `extract_data` function. If the same input type is to be used in several WBM pages, define this type and its value validation rule.

For example, the 'File Server' page (under the 'Storage' menu item of the 'Services' tab) contains a field called 'NetBIOS Workgroup', allowing the user to enter a string ≤ 15 characters long.

![File Server Page](image)

**Figure 28.8 File Server**

If you do not plan to use this type of input in other WBM pages, add the validation mechanism right after the `extract_data` function, as in the following example:

`rg/pkg/file_server/wbm/file_server.c`
1 void s_netbios_workgroup(int is_validation)
2 {
3    char *fs_workgroup;
4    extract_data_t fs_extract[] = {
5        {SYM_FS_WORKGROUP, {str:&fs_workgroup}, TYPE_STR_WITH_SPACES,
6            Tworkgroup_name},
7        {NULL}
8    };
9
10   extract_data(fs_extract);
11   if (strlen(fs_workgroup) > 15)
12       p_error(...);
13
14   extract_data(fs_extract);
15   if (strlen(fs_workgroup) > 15)
16       p_error(...);
17

**Line 5**

- **SYM_FS_WORKGROUP** – A unique HTML symbol, which represents the netbios workgroup field in the HTML form.

- **str**: The data type of the following variable (in this case, &fs_workgroup). This data type is defined in the `extract_data_t` structure (see `rg/pkg/web_mng/lib/wbm_util.h`).

- **&fs_workgroup** – A variable into which the input value is inserted, after the `extract_data` function is called.

- **TYPE_STR_WITH_SPACES** – Used as an example of a general-purpose input type used with editable fields (without a predefined value range/type).

**Line 6**

- **Tworkgroup_name** – The left side of the error message, which appears on the screen when an unacceptable string is entered.

**Line 11-12** Validation of the `fs_workgroup` value is performed.

If you are going to use the same input type more than once, it is recommended that you define a new type and its validation rule. For example, in case of the aforementioned page, define `TYPE_WORKGROUP` in the `rg/pkg/web_mng/lib/wbm_util.h` file:

```
TYPE_WORKGROUP = 56,
```

In this example, 56 is the type's unique identifier. Note that the order in which the types appear in the enum has no significance.

After the type is defined, open the `rg/pkg/web_mng/lib/wbm_util.c` file, and find the `extract_validate_data` function (which is ultimately called from `extract_data`). This function contains all input types currently used, with their validation rules. Add a case for the newly defined type. For example:

```
case TYPE_WORKGROUP:
    local_error = extract_str(list->name, list->u.str,
                              STR_WORKGROUP, list->error_title, scan_type, is_optional);
    break;
```

In the current example, the type's validation rule is defined in the `STR_WORKGROUP` case:

```
case STR_WORKGROUP:
    if (strlen(val) > MAX_WORKGROUP_NAME_LEN)
        return Terr_workgroup;
    ...
```
where the MAX_WORKGROUP_NAME_LEN is defined as 15 (see `rg/pkg/include/rg_def.h`). Note that Terr_workgroup is the right side of the error message that appears if validation fails.

As the last step, add the new type (TYPE_WORKGROUP) into the `extract_data` function located in the `rg/pkg/file_server/wbm/file_server.c` file. For example:

```c
void s_netbios_workgroup(int is_validation)
{
    char *fs_workgroup;
    extract_data_t fs_extract[] = {
        {SYM_FS_WORKGROUP, {str:&fs_workgroup}, TYPE_WORKGROUP,
         Terr_workgroup_name},
        {NULL}
    };
    extract_data(fs_extract);
    if (is_validation)
        return;
    set_set_path_value(rg_conf, Sfs "" Sworkgroup, fs_workgroup);
}
```

The data will be validated according to the rule defined for the type.

You can make the input field optional for the user, by adding the TYPE_OPTIONAL type:

```c
{SYM_FS_WORKGROUP, {str:&fs_workgroup}, TYPE_WORKGROUP | TYPE_OPTIONAL,
```

In addition, you can perform validation of several input types within the `extract_data` function, by adding types in the manner explained earlier. For example, consider an array, which validates the data a user enters to log into the WBM:

```c
extract_data_t extract[] = {
    {SYM_USERNAME, {str:&params.username}, TYPE_USERNAME, Tusername},
    {SYM_PASSWORD, {str:&passwd}, TYPE_EMPTY_PASS, Tlogin_pwd},
    {NULL}
};
```

Alternatively, you can write a separate `extract_data_t` array for each of the input types, as in the following example:

```c
eextract_data_t extract_username[] = {
    {SYM_USERNAME, {str:&params.username}, TYPE_USERNAME, Tusername},
    {NULL}
};
eextract_data_t extract_password[] = {
    {SYM_PASSWORD, {str:&passwd}, TYPE_EMPTY_PASS, Tlogin_pwd},
    {NULL}
};
eextract_data(extract_username);
eextract_data(extract_password);
```

## 28.3.6 Initialization

During reboot, all the entities are initialized. In this initialization stage, the entities that have WBM pages (such as firewall, print server, etc.) register their WBM objects in the WBM engine. For example, the entity's pages are registered by the `page_register()` function and the entity receives a handle (identifier) for every registered page. With this handle, the entity can redirect to and unregister the pages when it is uninitialized.
As described in Section 28.2, every entity (component) that involves WBM pages should have a WBM subdirectory. When the entity is initialized, its `init()` function is invoked, initializing its WBM pages:

```c
void <entity_name>_init(void) {
    /* The entity initializes its own WBM pages */
    #ifdef CONFIG_RG_WBM
        <entity_name>_wbm_init(void);
    #endif
    ...
}

void <entity_name>_uninit(void) {
    /* The entity uninitializes its own WBM pages */
    #ifdef CONFIG_RG_WBM
        <entity_name>_wbm_uninit(void);
    #endif
    ...
}
```

Typically, `<entity_name>_init()` and `<entity_name>_uninit()` reside in `rg/pkg/<entity_name>/main/main.c`, whereas `<entity_name>_wbm_init()` and `<entity_name>_wbm_uninit()` reside in `rg/pkg/<entity_name>/wbm/<entity_name>_wbm_init.c`. This file is dedicated to initializing the entity's WBM pages, and it does not include the page scan and print functions themselves—the page scan and print functions typically reside in separate `.c` and `.h` files under the entity's WBM subdirectory.

For example, the following files reside in the `rg/pkg/print_server/wbm` directory:

- `print_server_wbm_init.c`
- `print_server_wbm_init.h`
- `printer.c`
- `printer.h`

The file `printer.c` includes the page scan and print functions, whereas `print_server_wbm_init.c` includes, among others, page registration calls. When OpenRG reboots, the print server WBM initialization function that resides in `print_server_wbm_init.c` is called. This initialization function registers all the relevant print server pages, which are actually callbacks that reside in `printer.c`.

### 28.4 The Configuration File

The following are some of OpenRG's WBM `rg_conf` entries. Their values can be managed both from the 'System Settings' screen and directly from the CLI.

- **wbm/theme** The active user interface theme.
- **wbm/session_lifetime** The idle time in milliseconds before releasing a session. The default is fifteen minutes.
**wbm/confirm_needed** Determines whether or not the WBM will ask the user for confirmation of certain actions.

**wbm/language** The GUI language. The entry is the language's abbreviation. Other values that are not directly manageable from the WBM:

**wbm/is_login_welcome_done** Indicates whether a user has accessed the WBM at least once. After the user has visited the welcome page, this flag is automatically set to 1, and the welcome page won't be shown anymore (unless the user specifically changes this entry to 0).

**wbm/is_login_setup_done** Indicates whether a user has defined his initial user name and password for login. After the user defines the user name and password, this flag is automatically set to 1, and the login setup page won't be shown anymore (unless the user specifically changes this entry to 0).

### 28.5 Debugging Techniques

There are some common problems that are inherent in WBM development. As a result of this, several debugging techniques have been developed to cope with these problems.

#### 28.5.1 Common Problem 1

The GUI looks distorted.

1. Possible causes:
   - It is most likely that the HTML, generated by the C language code, is not formatted as needed.

2. Debugging techniques (perform in the escalating order below):
   - Save the page source (HTML) from your browser. It is recommended that you use an HTML editing tool so that you'll be able to see the HTML code auto-indented. If you're familiar with HTML code, try to trace the problem manually.
   - Use an HTML validation tool. For example, [http://validator.w3.org/](http://validator.w3.org/) provides a convenient method to post your HTML code and get a results-page in return. This technique guarantees finding HTML format errors. For example, the following HTML code is incorrect, because a `<TD>` object must be surrounded by a `<TR>` object:

   ```html
   ... 
   <TABLE>
   <TD>123</TD>
   <TABLE>
   ... 
   ``

   The site above is capable of tracing such errors.
   - Set all table borders to 1. This way all the table borders in the page will be highlighted (see Figure 28.9). For example - when using `gvim` or `sed` on the HTML source, type:
Reload the HTML source via your browser (write the name of the file in the URL box of your browser).

Figure 28.9  Login Screen with Highlighted Borders

- Trial-and-Error: Perform iterations of modifying the HTML code and then reloading it via your browser, until you get a satisfying result. For example, assume you are editing the following HTML code:

```html
...<TABLE border="1">
  <TR>
    <TD width="10">Hello World</TD>
  </TR>
...</TABLE>
```

The problem is that the word "World" appears below the word "Hello" instead of appearing beside it. You can modify the HTML code to:

```html
...<TABLE border="1">
  <TR>
    <TD width="10">Hello World</TD>
  </TR>
...</TABLE>
```

Save the file, reload it via your browser and verify that the problem is resolved. If so, you should add C code that adds `\n call p_space();`. If not, continue to trial-and-error.

### 28.5.2 Common Problem 2

The user's selection via the GUI is ignored.

1. Possible causes:
• The scan function is not called. For example, there are buttons such as btn_goto that are intercepted by the WBM generic layer. If you have such a button in your page, the page's scan function won't be called when you press that button.

• The scan function is called, but it does not refer to the correct variable, or it is not using the correct object type. For example, assume that the following code prints a check box to the page:

```c
...  
p_button_checkbox(cell, "is_vpn", ...);
...  
```

The corresponding scan function should extract the value of the variable "is_vpn":

```c
...  
exttract_data_t extract[] = {
   {"is_vpn", {num:&is_vpn}, TYPE_CHECKBOX, ...},
   {NULL}
};  
extract_data(extract);
...  
```

Verify that:

1. You are using the correct variable string ("is_vpn" in this example).

2. You are using the appropriate type (TYPE_CHECKBOX in this example).

2. Debugging techniques:

• Use the `misc wbm_debug_set` CLI command. In OpenRG's CLI prompt, type:

```sh
OpenRG> misc wbm_debug_set 1
```

This will enable WBM debug information (which is disabled by default). After typing this command, a debugging information footer will be added to every page that you browse. In this footer you will see a table with all the HTTP POST data variables and their values, including hidden variables, such as the navigation stack (see Figure 28.10).
You can always disable the debugging mode by typing:

```
OpenRG> misc wbm_debug_set 0
```

This debugging technique is perhaps the most powerful one, and it is strongly recommended that you use it when developing WBM pages.

- Use a debugger and place a breakpoint in the appropriate point. It is most likely that you should place the breakpoint in the start of the scan function. However, if elements in your page are taken care of in the WBM generic layer, you should place the breakpoint even earlier, in `wbm_cgi()` (the WBM CGI entry point).

For detailed information about running a debugger tool on OpenRG, refer to xref linkend="sect_gdb"/

### 28.6 Multilanguage Support

OpenRG's Web-based management Graphic User Interface (GUI) can be customized with virtually any language. You can easily add language strings to your new components for a seamless integration with existing components; moreover, you can add an entirely new language by translating the WBM GUI.
28.6.1 Language Files

A language file is a collection of User Interface (UI) strings and their corresponding translation to different languages. Every OpenRG component may have a separate language file. The naming convention for a language file is <component name>_lang.csv. Language files typically reside in the WBM subdirectory of a component. For example: pkg/print_server/wbm/printer_lang.csv.

28.6.1.1 File Structure

Open a language file to view its structure. All language files are in Comma Separated Values (CSV) format, and are comprised of the following columns:

- **NAME**: This column is used to identify the string in the application code.
- **DEF**: The default column is the English text that can be translated.
- **_PRIORITY**: Every column that begins with an underscore ("_") is a comment column. This column marks the translation priority, P1 being the highest and P5 the lowest. P1 is reserved for page titles, buttons and main screens. P2 is reserved for inner screens. P3 is reserved for error messages. P4 and P5 are used for less critical strings, such as acronyms (for example TCP, IP, etc.). The priority field is merely a directive for the translator; it has not real effect on OpenRG’s behavior.
- **_NEED_UPDATE**: This column is used to indicate that the translation should be updated. Again, it is aimed at the translator.
- **_COMMENT**: This column is for free, textual comments.
- **_EXAMPLE**: This column is for usage examples, accompanying the DEF column. It is also aimed at the translator.

The rest of the columns are the translation columns. They are marked with language shortcuts in lowercase (fr, es, ru, he, ja, etc.).

28.6.1.2 File Compilation

A language file is compiled using a special language compiler. In the compilation process, the language compiler creates c and h files from the language (CSV) file. The c file will contain an array named `<file name>_array`, where `<file name>` is the name of the language file without the ".csv" suffix. For example, `printer_lang.csv` will be compiled into `printer_lang.h` and `printer_lang.c`. `printer_lang.c` will contain an array named `printer_lang_array`, holding all the needed strings. In the next stage, the c and h files will be compiled by a regular compiler to o (object) files.

28.6.2 Adding Language Strings

Adding new text to OpenRG’s WBM can be done in one of the following methods:
1. By adding the text strings to one of the existing language files, in most cases to the main `pkg/web_mng/cgi/wbm_lang.csv` file. This is the simplest method, requiring only the quick addition of new rows to a language file.

2. By adding a new language file containing the new text. To do so, follow these steps:

   1. Name the new file according to the convention, with the "_lang.csv" suffix. The new file must contain the same columns as all your other language files.

   2. Register the new file using a call to `lang_register()` (and a corresponding call to `lang_unregister()`) from within the initialization sequence. The `lang_register()` function receives one argument `<file name>_<array>`, which is the name of the array created by the language compiler.

   3. Add the new file name to `pkg/language/lang_file_list.txt`.

   4. Add a target to the Makefile in the subdirectory of the language file:

   ```
   JMK_O_OBJS+=`$(JMKE_O_LANG_FILES)`
   ```

   $(JMKE_O_LANG_FILES) is automatically populated with the language files in the current directory.

### 28.6.3 Adding a New Language

In order to add an entirely new language to OpenRG, you must first collect all the language strings to be translated from the system, have them translated, and then integrate them back into the system. While the translation itself must be done by a professional translator, the collection and dispersion of the language strings is mostly automated by OpenRG. To collect the language strings from OpenRG, perform the following:

1. Go to the `rg/pkg/language` directory.

2. Create one merged CSV file from all language files:

   ```
   $ make merge_to_csv
   ```

   The file `all_language_files.csv` will be created in the build directory.

3. Create an Excel file from the CSV file:

   ```
   $ make csv2xls
   ```

   The file `all_language_files.xls` will be created in the build directory.

4. Open the XLS file using Excel.

5. Add an empty column with a code name for the language. There are two name conventions:

   1. Two lowercase letters, for example "en", "es".
2. A combination of two pairs, for example "en_GB", "en_US", representing the language and country.

6. Fill the following fields in the new column (they should exist in all_language_files.xls):

   **Tlang_id**  The code that you picked above.

   **Tlang_code**  The language code that will be displayed to the user (this value must be in uppercase letters).

   **Tlang_name**  The name of the language (English, French, Spanish, etc.) that will be displayed to the user.

   **Tlang_enabled**  Set to zero if you want the new language to be disabled.

   **VTlang_def**  Leave empty if the new language uses the DEF column as its default. However, if the new language is actually a slight modification of an existing language, such as Mexican Spanish to Spanish, a dependency can be built. The new language will inherit most of its strings from another language, but will differ in some. In this case, VTlang_def should be set to the code of the base language. Fields that are left empty will yield their value from the language you pick. This inheritance method can be layered. For example: es_MX (Spanish-Mexican) inherits from es (Spanish) inherits from DEF (most likely to be English).

7. Save the file.

After fulfilling these steps, the Excel file is ready to be sent for translation. When receiving the translated file from the translator, perform the following to integrate the new strings into OpenRG:

1. Create a CSV file from the Excel file:

   ```
   $ make xls2csv
   ```

   The file all_language_files.csv will be created in the build directory.

2. Split the merged CSV file to separate language files:

   ```
   $ make split_from_csv
   ```

3. Add the new language to the compiled languages:

   1. Open pkg/build/feature_config.c.

   2. Set the CONFIG_RG_LANGUAGES token to the list of languages that you want to include in the distribution. For example, "DEF es ja" will include the English, Spanish and Japanese languages. If you omit this token, all languages will be included in the distribution.

   3. Compile the development tree ('make config' and then 'make' from the root).
The new language will be added to the system, and you'll be able to switch the GUI to this language upon log in.
OpenRG Over UML

29.1 Overview

Jungo's User-mode Linux (UML) is a virtual gateway development and run-time environment that enables developers to fully develop, test and debug their gateways on a PC, before having the CPE hardware ready. Along with Jungo's leading software development platforms, this allows all embedded development activity on one machine, without the need for the actual target hardware.

29.2 Installing UML

You can download an installation guide for OpenRG over UML from:
http://www.jungo.com/openrg/install.html

29.3 The Virtual Flash Layout

OpenRG over UML uses a flash layout similar, if not identical, to that of a real OpenRG target board. The following paragraphs will touch lightly only on the differences between OpenRG over UML and OpenRG running on a real target board. For more information regarding OpenRG's flash structure, refer to Chapter 6. The diagram below shows the flash layout of OpenRG over UML's flash.img file. The difference between this layout and the flash layout of a real OpenRG target board is that all the images (RGLoader image and OpenRG images) are actually ELF executables, which can be run on a Linux host.
The structure of the OpenRG image (openrg.img) or of the RGLoader image (rgloader.img) differs from that of a real target board, as can be seen in the following diagram:

Figure 29.1 OpenRG over UML Flash Layout

rg_uml_run is a simple and small user-mode executable. Its tasks are:

- To extract the kernel (openrg.uml) and the file system (fs.img) to the /tmp directory.
- To run the UML kernel with the appropriate parameters

29.4 Running OpenRG over UML

In order to simplify working with OpenRG over UML, a small user-mode utility is provided — jrguml (Jungo OpenRG over UML). This utility aims to emulate OpenRG as it is running
on a real target board. It is therefore contained in a virtual full flash image file (flash.img), together with an RGLoader, just as its non-virtual siblings. To build OpenRG over UML from a UML OpenRG tree, simply follow the same instructions as you would have when compiling OpenRG for a real board:

```
$ cd rg.uml
$ ./rg_config.sh
$ make
```

In order to create (or update) the flash.img file, so that it contains the OpenRG over UML image (openrg.img) that you have just created, issue the following command:

```
$ jrguml burn
```

If you only want to run the supplied OpenRG image file and not compile a new one, you do not have to follow the two steps discussed above (compiling the image and burning it into flash.img). Instead, perform the following:

```
$ cd cd_image/images
$ cp openrg.img rg_factory /tftpboot  # Copy the images to tftp server dir
$ jrguml purge                        # Reset the flash.img file.
$ jnetwork start                      # Start the networking environment
$ jrguml start --std                  # Start the rgloader
< ... rgloader boots ... >
OpenRG boot> flash load -u tftp://192.168.1.10/openrg.img -s 1
OpenRG boot> flash load -u tftp://192.168.1.10/rg_factory -s 6
OpenRG boot> system reboot
```

**Note: jrguml cannot be run by a root user. Before issuing the above command, ensure you are not in root user mode.**

This is useful for loading a new OpenRG image when you do not have a UML RG tree or when you wish to simply load an openrg.img file into flash.img. After you clear the OpenRG images from flash.img, RGLoader will not have any image to boot, and will therefore provide you with a boot prompt. Using the boot prompt you can load a new OpenRG image from a standalone openrg.img file. To run OpenRG over UML from the flash.img file, use the following command:

```
$ jrguml start --std
```

What you will find once you start OpenRG depends on the contents of flash.img. If you have run jrguml purge, RGLoader will have no image to boot, and you will be prompted with a boot menu. You may instruct RGLoader to download and burn an OpenRG image into flash, just as you would have with a real OpenRG target board:

```
Boot> flash load -u tftp://192.168.1.10/openrg.img
Boot> system reboot
```

Of course, if you have used jrguml burn or have left at least one intact OpenRG image inside flash.img, you will see OpenRG booting and you will be prompted to enter the user name and password. If you wish to exit and shut down OpenRG over UML, first make sure to close any Telnet sessions. Then, issue jrguml stop in any other terminal. Remember you can always consult the MAN page for more details and options:

```
$ man jrguml
```
29.5 Virtual Networking for a Virtual Machine

OpenRG is a networking product. UML is a virtual machine. How can one develop, debug and test a virtual networking product then? The answer is quite simple—establish a virtual network! OpenRG over UML is configured as an Ethernet-to-Ethernet router. Its LAN and WAN Ethernet devices are *TUN/TAP* devices, meaning they are virtual tunnel devices that have one end on the host side and one end on OpenRG’s side. For more information on TUN/TAP devices and other methods of setting up a UML network, visit [http://user-mode-linux.sourceforge.net/networking.html](http://user-mode-linux.sourceforge.net/networking.html).

In order to create complex networking setups that simulate real-board in real-world scenarios, use *jnetwork* (for more information, type `man jnetwork`). However, for the vast majority of use cases, you do not need to use jnetwork directly. Instead, *jrguml* will run it for you, configuring the virtual network as needed. If you run `ifconfig` after running *jrguml* or jnetwork directly, you will find many new network interfaces which you are unfamiliar with. These new interfaces are either TUN/TAP devices or bridges. Two of these bridges are of special interest—*wanbr* and *lanbr*. Finding out which devices are enslaved to them is easy using the standard Linux *brctl(8)* command on your host (as a root):

```bash
% brctl show
```

You will notice that the interface uml_0 is enslaved to wanbr, while uml_1 is enslaved to lanbr. uml_0 is the host end of a TUN/TAP device pair—its counterpart is eth0 inside OpenRG over UML. Similarly, uml_1 on the host is paired with eth1 inside the UML. The other TUN/TAP pairs which are enslaved to these bridges, as to the other bridges as well, need not concern you. Devices such as wan0_0, lan0_0 and lan1_0 are simply placeholders for other virtual UML machines, which are a part of the Jungo virtual test lab—*JLab*. For more details regarding JLab, consult Jungo’s marketing representatives.

29.6 Virtual Removable Storage

OpenRG over UML includes a removable storage (disk) emulation, enabling you to develop and debug features that require a storage in order to run. The *jrguml burn* command creates an empty disk file, with the default size of 10MB (you may use the -d flag to specify a different size). When issuing the *jrguml start* command, the board boots with no disk. To add or remove a virtual disk, use the following CLI commands respectively:

```
OpenRG> misc diskplug in
```

and

```
OpenRG> misc diskplug out
```
29.7 Debugging

29.7.1 User-Mode Debugging

As with any other remote target, you need to use `gdbserver` on the target side and `gdb` on the host in order to communicate with it. The gdbserver executable is available in the RG tree under `rg/pkg/gdb/gdb/gdbserver/`. Simply copy it to your TFTP root directory, instruct OpenRG over UML to download it via TFTP, run it and then connect to it using gdb from your host. On your host:

```
$ cp rg.uml/pkg/gdb/gdb/gdbserver /var/lib/tftpboot
```

In the OpenRG prompt:

```
OpenRG> system shell
/ # tftp -g 192.168.1.10 -r gdbserver
/ # chmod +x gdbserver
/ # ps
/ # ./gdbserver /dev/ttyS1 --attach 7
```

These commands download the gdbserver and change its permissions to be executable. You then need to look at the `ps` command output for the OpenRG executable PID, so you can tell gdbserver to attach to it (in this example the PID is 7). The kernel then prints on which pseudo terminal the serial is connected, which should be similar to `/dev/ptyab --> /dev/ttyab`. Run gdb on your host and connect to the gdbserver:

```
$ cd pkg/main
$ rt gdb openrg
$ target remote /dev/ttyab
```

From this point onwards you can expect straightforward debugging (for more information refer to Figure 20.1).

29.7.2 Kernel-Mode Debugging

Debugging kernel-mode code is a breeze with OpenRG over UML. You simply need to attach to the UML kernel process.

```
$ ps auxw | grep openrg.uml | grep ' T ' 
```

The command above provides you with the PID of the process named 'openrg.uml'. However, the PID you should attach to is the resulting PID minus 2. Run gdb on your host and attach to the UML kernel process:

```
$ rt gdb os/linux-2.4/openrg.uml
(gdb) attach 8765
```

Again, from here on you have straightforward debugging (for more information refer to Figure 20.1).
30

Firewall

30.1 General

30.1.1 Overview

With the rapid adoption of broadband access and "always-on" Internet connections, it is more important than ever for home and SOHO users to protect themselves against hackers that may put the security of their computers and data at risk. OpenRG's firewall provides both the security and flexibility that home and SOHO users seek. Once installed in a residential/ SOHO gateway, it provides a managed, professional level of network security, while enabling the safe use of interactive applications such as Internet gaming and video-conferencing.

Additional features, including browsing restrictions and access control, can also be easily configured through a user-friendly Web-based interface, or remotely by a service provider. In general, OpenRG's firewall is a software module that filters network traffic. It does so by examining packets according to a ruleset. A ruleset is a set of rules, where each rule defines an atomic firewall decision. An example for such a rule is to drop all packets sent to http://www.bad_site.com. These rules are ordered in chains, which can be called for specific scenarios. The order of chains and the rules within a chain create a "decision tree" or a program that each packet has to traverse in order to decide whether it is dropped or accepted.

The following diagram provides a high level overview of the OpenRG firewall's static architecture.
30.1.2 Terminology

- **Rule** – A firewall command that performs a criteria comparison and takes action accordingly. For example, a rule that accepts incoming FTP traffic: Protocol: TCP, To IP: WAN-IP, Port: 21, ACCEPT.
• **Chain** – An ordered group of rules, used for performance and ordering. For example, all rules relevant to the WAN device eth0 will be stored on the 'dev_eth0_in' chain. The firewall will check the rules in this chain only if the packet received is on device eth0. If there are 50 rules in this chain, it will check them only if the underlying device is eth0. This results in better performance and avoids unnecessary checking. Chains are also used in order to gather rules and operations that have common functionalities. For example, all QoS related rules that belong to device eth0 on the outbound direction, will reside under the 'main_qos_eth0_out' chain.

• **Rule Precedence** – Rule precedence is defined by the order of the chains, and the order of rules within the chains.

• **Connection** – An established communication channel between two hosts. The definition of the connection contains the two host IP addresses, the communication protocol used, and various protocol-specific variables such as ports for TCP or UDP.

• **Stateful Packet Inspection** – A method in which the first packet between two hosts is passed through the rules. If the packet is accepted, a connection is established, and from this point on, all packets will be matched to this connection. This allows high-end performance and improved protection against attacks.

• **Local Area Network (LAN)** – The private subnet behind the gateway. The firewall usually assumes LAN traffic is allowed (trusted).

• **Wide Area Network (WAN)** – The Internet. Default firewall policy regarding the WAN is to allow outgoing traffic, and to block incoming traffic.

• **Network Address Translation (NAT)** – A mapping between private IP addresses to routable IP addresses.

• **Network Address Port Translation (NAPT)** – Enables hosts with private IP addresses to share the same routable IP address (one or more), using protocol specific information.

• **Reverse NAT (Redirection)** – A method to implement forwarding to a non-routable IP address from a routable IP address. Used in order to access a server in the LAN from the Internet (Port Forwarding).

• **Bi-NAPT** – An extension of Reverse NAT that allows LAN machines to access port forwarding with its "non-private IP" just like a WAN machine.

• **Application Level Gateway (ALG)** – Performs application layer NAT. Enables specific applications to function from inside the LAN. These applications would otherwise be blocked by the firewall or malfunction due to contradiction between the information that lies in the packet headers and the information inscribed in the application data section. ALGs perform NAT according to packet data.

• **Denial of Service (DoS)** – An attack that exhausts computer resources causing it to refuse any incoming connections.
• **Spoofing** – An attack in which the attacker forges its source IP address.

• **Replay** – An attack in which the attacker records a valid reply packet and retransmits it over and over again.

### 30.1.2.1 Network Address Translation

Network Address Translation (NAT) is an Internet standard that enables a local-area network (LAN) to use one set of IP addresses for internal traffic and a second set of addresses for external traffic. NAT serves three main purposes:

• Allows LAN IP addresses to be mapped to public IP addresses

• Enables various number of LAN hosts to share the same WAN IP address

• Supports port forwarding

#### 30.1.2.1.1 Address Translation

When a packet arrives from a LAN host, its source address is changed to OpenRG's WAN IP address, and when replied, its destination is changed back to the LAN private IP address. Using this method, each LAN host is mapped to a different WAN IP address.

#### 30.1.2.1.2 Port Translation

When a packet arrives from a LAN host, its source IP address is changed to OpenRG's WAN IP address and its TCP/UDP port (or other protocol data) is changed to a free OpenRG port (the default policy—do not change a port if not required). The WAN host replies to the OpenRG WAN IP address and TCP/UDP port, and the NAT module translates it back to the LAN IP address and LAN TCP/UDP port. Using this method, many LAN hosts can be mapped to one OpenRG WAN IP address.

![Figure 30.3 NAPT on Outgoing Packets](image)

#### 30.1.2.2 Stateful Packet Inspection

When an IP packet arrives at the firewall from the Internet, the firewall must decide if it should be forwarded to the internal network. In order to accomplish this, the firewall checks what connections have already been opened between the local network and the Internet. If there is an open connection that applies to the packet that had arrived from the Internet, it will be allowed through. Otherwise, it will be handled by the firewall according to its policy. This is known as stateful packet inspection. The firewall looks at the source and destination IP addresses, the source and destination ports and the sequence numbers, to decide if the packet belongs to a current open connection. The firewall performs stateful packet inspection and only allows
traffic into the local network on connections opened from inside the network or on services explicitly opened by the administrator. To summarize, the main characteristics of the SPI are:

- In SPI, only the first packet of a connection is passed through the rules. All the following packets are passed through the connection.

- SPI allows the reply of the WAN to be forwarded to the LAN and not to be blocked by firewall rules.

- SPI allows NAT/NAPT implementation.

- SPI allows protection against various attacks such as WinNuke, Replay, and TCP validity.

- SPI allow complex protocol translations (ALGs).

The first packet from the LAN/WAN host is checked against the rules. If it is found to be allowed in/out, it is accepted and a connection is created. From now on, each packet is matched first to the open connections. If matched, the packet is forwarded to the LAN/WAN and it does not need to be checked against the rules. This allows faster communication. The connection carries information about the ongoing communication, and thus enables making decisions that are based not only on the last packets but also on the state of the connection as a whole. Therefore, all attacks that are based on partial information about whether to accept or drop a packet that arrives (such as fragmentation attacks or replay attacks), can be easily spotted and stopped. The connection enables bidirectional traffic, and enables the set of rules to remain small and intuitive.

### 30.1.3 Flow Description

OpenRG’s firewall is a software module that has two main features: security and Network Address Translation (NAT). Security is provided by filtering each packet passing through the firewall’s protected interface, and dropping forbidden packets. If a packet is allowed through, the firewall may decide to NAT it in order to allow LAN access to the WAN, or redirect it to the LAN.

OpenRG’s firewall consists of two components: configuration (config) and data-path (kernel). The configuration part collects all the firewall configuration data as configured by the user and the different tasks residing on OpenRG. It then generates a set of rules (ruleset) and the firewall policy. This configuration is then fed into the data-path module for execution on packet flow. The firewall configuration is a set of:

- Devices on which the firewall is active.

- Ruleset—all the manual and automatic firewall configuration compiled into the internal language of the firewall.

- Global flags to control the security level, logging, etc. The kernel part receives every packet received or transmitted on the firewall’s active interface, and scans it according to the ruleset. As the result of the scan, a firewall decision is made to either drop or accept the packet. If the packet is accepted, the type of NAT requested for this packet is also achieved from
the ruleset, and the packet is NATed. An accepted packet may create a connection to allow stateful inspection of the following packets of the same session.

The stateful inspection allows the firewall to:

• Work faster by omitting policy checking for each packet, if it already has a valid connection.

• Perform sanity checks on connections such as replay packets, WinNuke or invalid TCP sessions.

• Perform NAT/NAPT and application level gateway actions for each packet in a connection.

![Firewall Packet Flow](image)

**30.1.4 Directory Structure**

*Figure 30.5* depicts OpenRG's firewall directory structure:

![Firewall Directory Structure](image)

The firewall directory is located in OpenRG's `/pkg/` directory, and contains the following sub-directories:
config The usermode firewall configuration directory. Contains the opcompiler directory for opcode generation.

filter The HTTP filtering directory.

kernel The kernel mode data path code directory. Contains the alg directory for ALG definitions, and the proto directory for protocol control (IP, TCP, etc.).

main The firewall's main task CLI commands.

proxy A directory for proxies that use the firewall for redirection (e.g. mail filter).

wbm The firewall's WBM directory.

30.1.5 Rules and Rule Structure

A rule is a set of opcodes that create a conditional operation on the packet. Rules are usually created by management or other OpenRG components. A firewall rule is a generic API for controlling the firewall operation, providing full control over matching and filtering. This is also the firewall external interface for tasks that wish to configure firewall rules. Each firewall rule consists of a match section and an action section. When a packet arrives at the firewall, it is scanned according to the match section. If it passes the criteria in the match section, the rule action will be taken. For example, the following is an advanced filtering rule:

```plaintext
{rule
 (0
  (enabled(1))
  (match
   (services)
   (src
    (item
     (0
      {ip(212.1.1.160))
     }
    (description(My 1st Inline Netobj))
   )
   (dst
    (item
     (0
      {start_ip(192.168.1.111))
       {end_ip(192.168.1.115))
     }
    (description(My 2nd Inline Netobj))
   )
  (src_is_exclude(0))
  (dst_is_exclude(0))
  (services_is_exclude(0))
  (dscp
    (value(-1))
    (mask(63))
  )
  (dscp_is_exclude(0))
  (priority(-1))
  (priority_is_exclude(0))
  (length
    (from(-1))
    (to(-1))
  )
  (match_length(0))
  (from(0))
  (to(0))
  (description(My Advanced Inline Netobj))
  (action)
})
```
The match section will match packets that have a source IP in the subnet 212.1.1.160 / 255.255.255.224, and a destination IP that falls in the range of 192.168.1.111 and 192.168.1.115. If the packet is matched, the action to take is to accept the packet and to generate a log message. The important firewall actions are:

- **Drop** – Drops the packet.
- **Reject** – Drops the packet and sends an ICMP error or a TCP reset to the origination peer.
- **Accept Connection** – Accepts the packet (stateful).
- **Accept Packet** – Accepts the packet (stateless).
- **Log** – Generates a log message.
- **Call** – Calls a chain (refer to Section 30.2.3).

After being translated to the opcode language, the rule above will be represented as follows:

```plaintext
(ACTION
  (TYPE(ACCEPT_CONN))
  (LOG(1))
  (RULE_PERMANENT_ID(0))
)
```

30.1.6 **rg_conf Structure**

The firewall section in rg_conf can be divided into five parts:

1. **fw/policy** – Includes all devices, advanced filtering chains, access control chains, and transparent proxies chains.
2. **fw/filter** – Determines whether the firewall performs filtering.
3. **fw/rule** – Gathers all the parameters regarding the firewall’s internal rules, for example port forwarding, remote access, and parental control.
4. **fw/log** – Determines the type of messages that will be logged in the firewall log.
5. **fw/protect** – Determines the firewall’s protection level against udpflood/icmpflood/synflood.
30.1.7 The Reconf Mechanism

The firewall's decision whether it needs to perform reconf is based on several conditions:

- When another component needs to perform reconf, and it specifically requests the firewall to perform reconf.
- When an rg_conf section relevant to the firewall is changed.
- When at least one device changes its routing level or trusted status. To view an implementation, refer to pkg/firewall/main/firewall.c.

When reconf is performed, the firewall builds a new ruleset from rg_conf and the predefined rules. This ruleset is transferred to the kernel via a ioctl, along with a list of the devices on which the firewall needs to work. The kernel replaces the current ruleset with the new one, registers/unregisters the firewall's hook on the given devices, and verifies the validity of current connections against the new ruleset.

30.2 Usermode Implementation

30.2.1 Opcodes

Opcodes are the basic building blocks of the firewall. Every opcode represents an atomic operation to be performed. The main groups of opcodes are:

- **Logical opcodes** – Implement conditions using the AND, OR and NOT operations, and control the flow of the program.
- **Flow control opcodes** – Handle the opcode tree flow, calling chains and terminating the flow when needed.
- **IP and protocol opcodes** – Compare information from the packet's layer 3 and 4 headers to a given parameter.
- **States opcodes** – Perform connection related operations (refer to Section 30.3.2).
- **Action opcodes** – Assign an action to a packet.
- **Flags opcodes** – Mark flags on the firewall's internal structures that are related to this packet (refer to Section 30.3.2.2 and Section 30.3.2.9).

Important opcodes include:

- **OP_AND** – Performs all its child opcodes sequentially, until one of them returns 0. If all opcodes return 1, OP_AND will return 1 as well. If one of the performed opcodes returns 0, OP_AND will return 0 as well. An example:
If COND_2 returns 0, COND_3 will not be performed, and OP_AND will return 0. This opcodes section is equivalent to the C code: 
```c
if (COND_1 && COND_2 && COND_3).
```

- **OP_OR** – performs all its child opcodes sequentially, until one of them returns 1. If all opcodes return 0, OP_OR will return 0 as well. If one of the performed opcodes returns 1, OP_OR will return 1 as well. An example:

```c
(OP_OR
 COND_1
 COND_2
 COND_3
)
```

If COND_2 returns 1, COND_3 will not be performed, and OP_OR will return 1 as well. This opcodes section is equivalent to the C code: 
```c
if (COND_1 || COND_2 || COND_3).
```

- **OP_DO** – Performs all its child opcodes sequentially, regardless of their return value. An example:

```c
(OP_DO
 OP_1
 OP_2
 OP_3
)
```

All three opcodes will be performed here. The return value of the OP_DO opcode is the return value of the last performed child opcode.

- **OP_NOT** – Reverse the returned value of its child opcode. An example:

```c
(OP_NOT
 COND_1
)
```

If COND_1 returns 1, OP_NOT returns 0 and vice versa.

- **OP_CALL** – Performs a function-like call to the given chain.

Implementing the logical building blocks, these opcodes allow creating a programmable opcode flow. An interpreter at the kernel passes each packet on the opcodes tree, until a firewall/NAT/QoS decision is reached for the packet.

### 30.2.2 Services

A service is a protocol-specific condition that defines a specific communication method. For example, HTTP is a TCP-based service with a well-known destination port 80, and TFTP is a UDP-based service with a well-known destination port 69. Services are generally used as filters in the firewall rules 'match' section. For more information on firewall rules, refer to Section 30.1.5. Services reside in `rg_conf` as:

- Global services residing in the `rg_conf/service` section.
- Inline services embedded in rules.
The following are definitions of services in OpenRG's firewall:

**Trigger**  The service's protocol, source port range and destination port range.

**Opening**  If needed (for example in TFTP), the return connection from the host defined by a protocol's (such as UDP) source port range and destination port range. Used mainly for port triggering, where a trigger connection is defined to cause the 'opening' connection to be opened.

Each service may contain as many triggers and as many openings as needed. A service enables you to perform packet matching according to:

- Protocol
- Port
- Type of ICMP packets (for ICMP)
- Packet trigger characteristics to be checked with this service
- Connection to open when a trigger packet opens a connection

The following is an example of an HTTP service:

```plaintext
http service:

{3  // index
  (name(HTTP))  // name
  (old_id(16777219))  // compatibility with previous versions
  (advanced(0))  // advanced flag
  (description(web server))
  (trigger 0
    (protocol(6))  // tcp
    (dst
      (start(80))  // port 80
      (end(80))
    )
  )
}
```

The following is an example of a TFTP service:

```plaintext
{7
  (name(TFTP))
  (old_id(16777223))
  (advanced(0))
  (description(Trivial File Transfer Protocol))
  (trigger 0
    (protocol(17))  // Trigger: To UDP port 69
    (dst
      (start(69))
      (end(69))
    )
    (src
      (start(1024))
      (end(65535))
    )
  )
  (open  // Open this ports on an outbound traffic to
    0
    (protocol(17))  // UDP
  )
}
```
30.2.3 Chains

A chain in the firewall is a list of rules, which are performed sequentially one after the other, until one of them is satisfied. By having a distinct functionality, the chain enables you to create a modular firewall ruleset. The flow of a packet in the firewall chains is controlled by rules that have the 'Call' as their action. An example for the use of chains is the 'Advanced Filtering' rules in the firewall's WBM, which provides you with an interface to fill up several chains:

- Initial inbound: first rules performed by the firewall for inbound packets.
- Initial outbound: first rules performed by the firewall for outbound packets.
- Inbound/outbound chains for each device: chains that consist of rules to perform when a packet is received or transmitted on a certain device.
- Final inbound: last rules performed by the firewall for inbound packets.
- Final outbound: last rules performed by the firewall for outbound packets.

Looking at the firewall ruleset as an execution program to the data-path module, the Chain represents calling to a function if some condition is matched. For instance, if the packet is incoming on device 'eth0' and NAT is available, call the `nat_main_eth0_out` chain. Chains can also return a value when their action is done.

30.2.4 Ruleset

The following section describes the OpenRG firewall's ruleset.

30.2.4.1 General

A ruleset is a tree-shaped component that contains the collection of all the per-device chains, predefined chains and all other chains, compounding the actual 'machine' through which packets will pass in the kernel. Distinguish between two kinds of rulesets:

- **VM_PKT** – This ruleset is created in order to filter and check every packet that is transmitted or received on a firewall protected device.
- **VM_CONN** – This ruleset is created in order to check the validity of established connections when the VM_PKT ruleset is changed (refer to Section 30.2.5).
30.2.4.2 Building a New Ruleset

The usermode builds new rulesets from scratch on every reconf (refer to Section 30.1.7) according to the following stages:

1. Generate opcodes chains—all non-empty chains are generated and stored in the ruleset. This is the stage in which all the predefined and user-defined rg_conf entries are being translated to opcode rules and chains (rg_conf chain to firewall chain related code can be found in pkg/firewall/config/adv_filter.c).

2. Generate per-device entry-point chains—for each firewall-capable device, the firewall tries to create inbound and outbound chains: main_<dev-name>_in and main_<dev-name>_out. Each of these chains will be added to the ruleset only if it is not empty (ruleset building and chain generating related code can be found in pkg/firewall/config/jfw_api.c).

3. Rule set optimization—drop logical opcodes (AND, OR) that have only one son underneath them:

\[
(\text{AND}\ \\
\text{COND}_1
)\]

It is logically equal to COND_1.

4. Rule set linkage—in this stage the ruleset is being indexed, which means that every command in the ruleset is being numbered sequentially. After indexing, all the chain names are replaced with chain indexes and the list of entry point index is generated for each device (linkage related code can be found in pkg/firewall/config/opcompiler/linker.c).

5. Loading the newly created ruleset to the kernel via a ioctl (loading rule set related code can be found in pkg/firewall/config/opcompiler/loader.c).

The firewall’s kernel holds 3 rulesets in an array:

- active ruleset
- soon-to-be-used ruleset
- conn ruleset

While building and loading the "soon-to-be-used" ruleset, the other (active) ruleset is still used for scanning traffic. Traffic is held only for a very short time needed to switch from the active ruleset to the new one. Later on, the old 'active' ruleset is deleted.

The global variable active_rule_set holds the index of the entry of the currently active VM_PKT. The global variable conn_rule_set holds the index of the entry of the VM_PKT.

After loading the new ruleset, the following operations occur:

- Updating of the list of devices on which the firewall listens.
• Reading the connection list and invalidation of illegal connections according to the new VM_CONN ruleset.

• Updating the active_rule_set and conn_rule_set variables (kernel operations upon new rulesets loading related code can be found in pkg/firewall/kernel/ioctl.c and pkg/firewall/kernel/ruleset.c).

The chains that build the ruleset can be categorized into the following categories:

• **Predefined chains** – Other than rules that are generated according to the user's preferences. The firewall uses predefined chains of opcodes that are used for routine firewall functionality. One example for such a segment is the preamble chain, which is the first one to be called for every packet that is scanned by the ruleset. Predefined chains can be divided into three types:

  1. **Preamble chain** – This chain handles the matching of the packet to an existing connection. If a packet matches a connection, the firewall stops the processing of the opcodes tree on the packet. This means that the full opcodes tree is usually run only on the first packet of a connection.

  2. **Fixed security checks** (chains openrg and openrg_in).

  3. **Chain callers chains** – Chains that decide the order and whether chains are called. The order of calling the chains is fixed.

• **User-Defined chains** – These chains contain rules defined by the user using the WBM.

• **Component-Defined chains** – Chains that contain component-related chains. When a component (external to the firewall) needs to configure firewall rules, it registers itself on the firewall. First, a `fw_external_conf_t` context is created:

```c
static fw_external_conf_t conf = {
    .name = <component_name>,
    .conf_get = <returns the component's rg_conf section>,
    .is_changed = <returns whether the components related rules in the rg_conf are changed>
};
```

This context is now registered on the firewall using the `mgt_fw_external_conf_register()` function.

Note: The firewall still needs to call the newly created chain. This code should be entered hard coded into `pkg/firewall/config/jfw_api.c` in the desired location in the ruleset flow.

### 30.2.4.3 Rule Set Entry Points and Chains Per Device Mechanism

When building the ruleset chains, an effort is made in order to minimize the packet's path through the ruleset. Creating a separate chain for each device and each direction saves the need for numerous conditions along the ruleset, and thus decreases the total time that the packet spends in the firewall and improves performance. The 'per-device'
chains related code can found in /pkg/firewall/config/jfw_api.c, starting with function jfw_dev_ruleset_vm_build().

Building the ruleset using chains per-device serves two purposes:

1. Avoiding unneeded operations on a packet per device.

2. Deciding whether the firewall's knet_hook (refer to Section 30.3.1) should be called on a certain device. If there is at least one chain that should be run on a device, the firewall's knet_hook is set on the device. Using firewall dump -pd, you can view on which devices the firewall hook is active (using administrator privileges).

The created 'per-device' chains actually create several entry points to the ruleset, which means that for different devices, packets will start their path in the ruleset from different entry points. When the user mode passes a new ruleset to the kernel, it also must pass a list of devices on which the firewall operates and their corresponding entry points in the ruleset. The code that creates the entry index for each device and transfers it to the kernel can be found in /pkg/firewall/config/opcompiler/linker.c.

The ruleset contains chains that can be roughly categorized into three groups:

1. **Firewall decision chains** – Chains that determine whether the packet will be accepted by the firewall, blocked or rejected.

2. **QoS decision chains** – Chains that apply QoS decision on the packet/connection.

3. **NAT decision chains** – Chains that determine whether the packet will be NATed, and how.

These three groups are independent and each marks its own decision on the packet. The chains arrangement inside the per-device chain is pre-determined. On an outbound per-device chain, the arrangement is as follows:

1. Preamble
2. Firewall decisions
3. QoS decisions
4. NAT decisions
5. Finalizing

On an inbound per-device chain, the arrangement is as follows:

1. Preamble
2. NAT decisions
3. Firewall decisions
4. QoS decisions

5. Finalizing

30.2.4.4 Translating an rg_conf Rule to an Opcode Rule – an Example

The following code is an example of an rg_conf representation of a firewall rule.

```plaintext
{rule
  (0
    (enabled(1))
    (match
      (services)
      (src
        (item
          (0 (ip(212.1.1.160))
          )
          (description(My 1st Inline Netobj))
        )
        )
      (dst
        (item
          (0
            (start_ip(192.168.1.111))
            (end_ip(192.168.1.115))
            )
          (description(My 2nd Inline Netonj))
        )
        )
      (src_is_exclude(0))
      (dst_is_exclude(0))
      (services_is_exclude(0))
      (dscp
        (value(-1))
        (mask(63))
      )
      (dscp_is_exclude(0))
      (priority(-1))
      (priority_is_exclude(0))
      (length
        (from(-1))
        (to(-1))
        )
      )
    (action
      (type(accept_conn))
      (log(1))
    )
    (rule_permanent_id(0))
  )
}
```

After being translated by the firewall, the opcode representation of this rule is as follows.

```plaintext
(IP_SRC_MASK(212.1.1.160:ff.ff.ff.ff)
 IP_DST_RANGE(192.168.1.111, 192.168.1.115)
 (DO
  LOG("fw/policy/0/chain/initial_inbound/rule/0")
  VAR_ADD(0, 1)=0 // fw/policy/0/chain/initial_inbound/rule/0
  ACCEPT
  RET(1)
)
```
30.2.5 VM_CONN Mechanism

The VM_CONN is a mechanism used by the firewall in order to check connections' validity, and update their parameters according to a new ruleset. A connection that was opened according to a previous configuration might be illegal according to the a new configuration, and hence should be removed. The VM_CONN ruleset is built in the same manner as the VM_PKT ruleset, except for the parts of the chains and opcodes that are omitted (such as preamble), because they do not have any meaning in the connection context. After a VM_CONN ruleset is loaded in the kernel, each connection is examined against it (related code can be found in pkg/firewall/kernel/firewall.c, function: vm_conn_run()).

1. A fin is created and the examined connection is attached to it.

2. The vm_conn ruleset is now called and the fin travels through it just like in the VM_PKT ruleset. The main change between these two rulesets is that the packets' information (such as source and destination IP addresses, ports, etc.) are taken from the connection's fields, and not from the packet headers as done in VM_PKT. This is done by re-using firewall code for both mechanisms:

   • The file pkg/firewall/kernel/cmd_exec.i is used for both VM_PKT and VM_CONN.

   • cmd_exec.i is included from both pkg/firewall/kernel/cmd_eval_pkt.c and pkg/firewall/kernel/cmd_eval_conn.c.

   • These files export 2 functions: fw_do_rule_pkt() and fw_do_rule_conn().

   • cmd_exec.i uses fields from either pkg/firewall/kernel/accessor_pkt.h or pkg/firewall/kernel/accessor_conn.h, which implement macros for accessing either packet fields or connection fields.

3. While traveling through the ruleset, the sticky chains list is being rebuilt as well, according to the new ruleset.

4. After the VM_CONN ruleset exits, the firewall's decision regarding this packet's validity is established. If invalid—the connection is terminated. If valid—the connection continues to live with its updated sticky chain list.

30.3 Kernel Implementation

The firewall's kernel section is called and operates via a knet_hook. For more information, refer to the knet_hooks mechanism description in Section 23.2.1.
### 30.3.1 Packet Flow

When caught on the firewall's knet_hook, the packet's handling in the firewall starts in `pkg/firewall/kernel/init.c`, function `fw_tx_hook()` / `fw_rx_hook()` first, where the possibility of using the *fw fastpath* mechanism is being checked. If the packet is "fw fastpathed", the hook returns (for more information, refer to Section 30.3.7). Otherwise, the packet is sent to the `fw_check()` function (which is in `pkg/firewall/kernel/firewall.c`). In this function, a `fw_info_t` structure (refer to Section 30.3.2.9) is built for the packet, and the packet is sent to the ruleset via `fw_do_rule_pkt()` (which is in `pkg/firewall/kernel/cmd_exec.i`). The following diagram details the packet flow inside the ruleset.

![Packet Handling Scheme](image)

When the packet returns from the ruleset, the fin flags contain the firewall's decision, which is returned to the knet_hooks mechanism.

### 30.3.2 Connections

#### 30.3.2.1 General

A connection represents a stream of data that passes through the gateway. OpenRG's firewall checks the first packet of the stream, and if it is accepted, a connection is opened. All the
following/replied traffic will pass on this connection, instead of passing on the full list of rules. After a short match, this traffic will be marked as accepted, and other operations applied on the first packet will be applied on it as well (NAT, QoS preferences, redirection, etc.). The benefits of using connections are the following:

- Improves performance
- Helps to monitor the stream’s state
- Helps to protect from various attacks (dos, IP spoofing, synflood, replay)
- Helps to gather statistics

30.3.2.2 The fw_conn_t Structure

A connection is represented by the fw_conn_t structure. Its implementation can be found in pkg/firewall/kernel/conn.h. The following scenario describes an FTP connection attempt, and displays the fw_conn_t structure fields used.

![Figure 30.7 FTP Connection Attempt](image)

In the above configuration, a LAN host, 192.168.1.2, attempts to establish an FTP connection with a WAN host, 192.168.61.1. Some of the fields in the created connection will contain the following values:

```c
typedef struct fw_conn_t {
    ...
    ...struct fw_conn_ops_alg_t *alg_ops;
    /** The device on which this conn is checked, matching the conn means the
     * packet is going in/out on this device */
    knet_if_ext_t *devx;

    u32 flags;
    fw_dir_t dir;  /** This is a definition of the connection itself. Needed
                    * to know on which flow the packet is (WAN->LAN or LAN->WAN)
                    */
    u8 protocol;  /** IPv4 next protocol */
    in_addr_t ip_in;  /** the IP in the LAN network: 192.168.1.2 */
    in_addr_t ip_out;  /** the external IP of OpenRG: 192.168.61.7 */
    in_addr_t ip_other;  /** the remote IP, like yahoo.com: 192.168.61.1 */
    nat_type_t nat_type;
    ...
} fw_conn_t;
```

30.3.2.3 NATing Packets with the Connections

NATing a packet’s address using a connection is done according to the following three fields:
If the packet is an incoming packet (WAN -> LAN), the firewall will replace the destination address from ip_out to ip_in. If the packet is an outgoing packet (LAN -> WAN), the firewall will replace the source address from ip_in to ip_out. The same applies to source and destination TCP/UDP ports, at the protocol handler (port_in, port_out, port_other).

### 30.3.2.4 Connection Data Structures

All connections are stored in two data structures:

1. **Hash table** – Each bucket in the hash table contains a linked list of connections that matches this hash entry. The ‘conn_hash_table’ defined in `pkg/firewall/kernel/conn.c` contains two hash tables, one for each direction:
   
a. Hashing on the hash table of DIR_OUT is done according to ip_in, ip_other, and potentially TCP/UDP port_in and port_other.
   
b. Hashing on the hash table of DIR_IN is done according to ip_other, ip_out, and potentially TCP/UDP port_other and port_out.

2. **Global linked list** – Used for sequential scanning all the connections.

### 30.3.2.5 Opening and Matching the Connection

A new connection is opened as part of the NAT/route or ALG handling in the firewall. When another packet of this stream will reach the ruleset, first it will be sent to the preamble chain (preamble_chain_fw_on, or preamble_chain_fw_off), where it will reach the OP_CONN_MATCH opcode. This opcode searches the connection's hash table for a matching connection. Upon a match the following opcodes are called:

- **OP_CONN_DO** – Marks the packet as accepted, calls the protocol specific code, calls the ALG code and the sticky chains if such exist.
- **OP_CONN_DO_NAT** – Performs a NAT action if needed on the packet.
- **OP_CONN_DO_BNAPT** – If port forwarding (local server) is defined, this opcode is called only in case a LAN client tries to access the local server using OpenRG’s WAN address.

When matching a connection (`fw_conn_get()`), the `fw_bucket_search()` function is used in order to locate a suitable connection. This function tries to match a connection to a given list of parameters (source and destination IP addresses, protocol, device direction and protocol-related data) according to a calculated bucket seed. First, a connection that fits the requirements with no wildcards is searched for. If not found—a search for one that fits the requirements is performed but with wild protocol options. If still not found, a compromise is made for a wild connection.
### 30.3.2.6 Sticky Chains

In some cases, even though a connection is needed, there are some chains that need to be executed for every packet, in order to simulate the exact actions on the packet, as if it went through the entire ruleset. Since sticky chains usually hold a very small number of rules, they do not cause a significant degradation in performance. An example for a sticky chain:

```
1 (AND chain=qos_sticky_chain_1
2   DO
3     PKT_MARK(mask:ffffffffffffff0000, value:20000)
4     IP_DSCP_SET(2a, 3f)
5     IP_PRIO_SET(4)
6     RET(1)
7   )
8 )
```

Note that no conditions are checked in the sticky chain—only marking operations.

**Line 3** Marks a certain class ID on the packet.

**Lines 4-5** Sets DSCP and priority on the packet.

In order to attach these chains to the connection context, the sticky chains mechanism is used: during the ruleset execution of the first packet, some of the calls to chains it passes through are defined as sticky calls. This is done by using the opcode OP_STICKY_CALL instead of OP_CALL. These chain indexes are saved in the "conn_call" linked list on the fw_info_t structure of the packet, and when a connection is created, this list is copied to it. During the OP_CONN_DO execution, the connection's sticky list is checked. If it is not empty—all the sticky chains in this list are executed.

### 30.3.2.7 Wild Connections

In some cases, when a connection is opened as a part of an ALG, not all details needed in order to open a full connection are known. For example, a SIP packet can inform that it expects RTP traffic at 192.168.1.5:5000 and 192.168.1.5:5001, but cannot provide information about the IP address or source port of the incoming packet. In this case, the ALG will open a 'half-defined' connection that matches all traffic destined to 192.168.1.5:5000, and mark the IP address and source port of the incoming packet with asterisks ("*"). This connection is called a wild connection. When a packet is matched on a wild connection, the ALG "un-wilds" it by filling its now known details in place of the asterisks, turning it into a standard connection. The types of available wild connections are:

- WILD_IP_OTHER – unknown ip_other
- WILD_PORT_OTHER – unknown port_other
- WILD_IP_IN – unknown ip_in
- WILD_PORT_IN – unknown port_in
- WILD_CONN – all the parameters are unknown
30.3.2.8 Connection Timeouts

A connection must have an expiration timeout, which is defined according to its protocol and state:

- An un-established TCP connection (before the three-way handshake is completed) – 2 minutes
- An established TCP connection – 5 days
- A UDP connection – 2 minutes
- An ICMP connection – 6 seconds

30.3.2.9 fw_info_t – A Packet Information Data Structure

fw_info_t is the main data structure that holds all the packet information. For its implementation, refer to pkg/firewall/kernel/fw_info.h. Important fields in this structure are:

```c
typedef struct fw_info_t {
    knet_buf_info_t *binfo;        /* pointer to OpenRG’s extension to the 
                                    packet information structure (Add reference) */
    fw_dir_t dir;                  /** Direction of the packet on the device */
    struct fw_conn_t *conn;        /** The firewall conn related to this packet */
    fw_conn_call_list_t conn_call; /** Chains list to be called 
                                    /* from the connection */
    int flags;                     /** Decisions regarding the packet */
    ...
} fw_info_t;
```

30.3.2.10 Connection Layers

A connection is built of three layers:

1. **IP Layer** – this layer performs all IP header related operations:
   - Replacing IPs in the header (NAPT)
   - Updating header len
   - Recalculating IP checksum

2. **Protocols Handling** – the protocols implementation in the firewall is done by filling a set of callbacks, defined in the fw_conn_ops_proto_t structure (refer to pkg/firewall/kernel/proto_ops.h). The following are important callbacks in this structure:
   - `match` – Matches a packet to a connection according to the protocol definitions.
   - `hash` – Returns a hash value of the protocol fields.
• action – Performs all protocol related actions upon CONN_DO opcode. During this function a new connection is allocated.

• action_nat – Performs the protocol's NAT operation upon the CONN_DO opcode.

• close – Closes the given connection.

3. Specific Protocols:

• UDP: refer to the implementation in pkg/firewall/kernel/proto/udp.c and pkg/firewall/kernel/proto/tcp_udp.c. The UDP callbacks handle the UDP connections’ creation and maintenance tasks, which include:
  - Connection creation/matching
  - Manage connections timeout
  - Connection termination

• TCP: refer to the implementation in pkg/firewall/kernel/proto/tcp.c and pkg/firewall/kernel/proto/tcp_udp.c. The TCP callbacks handle the TCP connections’ creation and maintenance tasks, which include:
  - Connection creation/matching
  - The three-way handshake statistics handling
  - Keeping the connections’ sequence and ACK IDs
  - Validating that the TCP packet is in the given window/seq
  - Terminating connection sequence (FIN -> FIN-ACK -> ACK)
  - Kill connection upon RST

30.3.3 Application Level Gateway (ALG)

30.3.3.1 Overview

30.3.3.1.1 What is an ALG?

A firewall/NAT works on layers 2-4 of OSI: MACs, Protocol, source IP, destination IP, and in case of TCP/UDP – source port and destination port. The general engine has no knowledge of specific applications (layer 7) data. In order to perform actions according to application-specific data, an ALG has to be activated. The ALG is a code that has knowledge of the
application data, can parse TCP/UDP payload according to a specific application, and can act accordingly.

### 30.3.3.1.2 Functionalities of an ALG

- **Application specific NAT** – Some protocols (i.e. FTP, SIP, and others) send their source IP within the packet's payload. The ALG modifies the IP payload.

- **Firewall pinhole** – Some protocols use two streams, one for control and one for data (i.e. FTP, SIP, etc.). In order to allow the incoming data stream to pass the firewall, the ALG can open a pinhole in the firewall according to the information found in the control stream.

- **QoS** – OpenRG's Jfirewall can maintain QoS ruling through control and data streams. That is, a QoS decision is only made for the control stream, but maintained through the data streams as well. This is useful when the data stream cannot be marked because it has no distinguished IP/port attributes (for instance, SIP's RTP stream can be at any UDP port, hence the data cannot be generally marked).

### 30.3.3.1.3 Supported ALGs

All the firewall's ALGs are implemented at: `pkg/firewall/kernel/alg`. Supported ALGs include:

1. Instant messaging:
   - AIM – AOL Instant Messenger
   - MSNMS

2. VoIP:
   - H323
   - SIP
   - MGCP
   - RTSP

3. Encrypted protocols:
   - GRE – an IPSec related protocol
   - IPSec
   - L2TP
   - PPTP

4. General:
• DHCP
• DNS
• FTP

30.3.3.2 Working with Jfirewall's ALGs

This section describes the components of Jungo's firewall (Jfirewall) ALG, and provides a practical step-by-step guide for adding a new ALG. Each step is accompanied by an example using the DNS ALG—a very simple ALG, which has a sole purpose of ensuring that each DNS query is answered by no more than a single reply. It is presented as a test-case. In order to view all the occurrences that relate to this ALG, perform "grep" for ALG_DNS at the OpenRG codebase, starting at the pkg/firewall folder.

30.3.3.2.1 What is an ALG Composed Of?

- **enum fw_alg_type_t** – This enum defines the list of ALGs. This fixed list is defined in pkg/include/nums.h.

- **rg_conf entry** Sets a rule for specific IPs/ports. The ALG rules are defined in rg_conf, at fw/policy/0/chain/alg_out and fw/policy/0/chain/alg_in. These chains define device, IP and port filters for the various ALGs. For example, DNS – assigns DNS_ALG ALG to destination port 53. The exact format can be viewed in the above rg_conf entries.

- **Firewall rules** – Directly derived from the rg_conf entry, and sets the ALG to the appropriate connection according to the filter provided in rg_conf. For example, assigns DNS_ALG ALG to traffic aimed to port 53.

- **Application-specific logic** – Logic that runs at kernel mode, as part of packet routing/bridging handling, and is responsible for opening a firewall and NAT connection, matching packets and modifying them, all according to application-specific logic. This logic is concentrated in a single file. For example: pkg/firewall/kernel/alg/dns.c.

- **Registration Code** – The kernel logic of the ALG is registered to the firewall.

30.3.3.3 How to Add an ALG

30.3.3.3.1 Definitions

- Add a new type of ALG to the ALG type list `fw_alg_type_t`, in pkg/include/nums.h. This enum defines all existing ALGs. For example: ALG_DNS.

- Add a rule that triggers the ALG to specific ports/IPS in `fw/policy/0/chain/alg_out` and/or `fw/policy/0/chain/alg_in`. This is usually done in the default rg_conf, using the
fw_alg_add() function, which is called from pkg/main/gen_rg_def.c. The decisions that must be made in order to add the ALG are:

1. Does the ALG need to work only outbound? Or both inbound and outbound?

2. What are the protocol, ports (and possibly devices and IPs) that the ALG should work on?

For example: adding ALG_DNS at fw_alg_add(): add ALG_DNS to UDP port 53.

- Add to alg_type_name() at pkg/firewall/user_kernel_common.c the name of the ALG. This name is used when viewing connections' details.

30.3.3.3.2 Kernel

- Add the .c file. For example: pkg/firewall/kernel/alg/dns.c.

- Define a structure of type fw_conn_handler_t. This structure defines handler functions for ALGs, which override the default protocol handlers. The main functionality of this structure is to supply callbacks for opening connections for outgoing/incoming packets. The structure contains the following members:

  - alg_type – The new ALG defined in enum fw_alg_type_t. For example: ALG_DNS.
  - name – The ALG name. For example: "DNS".
  - protocol – Usually either IPPROTO_UDP or IPPROTO_TCP.
  - nat_new – Points to a function that opens a connection as a result of an outgoing packet, which was assigned the added ALG. Usually the operation of the function is to open the connection (using the appropriate TCP/UDP opening connection functions), and then assign the added ALG to the connection. For example: dns_conn_open().
  - redir_new – Points to a function that opens a connection as a result of an incoming packet, which was assigned the added ALG. Usually the operation of the function is to open the connection (using the appropriate TCP/UDP opening connection functions), and then set the added ALG to the connection. For example: dns_open_redir(). The redir function is required for the BNAPT function.

- Define a structure of type fw_conn_ops_alg_t. This structure supplies callbacks for handling packets of ALGed connections, according to application specific logic. The structure contains the following members:

  - type – The new ALG defined in enum fw_alg_type_t. For example: ALG_DNS.
  - name – The ALG name. For example: "DNS".
match – this function decides whether or not a packet belongs to the ALGed connection. This callback is run as part of matching a packet to a connection: when a packet is received/transmitted on a firewall/NAT/QoS enabled device, the firewall tries to match the packet to an existing connection. Matching is done according to device+protocol +source IP/port+destination IP+port. If the connection has an ALG assigned to it, and the ALG has a 'match' callback, the match is also "anded" with the result of the ALG match callback. The function returns either FW_MATCH, or FW_NOMATCH. This allows, for example, to create several ALG connections on the same ports, and distinguish between them only by an application specific ID. For example, dns_match() – distinguishes DNS connections by their DNS session ID.

action – this function sets application-specific logic on a packet that was matched to the ALGed connection. The logic can perform layer-7 parsing, open new child connections (for firewall pinhole), layer-7 NAT, etc.

Return values:

• FW_MATCH Packet is OK

• FW_INVALID Packet will be blocked

• ret_flags A flag ALG_EXIT will mark the caller to remove the ALG from the connection

An example: dns_pkt() does two things: keeps the DNS session ID of the session, and closes the connection after the first reply arrives.

open – initialization operations done when assigning an ALG to a connection.

Return values:

• ret_flags A flag ALG_INT_ERROR will mark that opening the ALG context has failed

An example: dns_open() allocates context for keeping the DNS session ID.

close – cleaning actions done when the connection is closed, or the ALG is removed from connection.

print – prints ALG information when displaying the connections list. Needed for debug and tracking purposes. An example: dns_print().

Registering ALG – The ALG kernel is registered by calling fw_conn_handler_register(), and providing it as an argument the structure of type fw_conn_handler_t. The registration function is called from alg_init() at pkg/firewall/kernel/alg/alg.c. For example: alg_dns_init().
Figure 30.8  Packet Flow with ALG at JFirewall
30.3.3.4 Miscellaneous ALG Aspects

30.3.3.4.1 Connections Family

A typical network transaction is many times composed of more then a single layer-4 (IPs +ports) connection. Many applications use several layer-4 streams in order to complete a single task. For instance, a SIP conversation is composed of one or more control streams for control, and one or more RTP streams. Keeping track of these relations is important mainly for closing the connections together. It can also be used for assigning common QoS attributes to all connections. These relations are not known at layer-4 of the networking, and require specific application knowledge. Jfirewall contains infrastructure for arranging connections in hierarchic families – a parent and children.

30.3.3.4.2 Wild (Uninitiated) Connections

Wild connections is how Jfirewall opens pinholes for expected traffic. A wild connection is a partly uninitiated connection, which some of its connectivity details are not yet known. When the first packet is matched to the connection, it causes the connection to be initialized, and all the unknown IPs/ports details are set according to this packet. From this moment the connection is a regular connection.

For example: FTP ALG starts with a control connection, usually on TCP port 21. When the client issues a get command, it sends (in non-passive mode) a PORT command that tells the server that the client waits for the file in a specific port. The server sends the file from any source IP+port, to the IP+port provided by the client. The FTP ALG will NAT the IP+port at the PORT command as needed, and also open a wild connection that can accept traffic from an unknown IP+port. An example of the wild connection:

```
192.168.1.10:20 <---> 204.4.4.4:20 <-----> *.*.*.*:
```

After a packet arrives, for instance, a packet from 204.4.4.5:1024 to 204.4.4.4.4:20, the same connection changes to:

```
192.168.1.10:20 <---> 204.4.4.4:20 <-----> 204.4.4.5:1024
```

30.3.3.4.3 Packet Modification

An ALG may modify the payload. That may pose some issues that must be taken into consideration.

-Checksum change – when changing the payload, the TCP/UDP checksum should be updated as well. This is done by Jfirewall automatically, by re-calculating the TCP/UDP checksum.

- Length change – when changing a packet, the packet length may be modified. In rare cases, it may even exceed the device MTU. This length modification has to be reflected in the IP header and packet length. This is done automatically by Jfirewall.

- TCP seq/ack adaptation – when a TCP packet length changes, it has impact on the buffer stream that the client and server view. This means that the TCP window's seq/ack
parameters must be updated from this moment on, reflecting the length change. This is done automatically by Jfirewall.

### 30.3.3.4.4 Text Parsing

Simple text protocols can be parsed by simple text searches. For more sophisticated text parsing, OpenRG has a library for parsing (located at `pkg/util/parse.c`). This library:

- Allows using a C string function freely, by temporarily adding '0' at the end of each token.
- Allows parsing strings 'in-place', instead of copying them – which saves malloc calls and CPU time.
- Allows 'keyword-value' searches, and searching for delimiters, in addition to using the C string library.
- Allows replacing tokens with different length tokens, while automatically maintaining the integrity of the rest of the buffer.
- Usage of the standard C library allows the parsing to be run both at kernel mode and user mode (refer to Section 30.3.3.4.5).

This parsing library is not discussed in detail in this guide. For a usage example, refer to file `pkg/firewall/kernel/alg/sip_parse.c`.

### 30.3.3.4.5 Unittests

Some ALGs require intensive parsing. There is hardly any way, in regular QA routine, to cover all the extreme cases that such an ALG can render. This causes a major problem when maintaining the ALG – even if the ALG was perfectly QAed once, there is no way of ensuring that a minor change in it will not affect any extreme case that can be met.

A partial solution is to write the ALG code in a way that separates the linkage between logic and the firewall/kernel. This way of coding is appropriate to ALGs with intensive logic. It allows the ALG 'logic' to call once to firewall/kernel callbacks, and once to unittest callbacks that imitate the firewall/kernel work.

This allows writing unittest for the ALG: the unittest runs in usermode, on the compilation host. It has a book of test cases, and passes each test through the ALG, verifying that the outcome of each test matches what the unittest expects to see. If it does not – the unittest reports an error to the screen. This allows modifying the more complex ALGs, with high assurance that the modification does not damage any unwanted scenarios, hence highly improving its maintainability.

### 30.3.3.4.6 Loopback Port Forwarding

Jfirewall enables LAN clients to access a LAN port forwarding, by accessing OpenRG’s WAN IP (for instance, LAN1 can access a web server on the LAN which is defined with a port
forwarding rule, by accessing OpenRG’s WAN IP port 80). In order to do this, both source IP and destination IPs are NATed (BNAPT).

30.3.3.4.7 Usermode ALG

This is a special type of ALG, which re-routes traffic to socket at usermode. This allows writing proxies that see a complete TCP stream – not only TCP packets.

30.3.3.4.8 QoS

Jfirewall QoS marking can mark all ALG connections (both control and data) at the same marking. This allows, for instance, RTP connections of unknown ports to receive VoIP marking.

There is also an ability to give different QoS attributes to different connections (microflows), as done with compliance to the TR-098 management protocol.

30.3.3.4.9 Fastpath

Fastpath (hardware acceleration) is not supported for ALGed connections.

30.3.3.4.10 Connections List

In order to view the firewall connections, including ALGs defined on them, ports, connection family, and ALG-specific data – from OpenRG’s CLI type:

```
firewall dump -ps
```

The same connections list can be viewed from the WBM, under the 'Firewall' tab.

30.3.4 Network Address Port Translation (NAPT)

NAPT is a translation of addresses and a port from a private network to an address and a port in a public network. NAPT hides the private addresses of LAN clients, and spares public addresses, which is a very limited resource with IPv4. NAT/NAPT is performed in all three layers of the connection:

1. **IP Layer** – Replace IP and fix IP checksum
2. **Protocol Layer** (in case of TCP/UDP) – Replace port and fix TCP/UDP checksum
3. **ALG layer** – Replace IPs in the body of the packet

For more information about the NAPT implementation of the IP layer, refer to `pkg/firewall/kernel/proto/ip.c` and `ip.h`. For NAPT implementation at protocol layer, refer to `pkg/firewall/kernel/proto/<proto_name>`. 
30.3.5 Opcode Handling

The trees of opcodes reside in `rule_set[MAX_RULE_SETS]` inside `pkg/firewall/kernel/ruleset.c`, and are of type `fw_ruleset_t`. The function `VM_FUNC` in `pkg/firewall/kernel/cmd_exec.i`, which translates to either `fw_do_rule_pkt()`, or `fw_do_rule_conn()` (refer to Section 30.2.5), is the one that passes the packet through the ruleset. `VM_FUNC` scans the ruleset starting from an entry given to it, which is taken from a device-specific data. The parsing of the ruleset is done with an assistance stack, which is defined in `pkg/firewall/kernel/cmd_stack.h`. Consider the following opcodes segment:

```
(AND
  IP_SRC_MASK(212.1.1.160:ff.ff.ff.ff)
  IP_DST_RANGE(192.168.1.111, 192.168.1.115)
)
```

The OP_AND opcode "pushes" its two sons one level down the stack, and only if both of them return 1 - the OP_AND opcode will return 1 itself. When the packet is popped from the upper most level of the stack, it is returned to the `fw_check()` function.

30.3.6 NAT/Connection Mapping

The firewall’s NAT acts as a ‘Port Restricted Cone’ as defined in RFC3489. This means that each LAN_IP:LAN_PORT is uniquely translated to exactly one RG_IP:RG_PORT, regardless of the destined WAN_IP:WAN_PORT. This mapping is implemented using the dual hash table: `map_hash_table[2]`. This hash table is used to search mappings both for outgoing traffic (by ip_in+port_in) and for incoming traffic (by ip_out+port_out). This is similar to how `conn_hash_table[2]` is used, only without considering the ip_other:ip_out. Whenever a connection is created:

- An outgoing mapping is searched, using the `map_hash_table[DIR_OUT]`. If exists, this mapping is used.
- Otherwise, conflicts are searched using `map_hash_table[DIR_IN]`. If such exist, another mapping is searched (trying to choose different port).
- After finding a unique mapping, it is inserted to the hash table, to both sides. Main functions in the NAPT implementation are:
  - `fw_conn_do_nat()` – Performs NAT on the matched packet according to its connection.
  - `fw_tcp_udp_conn_nat()` – NAT/NAPT TCP/UDP packet.
  - `fw_conn_map_get()`
  - `dev_nat_chain_do()` – performs NAT to initial matched/cloned connection for the outgoing packet.
30.3.7 Firewall Fastpath

Note: This section is not related to hardware fastpath.

As described before, when a packet passes through the firewall, a `fw_info_t` structure is initialized for it. Then, it is sent to the ruleset, and after a few opcodes and security checks it is matched to a connection and NATed. The firewall fastpath bypasses the opcodes tree with a short and quick code. This can be done only for pure data traffic—that is, legal TCP/UDP data packets, which already have an existing connection. Fortunately, this is the majority of packets. Packets that do not match this pattern are routed to the regular opcodes lane. Two basic stages are performed during the fastpath operation:

1. Validating that the packet can be fastpathed. For example, non-IP packets or fragments are not fastpathed. This is implemented in `pkg/firewall/kernel/cmd_eval_pkt.c`, in function `is_fw_fastpath_candidate()`.

2. A match is searched in the connections hash table for the packet. This is implemented in `pkg/firewall/kernel/cmd_eval_pkt.c`, in function `fw_fastpath_pkt()`. A connection can be matched to a packet via fastpath only if it is marked as `CONN_FW_FASTPATH_ENABLE`. UDP connections are marked with this flag upon creation, but TCP connections are marked with this flag only after the three-way handshake is over.

Basically, the fastpath code performs the same operations as the non-fastpath code, only in a quicker manner. Firewall fastpath code is implemented in `pkg/firewall/kernel/init.c` and `pkg/firewall/kernel/cmd_eval_pkt.c`.

30.3.8 Fragments Handling

Fragments pose many problems to the firewall, as they do not contain TCP/UDP headers, and their defragmentation is a source for a wide variety of attacks. The firewall handles this by an assembling/disassembling mechanism:

- The firewall assembles the packet (`frag_cache_defrag()` called from OP_DEFRAG). Fragments are delayed until all the fragments arrive.

- The complete packet passes through the firewall’s opcodes tree.

- If the packet was accepted by the firewall, OpenRG will break it again to fragments and pass them for further handling (`frag_cache_frag()`).

The pending fragments are held in a list called `frag_cache_ctx` of type `frag_cache_t`. This list has a limited size (up to 10 packets), to prevent memory exhausting. An expiration mechanism keeps partial fragments from occupying this list forever. Fragments handling code can be found in `pkg/kernel/common/frag_cache/frag_cache.c`. It is also possible to ban fragments entirely.
30.3.9 IP Statistics

Firewall statistics on connections can be used for accumulating statistics about IPs and applications. During its life cycle, the connection updates its packets and byte counters. When closed, this information, along with the IP, port, protocol, and device name are sent to the statistics section. The information that arrives from the dead connections is sent to the usermode through a char device, where it is stored in an hash table. All data is stored twice according to the IP and according to the application—and accumulated in these entries. Upon request from the usermode/TR69, these statistics are sorted and retrieved. Firewall statistics implementation can be found in: kernel side – pkg/firewall/kernel/ip_stats_chardev.c, usermode side – pkg/firewall/config/ip_stats.c.

30.4 Debugging Methods

30.4.1 CLI Commands

- Firewall dump commands. These commands dump the firewall’s current status:

  - `firewall dump -ai` – Prints the indexed VM_PKT ruleset
  - `firewall dump -v -ai` – Prints the indexed VM_CONN ruleset
  - `firewall dump -pd` – Prints the list of devices on which the firewall is active

  OpenRG> firewall dump -pd
  list of firewall active devices: br0 eth0

  - `firewall dump -pc` – Prints a list of all chains

  OpenRG> firewall dump -pc
  trans_pxy
  access_ctrl_allow
  http_pxy
  rg_tasks_in
  rg_tasks_out
  http_intercept_br0_in
  nat_chain
  alg_chain
  svc_rmt_man
  openrg
  openrg_out
  security
  preamble_chain_fw_on
  preamble_chain_fw_off
  main_br0_in
  main_br0_out
  nat_main_eth0_out
  main_eth0_in
  main_eth0_out

  - `firewall dump -c <chain_name>` – Prints the opcodes of the requested chain

  OpenRG> firewall dump -c openrg_out
  list of mac cache active devices:

  (DO chain=openrg_out // OpenRG policy OUT
  (AND // [23] IGMP packet
IP_PROTO(IGMP)
VAR_ADD(0, 1)=0  // [23] IGMP packet
ACCEPT
RET(1)
}
IP_PROTO(UDP)
IP_DST_MASK(224.0.0.0:f0.00.00.00)
VAR_ADD(0, 1)=0  // [24] Multicast IGMP connection
ACCEPT
RET(1)
)
(AND // Accept OpenRG initiated outbound traffic
(OR // OpenRG IP
IP_SRC_MASK(192.168.1.1:ff.ff.ff.ff)
IP_SRC_MASK(192.168.61.2:ff.ff.ff.ff)
)
ACCEPT
RET(1)
)

• **firewall dump -ps** – Prints all connections

OpenRG> firewall dump -ps
Active Connections 1, quota 1000.
New connections will be created above the quota if there are more than 1228800 bytes of free memory. Current free memory is 53575680 bytes.
Fastpath packets: 101, Fullpath packets: 132
ESTABLISHED/ESTABLISHED ttl 431996 bytes 2.9/3.2 pkts 45/62 sticky 0 eth0 NAPT
Outgoing  FW-FASTPATH FP-CAP HW-FP-REQ

• The **firewall trace** command prints a serialization of all the packets that pass through the firewall hook and optionally match the filtering criteria. This is somewhat similar to tcpdump, but with firewall orientation. Also, since it is done from knet_hooks, it is done in a lower level than tcpdump, and catches packets that sometimes do not reach tcpdump. For example: `firewall trace e -f ip.src=192.168.1.2`. This command will print all the packets that pass through the firewall and their source IP address is 192.168.1.2. For the complete list of filter and command help, use: `firewall trace -h`.

OpenRG> firewall trace e -f ip.addr=192.168.1.2
Returned 0

commands :
0549,0550,0514,0515,0516,0517,0518,0520,0521,0522,0523,0531,0532,0536,0537,
0541,0542,0545,0551,0552,0553,0554,0555,0556,0557,0179,0180,0181,0182,0183,
0184,0185,0186,0187,0188,0190,0191,0192,0193,0194,0195,0196,0197,0198,0199,
0550,0560,0561,0562,0566,0567
ACCEPTED

commands :
0570,0571,0514,0515,0516,0517,0518,0520,0521,0522,0523,0531,0532,0536,0537,
0541,0542,0545,0572,0573,0574,0575,0576,0577
ACCEPTED
30.4.2 Analyzing the Firewall Trace Output

The output for each packet contains three parts:

1. Details about the packet – Direction, device captured on, IP header details, and protocol-specific details.

2. A list of firewall commands – Contains the indexes of all opcodes that the packet passed through before a firewall decision was reached for it. In order to associate the indexes with the actual opcodes, you can use the output of `firewall dump -ai`. Note that the indexes of the opcodes may change when reconf is performed. Make sure to use a `firewall dump` output that was taken just before or after the `firewall trace` command. At the end of the commands list the firewall's decision is noted: ACCEPT/DROP/REDIRECT/HANDLED.

   Note: When a packet passes on devices on which the firewall hook is not active, they will not be printed by the firewall trace command.

3. The `firewall variable` command – its mechanism is a counter mechanism that supplies information about chain hit statistics. Whenever a packet passes through the `OP_VAR_INC` opcode, it increments the counter of the this chain. The output code below presents the firewall variables table.

   **Index**  Sequential index of the variable value—number of hits on this counter.
   **Name**  The counter name as given in the `OP_VAR_INC` opcode.
   **Operator Index**  The index of the operator in the ruleset, providing the number of packets that had passed through a certain chain.

   Note: If a chain is not a sticky chain—you should probably see only one hit per connection in its counter. This is caused due to the fact that all packets in this stream (except for the first one) are matched to its connection in the preamble chain, and do not go through the entire rule set.

   You can add your own variable printing in newly created chains if you want to be sure that packets pass through them. Simply add the `OP_VAR_INC` opcode to the chain.

```plaintext
OpenRG> firewall variable
    'index' 'value' 'name' [operator indexes]
    0   0 fw/policy/0/chain/trans_pxy/rule/5 [15]
    1   0 fw/policy/0/chain/trans_pxy/rule/5/action/action [20]
    2   0 fw/policy/0/chain/access_ctrl_allow/rule/0 [30]
    3   0 fw/policy/0/chain/access_ctrl_allow/rule/1 [53]
    ...
    24  0 Application Level Gateway // ALG_AIM [272]
    25  0 Application Level Gateway // ALG_PPTP [279]
    26  0 Application Level Gateway // ALG_IPSEC [286]
    27  0 Application Level Gateway // ALG_L2TP [293]
    28  1 Application Level Gateway // DHCP [301]
    29  121 Application Level Gateway // DNS [308]
    30  0 Application Level Gateway // ALG_FTP [314]
    31  0 Port trigger: L2TP [321]
    32  0 Port trigger: TFTP [329]
    33  0 Remote Admin accepted [338]
```
<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>0 [23] IGMP packet [346 373]</td>
</tr>
<tr>
<td>35</td>
<td>0 [24] Multicast IGMP connection [352 379]</td>
</tr>
<tr>
<td>36</td>
<td>0 ICMP protection [397]</td>
</tr>
<tr>
<td>37</td>
<td>0 Spoofing protection [410]</td>
</tr>
<tr>
<td>38</td>
<td>0 Broadcast/Multicast protection [418]</td>
</tr>
<tr>
<td>39</td>
<td>0 Echo/Chargen/Quote/Smork protection [431]</td>
</tr>
<tr>
<td>40</td>
<td>0 Defragmentation failed: [454 527]</td>
</tr>
<tr>
<td>41</td>
<td>0 ICMP REPLAY [466]</td>
</tr>
<tr>
<td>42</td>
<td>0 ICMP REDIRECT [469]</td>
</tr>
<tr>
<td>43</td>
<td>2243 ARP [491]</td>
</tr>
<tr>
<td>44</td>
<td>0 Bridge STP [508]</td>
</tr>
</tbody>
</table>
31

Quality of Service in a Nutshell

OpenRG's Quality of Service (QoS) module provides essential network traffic management functions, such as bandwidth limitation, traffic prioritization and queuing, as well as the TCP serialization control. By performing these functions, the QoS mechanism manages congestion of the traffic when it meets a bottleneck node on the network. Typically, the most significant bottleneck of the network appears where the high-speed LAN (a 100Mbps Ethernet NIC) meets a limited broadband bandwidth (an ISP's modem/router).

OpenRG's QoS utilizes the Traffic Control (TC) mechanism, which is part of the Linux kernel. In particular, OpenRG's QoS uses a number of Queueing Disciplines (Qdiscs) implemented in the Linux traffic control. These qdiscs have been tailored according to specific OpenRG design needs. This chapter provides an overview of how each of the qdiscs has been integrated into OpenRG's QoS, and how both the TX and RX traffic control mechanisms operate. In addition, this chapter provides you with a reference to specific locations in the RG tree, where you can further learn about OpenRG's QoS, by familiarizing yourself with the major parts of its source code.

31.1 Scheduling and Shaping the Egress Traffic

31.1.1 Using the Queuing Disciplines

The main qdiscs implemented in OpenRG's QoS are pfifo_fast, PRIO, and Hierarchical Token Bucket (HTB). The HTB qdisc can also be configured to use additional qdiscs attached to its traffic queues. These qdiscs will be described further in this section. Each of the mentioned qdiscs utilizes its unique packet enqueuing and dequeuing algorithms for handling the egress traffic.
Note: When OpenRG is up and running, managing the QoS settings can only be performed via the WBM or TR-098. When using the WBM, the main (root) qdisc of the network device is selected from the 'Queue Policy' drop-down menu in the 'Edit Device Traffic Shaping' screen. Selecting the 'Strict Priority' option enables the PRIO qdisc, while the 'Class Based' option enables the HTB qdisc. The pfifo_fast qdisc is not reflected in the WBM for reasons mentioned further.

- **pfifo_fast** – The default qdisc used on OpenRG's network devices. This is a classless qdisc, which manages three traffic queues, also known as bands. Within each band, the First IN First Out (FIFO) traffic queuing rule is applied. Mapping between the packet priorities and this qdisc's bands is performed as described in Section 31.1.2. Packets enqueued in the lower-priority bands are dequeued only if the higher-priority band is empty. Therefore, the pfifo_fast qisc is also referred to as the Strict Priority Queuing (SPQ) mechanism. Each band can receive a limited number of packets (usually, a hundred). In general, the length of the queue depends on the network device's configuration. When a band is full, the incoming packets are simply dropped, until the band has enough bandwidth for further enqueuing. Note that the pfifo_fast qdisc is unconfigurable—only its standard settings can be used.

- **PRIO** – In its default configuration, OpenRG's PRIO qdisc has three traffic bands. If required, PRIO can also be extended to contain up to eight bands. Much like the pfifo_fast qdisc, PRIO implements the classless SPQ policy. Mapping between the packet priorities and this qdisc's bands is performed as described in Section 31.1.2.

- **HTB** – This qdisc enables you to limit the network device's bandwidth, and to fine-tune the distribution of the bandwidth between your applications/LAN clients, by defining as many traffic classes as you need. The HTB reserves bandwidth for the classes according to their defined traffic rate settings, and splits the remaining bandwidth between the classes according to their assigned priority values. Unclassified traffic goes to the default (automatically created) class, which has no predefined bandwidth, unless the remaining (excess) bandwidth can be allocated for it. The bandwidth allocated for each user-defined class is computed according to the following parameters:

  - **Reserved Rate** – The minimum amount of bandwidth (rate), which a class can request. Note that if a class contains a number of internal sub-classes, the rate supplied for the parent class should be a sum of the rates of its children.

  - **Ceil Rate** – The maximum amount of bandwidth that a class can use. This limits how much bandwidth a class can borrow from other classes, in case they do not use the bandwidth at that moment. The default (undefined) ceil is set to 'Unlimited', meaning that the class can borrow as much bandwidth as other classes can lend it.

Each class can receive a priority, where (0) is the highest priority, and (7) is the lowest one. Note that the default class has priority 4 (by default). Prioritizing the classes is important for distributing the excessive bandwidth between them. For example, class A with priority (1) will be granted a much greater portion of excessive bandwidth than class (B) with priority (3), but for both classes the granted bandwidth cannot exceed their ceil rate. The HTB mechanism prevents high-priority classes from starving the classes with a lower priority, by enforcing a non-zero traffic rate in each user-defined class.
Each of the HTB classes has a queuing discipline attached to it. If none is specified, the default qdisc is PRIO, but it can be changed via the WBM to one of the following disciplines. The selected discipline determines how to split the bandwidth between entities that belong to the same class.

- **FIFO** – The First In First Out discipline treats all packets equally by placing them into a single queue, and dequeuing them in the same order as they entered the queue. This is the most basic qdisc used in OpenRG's QoS mechanism.

- **Fairness** – OpenRG implements the Stochastic Fairness Queuing (SFQ) discipline. This discipline ensures that each packet flow within a class has fair access to the network resources, by going over the queue of each flow and dequeuing one packet at a time.

- **Random Early Detection (RED)** – This discipline randomly drops TCP packets from the queue. The probability of dropping the TCP packets depends on the queue size. Using this method instead of the traditional "Tail Drop" one reduces the TCP packet retransmission, and smoothes out the sawtooth pattern of the TCP traffic flows.

- **Weighted Round Robin (WRR)** – This qdisc is classful. All non-empty traffic classes will get a bandwidth proportional to their respective weight (a numeric value, between 1 and 10000). The weight of each class is set manually in the WBM.

The HTB bandwidth limitation capability is utilized for creating and managing the traffic congestion on the network device. The following diagram illustrates an example of the HTB tree created when OpenRG's WAN device's bandwidth has been limited, and distributed between the HTB classes. Note that this QoS configuration can be applied on any OpenRG network device.
### 31.1.2 Packet Classification Mechanisms

When using either of the mentioned qdiscs, packets are distributed between the traffic queues in the following way. When a packet arrives at OpenRG's firewall, the firewall checks if this packet matches a defined QoS rule that associates the packet's attributes with a specific band/class. The following packet attributes might be inspected by the firewall when matching the packet to a QoS rule:

- Source/destination MAC
- Source/destination IP/hostname
- Source/destination port
- DSCP/801.1p value
- Packet Length

When the packet matches the rule, the next steps depend on the selected qdisc. When p_fifo_fast or PRIO is used, the firewall assigns the detected priority value to the priority field of the skb structure associated with the packet. The priority value can be either 0 (High), 1 (Medium), or 2 (Low).
1 (Medium), or 2 (Low). In case of the HTB qdisc, the firewall adds a class ID associated with the QoS rule to the `nfmark` field of the `skb` structure. The kernel enqueues the packet in a band (or class, in case of HTB), which is associated with the value assigned to the relevant `skb` field. In case of HTB, if no value has been assigned to the `nfmark` field, the packet is enqueued in the default HTB class. Owing to the firewall's Stateful Packet Inspection (SPI) capability, the subsequent packets (of the same type as the first one) are forwarded for enqueuing without first matching them to the QoS rule.

### 31.1.3 Packet Priority Mappings

On a wired connection, the order in which OpenRG's kernel schedules the packets for enqueuing is determined by a DSCP or 802.1p priority value stored in a packet. OpenRG's QoS supports the standard DSCP implementation, which uses 64 hexadecimal priority values (0x0 - 0x3F). Moreover, OpenRG provides an editable mapping between the DSCP values and eight 802.1p values. The 802.1p header includes a 3-bit prioritization field, which allows packets to be grouped into eight levels of priority (0-7), where level 7 is the highest one. Both of these types of packet priority markings are ultimately mapped to one of three traffic priority levels available in OpenRG's QoS: low, medium, and high. Note that OpenRG can also be configured to provide up to eight priority levels. On a wireless connection, OpenRG maps one of the QoS priorities to the packet's Wi-Fi Multimedia (WMM) value.

### 31.1.4 The TCP Serialization Control

In addition to prioritizing the traffic, you can also enable and configure the TCP serialization on the network device. After limiting the network device's bandwidth, a big data packet might block the connection for an unacceptable period of time. This often occurs on slow connections. For example: a 1500 bytes long packet arriving at a 56Kbps connection might block the next packet for 214ms. This next packet might be of higher priority (e.g., voice). Such a delay is unacceptable for the voice stream. To prevent this serialization delay, OpenRG's QoS utilizes the Maximum Segment Size (MSS) clamping. MSS is the largest amount of data that TCP is willing to receive, and it is marked in the TCP SYN/SYN-ACK packets. The MSS clamping value (entered in milliseconds, and translated into bytes) is manually defined by the user in the 'Maximum Delay' field located in the 'Edit Device Traffic Shaping' screen.

### 31.2 Managing The Ingress Traffic

In addition to the qdiscs managing the egress traffic, OpenRG also implements the ingress qdisc. This qdisc can be configured to operate in two modes:

- **Policing** Enabled by default. In this mode, the ingress qdisc can only drop packets that do not match the defined QoS rules.

- **Scheduling** Enabling this mode creates a pseudo device, on which one of the mentioned egress qdiscs is defined. The RX traffic entering OpenRG's network device is transmitted via the pseudo device to the LAN. As an egress qdisc is defined on the pseudo device, the traffic
coming through this device is treated as the TX traffic. This mode enables queuing of the traffic, before it is sent to the LAN clients.

31.3 The QoS Source Code

You can learn more about the implementation of the QoS module by looking at the following parts of the QoS source code:

- **rg/os/linux-<2.4 or 2.6>/net/sched** – Files in this directory handle the Linux implementation of the queuing disciplines used in OpenRG's QoS.

- **rg/pkg/qos/ingress_hook.c** – This file handles the ingress qdisc mechanism.

- **rg/pkg/qos/main/qos.c** – This file handles creation of qdiscs and traffic filters in the usermode.

- **rg/pkg/qos/wbm/** – Files in this directory handle the QOS module's Web-based management.

- **rg/pkg/iproute2/tc** – Files containing code of the TC command line utility, which can be used for debugging OpenRG's QoS configuration (defined qdiscs, classes, etc.). To use the TC utility, compile the code, load the resulting executable to OpenRG's RAM using FTP/TFTP, and run it from the shell.
32
Dynamic DNS

32.1 Overview

OpenRG’s Dynamic DNS (DDNS) service enables you to define a unique domain name for your gateway’s Internet connection, thereby allowing you to access the gateway or your home network’s services just by pointing the browser to this name. When using this feature, you will not need to check and remember your gateway’s Internet IP address, which may change in case of a disconnection from the ISP’s network.

The following illustration depicts the Dynamic DNS architecture in OpenRG:

![Dynamic DNS Architecture](image)

Figure 32.1 Dynamic DNS Architecture
32.2 Adding A New Provider

OpenRG supports updating Dynamic DNS records to supported providers of such a service. A number of mainstream providers are supported. OpenRG also supports an API to enable the addition of new providers, assuming that the update protocol is known. The API is divided into two parts:

1. 'MAIN'—contains the implementation of the update protocol for the new provider.

2. 'WBM'—contains the provider-specific part of the DDNS Web-Based Management. The generic DDNS WBM currently supports fields common to all providers, such as Username, Password, Hostname and the WAN device whose IP is used for updating the DNS entry of the hostname. Other fields such as MX, Wildcard and Offline are not supported by all providers and should be added to the provider-specific part of the DDNS WBM. In addition, provider-specific configuration entries reside in `ddns/host/%/provider`.

32.2.1 Main Task API

The DDNS Main Task has a provider-specific part, which implements the provider-specific update protocol. The provider-specific update protocol resides in the `rg/pkg/ddns/main/provider/<provider>.c` file and should be added to the Makefile. The API consists of a number of callbacks called by the generic DDNS code to implement the update protocol functionality and return specific data on requests. These callbacks are registered on initialization. The generic DDNS code supplies callbacks to the provider-specific code so it can be informed on progress and status of the update. These callbacks are supplied on each update request.

32.2.1.1 Provider Callbacks

The following callbacks should be implemented in `rg/pkg/ddns/main/provider/<provider>.c`.

- **Provider** – This is not actually a callback, but a string defined in `rg/pkg/ddns/main/provider/provider.h`. It is used as the ID for the provider. Must be implemented.

  ```c
  #define PROVIDER "provider"
  ```

  Provider should be defined in `rg/pkg/ddns/main/provider/provider.h`, as it is also required by the WBM.

- **<provider>_open** – Initiates the DDNS update process. Must be implemented.

  ```c
  static ddns_t *<provider>_open(ddns_params_t *params, ddns_callbacks_t *cb, void *context)
  ```

- **ddns_t** – Context of provider. Allocated and defined by provider code.
• **ddns_params_t** – Generic DDNS parameters allocated in generic code. Defined in `rg/pkg/ddns/ddnst.h`.

• **ddns_callbacks_t** – Generic DDNS callbacks implemented in generic code. Defined in `rg/pkg/ddns/ddnst.h`.

• `<provider>_close` – Closes or aborts the DDNS update process. Must be implemented.
  
  ```c
  static void <provider>_close(ddns_t *o)
  ```

• `<provider>_changed` – Returns whether any of the provider-specific `rgconf` entries have been changed. Optional.
  
  ```c
  static int <provider>_changed(set_t **old, set_t **new)
  ```

• `old, new` – Old and new ’set’s of `ddns/host/%/provider`.

• `<provider>_offline` – Returns whether the ‘offline’ option is set if it is relevant in this provider. Optional.
  
  ```c
  static int easydns_offline(set_t **set)
  ```

• `set` – set of `ddns/host/%/provider`.

• `<provider>_response_description` - Receives the last response of the provider's DDNS server (a string) and returns a detailed description of the response (also a string). Optional.
  
  ```c
  static char *<provider>_response_description(char *response)
  ```

• **Provider update timeout** – Integer that denotes the number of days after which an update must be performed even if the designated IP has not changed. Optional.

These callbacks are registered and unregistered using the following functions defined in `rg/pkg/ddns/main/ddns.h`:

```c
void ddns_provider_register(char *name, ddns_provider_open_t open,
                           ddns_provider_close_t close, ddns_provider_changed_t changed,
                           ddns_provider_offline_t offline,
                           ddns_provider_response_description_t response_description, int timeout);
```

Optional callbacks are NULL or 0.

```c
void ddns_provider_unregister(char *name);
```

### 32.2.1.2 Generic Code Callbacks

The generic code callbacks are passed to `<provider>_open` and are defined in `rg/pkg/ddns/ddnst.h`.

• **ddns_update_done** – Called by the provider code when the update is completed successfully or fails. It returns the generic DDNS code's context received in
<provider>_open the update's status and the IP actually updated reported by the DDNS server if it exists.

```c
void (*ddns_update_done)(void *context, ddns_status_t status, char *last_ip);
```

`ddns_status_t` – Possible status of update defined in `rg/pkg/include/enums.h`:

- **DDNS_STATUS_UPDATED** – Update completed successfully
- **DDNS_STATUS_ERROR** – Update failed
- **DDNS_STATUS_NOT_UPDATED** – Update not performed

- `ddns_provider_set_get` – Called by the provider code to get the set of `ddns/host/%/provider` in order to write the DDNS server's response in `ddns/host/%/provider/response`.

```c
set_t **(*ddns_provider_set_get)(void *context);
```

### 32.2.1.3 Configuration File Entries

The configuration file entries include a DDNS section, under which you will find `ddns/host/%/provider` entries. These entries are provider-specific. The following entries will always be specified:

- **ddns/host/%/provider/name** – The name of Dynamic DNS provider.
- **ddns/host/%/provider/response** – The Dynamic DNS's response to the last Dynamic DNS request. The response depends on the provider.

### 32.2.1.4 Implementation Steps


2. Implement the update protocol asynchronously in `<provider>_open`. When the protocol finishes successfully or unsuccessfully, save the provider's ASCII response to `ddns/host/%/provider/response` with the aid of `ddns_provider_set_get`. Call `ddns_update_done` to return the status of the update and actual updated IP (if available).

3. Implement `<provider>_close` to close or abort the asynchronous update initiated by `<provider>_open`. This is called by the generic code from `ddns_update_done`.


5. Implement functions `<provider>_register(void)` and `<provider>_unregister(void)`, which call `ddns_provider_register` and `ddns_provider_unregister` respectively with the relevant parameters. These
functions should be defined in \texttt{rg/pkg/ddns/main/provider/provider.h} and called from \texttt{ddns_init()} and \texttt{ddns_uninit()} respectively in \texttt{rg/pkg/ddns/main/main.c}.

### 32.2.2 Provider-Specific WBM Fields

The DDNS WBM has a provider-specific part, which implements the provider-specific WBM fields. The provider-specific WBM fields reside in the \texttt{rg/pkg/ddns/wbm/provider/<provider>.c} file and should be added to the Makefile. The API consists of a number of callbacks called by the generic DNS WBM code to implement provider-specific WBM fields and functionality. These callbacks are registered on initialization.

#### 32.2.2.1 Provider Callbacks

These callbacks should be implemented in \texttt{rg/pkg/ddns/wbm/provider/<provider>.c}:

- **Management URL** – This is not actually a callback but a string containing the URL of the provider's Internet user-account management page. Must be implemented.
  
  ```c
  #define PROVIDER_URL "<providers_account_url>"
  ```

- **p\_<provider>** – WBM 'p_' function, which displays the provider-specific fields. Optional.
  
  ```c
  static void p_<provider>(html_t **table, set_t **set)
  ```

- **table** – html_t of father table of the implemented fields.

- **set** – Set of \texttt{ddns/host/\%/provider} so this function can read and display provider-specific parameters in WBM.

- **s\_<provider>** – WBM 's_' function, which reads and saves fields set in the provider-specific part. Must be implemented only if \texttt{p\_<provider>} has been implemented.
  
  ```c
  static void s_<provider>(set_t **set)
  ```

- **set** – Set of \texttt{ddns/host/\%/provider} so this function can write provider-specific parameters read from WBM to \texttt{rgconf}.

These callbacks are registered and unregistered using the following functions defined in \texttt{rg/pkg/ddns/wbm/ddns.h}:

```c
void ddns_provider_wbm_register(char *name, p_ddns_provider_t p_provider,
                               s_ddns_provider_t s_provider, char *url)
```

Optional callbacks are NULL.

```c
void ddns_provider_wbm_unregister(char *name)
```
32.2.2.2 Implementation Steps

1. Create file `rg/pkg/ddns/wbm/provider/<provider>.c` and add it to `rg/pkg/ddns/wbm/provider/Makefile`.

2. Define Management URL.

3. Implement provider-specific WBM 'p_' and 's_' functions in `p_<provider>` and `s_<provider>` respectively if required. Note, the provider-specific WBM fields are part of an existing table whose handle is passed to `p_<provider>`.

4. Implement functions `<provider>_wbm_register(void)` and `<provider>_wbm_unregister(void)`, which call `ddns_provider_wbm_register` and `ddns_provider_wbm_unregister` respectively with the relevant parameters. These functions should be defined in `rg/pkg/ddns/wbm/provider/provider_wbm.h` and called from `ddns_wbm_init()` and `ddns_wbm_uninit()` respectively in `rg/pkg/ddns/wbm/ddns_wbm_init.c`. 
A Virtual Private Network (VPN) allows the home or SOHO user to establish a secure, and cost-effective communications path over the Internet, linking a local network to a remote computer or site. Widely used VPN technologies such as IPSec and PPTP allow users to safely and securely access a company intranet from a remote location, or connect to the home/SOHO network when traveling. OpenRG's three modules, IPSec, PPTP Server and PPTP Client, enable PCs on the local network to communicate over VPN connections with remote computers, without needing to run any additional VPN software. Pass-through support allows PCs on the local network to establish direct VPN connections with remote servers passing through the gateway's Firewall and NAT.

33.1 Public Key Infrastructure

Public Key Infrastructure (PKI) cryptography uses a pair of keys: a public key, which encrypts data, and a corresponding private key, for decryption. Your public key is made known to the world, while your private key is kept secret. Anyone with access to your public key can encrypt information that only you can read. The public and private keys are mathematically associated; however it is computationally infeasible to deduce the private key from the public key.

Anyone who has a public key can encrypt information but cannot decrypt it. Only the person who has the corresponding private key can decrypt your information. Technically, both public and private keys are large numbers that work with cryptographic algorithms to produce encrypted material. The primary benefit of public-key cryptography is that it allows people who have no preexisting security arrangement to authenticate each other and exchange messages securely. OpenRG makes use of public-key cryptography to authenticate and encrypt Wireless and VPN data communication.
33.1.1 Digital Certificates

When working with public-key cryptography, you should be careful and make sure that you are using the correct person's public key. Man-in-the-middle attacks pose a potential threat, where an ill-intending 3rd party posts a phony key with the name and user ID of an intended recipient. Data transfer that is intercepted by the owner of the counterfeit key can fall in the wrong hands.

Digital certificates provide a means for establishing whether a public key truly belongs to the supposed owner. It is a digital form of credential. It has information on it that identifies you, and an authorized statement to the effect that someone else has confirmed your identity. Digital certificates are used to foil attempts by an ill-intending party to use an unauthorized public key. A digital certificate consists of the following:

A public key A value provided by a designated authority as a special key that can be used to effectively encrypt messages.

Certificate information The "identity" of the user, such as name, user ID and so on.

Digital signatures A statement saying that the information enclosed in the certificate has been vouched for by a Certificate Authority (CA). Binding this information together, a certificate is a public key with identification forms attached, coupled with a stamp of approval by a trusted party.

33.1.2 X.509 Certificate Format

OpenRG supports X.509 certificates that comply with the ITU-T X.509 international standard. An X.509 certificate is a collection of a standard set of fields containing information about a user or device and their corresponding public key. The X.509 standard defines what information goes into the certificate, and describes how to encode it (the data format). All X.509 certificates have the following data:

The certificate holder's public key the public key of the certificate holder, together with an algorithm identifier that specifies which cryptosystem the key belongs to and any associated key parameters.

The serial number of the certificate the entity (application or person) that created the certificate is responsible for assigning it a unique serial number to distinguish it from other certificates it issues. This information is used in numerous ways; for example when a certificate is revoked, its serial number is placed on a Certificate Revocation List (CRL).

The certificate holder's unique identifier this name is intended to be unique across the Internet. A DN consists of multiple subsections and may look something like this: CN=John Smith, EMAIL=openrg@jungo.com, OU=R&D, O=Jungo, C=US (These refer to the subject's Common Name, Organizational Unit, Organization, and Country.)

The certificate's validity period the certificate's start date/time and expiration date/time; indicates when the certificate will expire.
The unique name of the certificate issuer the unique name of the entity that signed the certificate. This is normally a CA. Using the certificate implies trusting the entity that signed this certificate. (Note that in some cases, such as root or top-level CA certificates, the issuer signs its own certificate.)

The digital signature of the issuer the signature using the private key of the entity that issued the certificate.

The signature algorithm identifier identifies the algorithm used by the CA to sign the certificate.

33.1.3 Replacing OpenRG’s Certificate

You may wish to replace the default OpenRG certificate with your own one. You can do so from the WBM (refer to the 'Certificates' section of the OpenRG User Manual). However, if you would like to change the certificate prior to OpenRG’s compilation, perform the following:

1. In your development tree, go to the `rg/pkg/main` directory.

2. Replace the certificate (`openrg.crt`) and public key (`openrg.key`) files with your own ones.

3. Recompile the image and burn it to the board.

33.2 Relevant rg_conf Entries

The following rg_conf sections and entries are relevant to the VPN's operation.

- IPsec entries
- IPSec policies
- PPTP entries

To learn more on how to set up and configure a VPN connection, please refer to the "Network Connections" chapter in OpenRG’s User Manual.
This section describes how to add a proxy to OpenRG. The proxy is a service that intervenes between a client and a server; it listens on a loopback to a port, processes the data and passes it on. You can define a proxy either as part of the Main Task or as a separate process, and add a redirection rule at the firewall for the proxy. The firewall will redirect all requested traffic to the given port (and optionally an IP), and the proxy will listen on that socket and process the traffic before passing it on. Adding a proxy is divided into three steps:

1. Call a function to redirect traffic from the firewall to your OpenRG. This function creates a redirection rule to a given port. You can use the `fw_proxy_rule_add` function, defined in `rg/pkg/firewall/jfw_rules_config.c`. This function calls `fw_rule_trans_pxy_rule_add`, which calls `fw_rule_trans_pxy_rule_add_to_chain`. You must provide a port number, to which the traffic will be redirected. A rule is returned, and then you must add a service.

```c
set_t **fw_proxy_rule_add(service_t *svc)
{
    static u16 rdir_port = GENERIC_PROXY_LISTEN_PORT;
    set_t **rule;

    rule = fw_rule_trans_pxy_rule_add(rg_conf, rdir_port++);
    fw_set_rule_add_inline_service_match(rule, svc);
    set_set_path_flag(rule, Senabled, 1);
    return rule;
}
```

2. Create a process that listens on a loopback to the given port.

3. Add code to retrieve the original recipient connection information. When the traffic is redirected to OpenRG, the originating IP address is stored and changed, and must be restored before the proxy transmits the packet. The information is stored in `userspace_conn_lookup_t`:

```c
typedef struct {
    in_addr_t ip_other;
    in_port_t port_other;
} userspace_conn_lookup_t;
```
This structure is defined in `rg/pkg/firewall/kernel/firewall_interface.h`. Use the following function, taken from `rg/pkg/firewall/config/nat.c` to retrieve the original recipient connection information:

```c
int ipnat_rdir_look(u16 src_port, u32 *dst_ip, u16 *dst_port, u32 *user_data)
{
    userspace_conn_lookup_t lookup;
    memset(&lookup, 0, sizeof(lookup));
    lookup.port_out = src_port;

    if (openrg_module_ctrl(KOS_CDT_JFW, USERSPACE_CONN_LOOKUP, &lookup))
        return -1;

    *dst_ip = lookup.ip_other;
    *dst_port = lookup.port_other;
    *user_data = lookup.context;

    return 0;
}
```
Voice over IP (VoIP)

OpenRG's VoIP module provides developers with a production-ready software solution based on the Asterisk stack, targeting the development of Analog Telephone Adaptors (ATAs) and Private Branch Exchanges (PBX).

- ATA is home-oriented, where you can connect one or more phones to your VoIP box and they will function as separate telephones with different telephone numbers. They are separate entities, each with its own configurations.

- PBX is office-oriented, where the (unlimited number of) phones that you hook up are extensions within the office. For the outside world, the PBX is a telephone exchange entity.

OpenRG's PBX manages both Plain Old Telephone Service (POTS) and Voice over IP (VoIP) devices, utilizing VoIP lines to connect them to telephony service providers (proxies). Devices within OpenRG's PBX can freely communicate with each other, thus creating a cost-effective telephony environment.

OpenRG's PBX is available in two different versions—Home PBX and Full PBX. The Home PBX is a lighter version including only the necessities for running a basic PBX in your home, while the Full PBX features vast capabilities aimed at providing you with all aspects of a telephony exchange system. While this section covers the Full PBX, notes are incorporated for features that are not available with the Home PBX version.

Note: An external storage device is used by the Asterisk PBX to store large files, as well as files that you may wish to save between reboots. These files include voice mail greetings, voice mail messages, auto-attendant recordings and music on hold. When the external storage device is either missing, full or wrongly formatted, the voice mail, auto-attendant and music on hold features are disabled.

Asterisk configured as ATA supports the SIP and H.323 signaling protocols. This means that it can act as a SIP/H.323 client towards the world and communicate with SIP/H.323 service
providers (see Figure 35.1). Asterisk configured as PBX supports the SIP, H.323, and MGCP signaling protocols. It too can act as a SIP/H.323 client towards the world, but also as a server towards the LAN—it can communicate with SIP and MGCP IP phones. However, Asterisk cannot register with an external MGCP service provider.

Figure 35.1 Asterisk Signaling Protocols

On top of the signaling protocol, Jungo provides advanced call control features, real-time data handling, Quality of Service, DSP integration, and an extensive Web-based management, as described later in this chapter.

35.1 Compiling VoIP

OpenRG can be compiled either as ATA or PBX, but not as both. When compiling an image, use either MODULE_RG_ATA or MODULE_RG_PBX.

35.1.1 Compiling an ATA Image

When compiling an ATA image, you can use any combination of the following configuration flags, as long as each signaling protocol (i.e. SIP/H.323/MGCP) is defined only once:

- CONFIG_RG_VOIP_RV_SIP
- CONFIG_RG_VOIP_RV_H323
- CONFIG_RG_VOIP_RV_MGCP
- CONFIG_RG_VOIP_OSIP
- CONFIG_RG_VOIP_ASTERISK_SIP
For example, you can utilize your SIP implementation from oSIP, your H.323 implementation from Asterisk, and your MGCP implementation from Radvision: CONFIG_RG_VOIP_OSIP, CONFIG_RG_VOIP_ASTERISK_H323, and CONFIG_RG_VOIP_RV_MGCP. However, you cannot use two SIP implementations together: CONFIG_RG_VOIP_OSIP and CONFIG_RG_VOIP_ASTERISK_SIP.

35.1.2 Compiling a PBX Image

When compiling a PBX image, you can only use Asterisk configuration flags. Use any combination of the following configuration flags:

• CONFIG_RG_VOIP_ASTERISK_SIP
• CONFIG_RG_VOIP_ASTERISK_H323
• CONFIG_RG_VOIP_ASTERISK_MGCP_CALL_AGENT

35.2 The Signaling and Voice Stages

From the caller's point of view, placing a call on an IP network is quite similar to placing a call on a standard PSTN network. The caller dials, receives a ring back or busy tone, and starts a voice conversation. The main difference in VoIP is that packet rather than circuit switching is used. In general, a VoIP telephony conversation can be divided into two stages, the signaling stage and the voice stage.

The signaling stage involves dialing, call initiation, ringing, negotiations on media stream codecs, and packet switching initiation/termination. The voice stage occurs after a session has been initiated with one of the signaling protocols, and before termination. Voice is exchanged using the RTP/RTCP protocols, where each party encodes their respective voice packets, and sends them over UDP to the destination IP and port that was decided upon during the negotiation phase.

35.3 Reference Hardware Platforms

The OpenRG VoIP module targets the following application products:

• ATA Gateways
• Voice Gateways
• PBX Gateways

An ATA Gateway is a network device that connects analogue telephones to the IP network. It implements VoIP protocols, and has a DSP that handles digital voice signals. An ATA usually
has one Ethernet port that connects to the network through a DSL/Cable modem or router, and
one or more analogue telephone ports that enable the connection of regular phones. ATAs have
HTTP-based user interfaces and provide calling features such as hold, transfer and conference.

A Voice Gateway is an advanced ATA that also provides routing functionality, network
security, and usually includes an integrated broadband modem. Voice Gateways also have
HTTP-based user interfaces that provide advanced calling features such as hold, transfer,
conference, call blocking, and more.

A PBX Gateway is a network device enhanced with a private telephone exchange, which is
located in an end-user organization's premises. PBX connects between the internal telephones
of the organization, and also connects them to the public switched telephone network (PSTN)
via the trunk lines. In addition, PBX may provide such features as auto-attendant, day and night
mode handling for incoming calls, dial plan setup for the least cost routing, music on hold,
voicemail, etc.

35.4 Architecture

This section provides a description of the VoIP module and its internal workings. Since
OpenRG's VoIP module behaves as any other application coordinated by OpenRG, it may
interest you to refer to Chapter 15 for a detailed account of the OpenRG software architecture,
and specifically, the relationship between the Coordination Logic and its dependent tasks.

35.4.1 Asterisk Architecture

Figure 35.2  Asterisk VoIP Architecture

The Asterisk-based VoIP module architecture, depicted in Figure 35.2, describes the
relationship between its constituting elements. As aforementioned, the association between the
Configuration Database, WBM, Coordination Logic and Phone Logic is exactly the same as with any other OpenRG task (for more information, refer to Chapter 15).

However, the relationships between the remaining entities are different, due to the emphasis put on quality of service inherent to voice applications, and the integration with the Asterisk code. Asterisk is a GPL library that supplies PBX implementation and can also function as an ATA gateway. It runs as a separate multi-threaded process.

### 35.4.1.1 Coordination Logic

The *jasterisk* is an OpenRG entity, serving as 'glue' between VoIP and the rest of the system. It is responsible for extracting the VoIP parameters from the configuration database, and controlling the Asterisk process. The *jasterisk* controls the Asterisk process as follows:

1. Runs Asterisk when the entity is opened.
2. Terminates Asterisk when the entity is closed.
3. Reloads Asterisk when there is an IP or configuration change.

The following are methods that *jasterisk* uses to communicate with Asterisk:

1. **Configuration Files** – *jasterisk* generates Asterisk's configuration files, based on configuration data saved in `rg_conf`. For more information, refer to Section 35.5.4.

2. **Manager** – *jasterisk* uses Asterisk's built-in manager mechanism to communicate with Asterisk while it is running, as opposed to configuration files, which are prepared before Asterisk starts. The manager is used either to query Asterisk for up-to-date status information, or to make Asterisk perform certain actions. For more information, refer to Section 35.4.1.7.

3. **Command Line Interface** – Asterisk supplies a full set of CLI commands that can be used to perform management actions, extract information, and debug. For more information, refer to Section 35.4.1.8.

### 35.4.1.2 The jrtp Module

The *jrtp* module is a linux kernel module designed for optimal RTP to DSP path with minimal delay. It is responsible for:

- Sending Real-time Transport Protocol (RTP) packets with payload received from the DSP.
- Receiving RTP packets and passing the payload to the DSP.
- Sending and receiving Real-time Control Protocol (RTCP) packets
35.4.1.3 The jdsp Module

The jdsp kernel module (phone_mod) is responsible for hiding differences between different kinds of telephony hardware by wrapping the driver provided by the hardware vendor, and by providing a uniform API to the higher layer. jdsp consists of the following components:

**Generic DSP** A linux kernel character device that represents DSP and telephone ports for the upper layer, handles DSP generic logic, and interfaces with the RTP module.

**DSP Specific Layer** The DSP abstraction code designed to provide a standard DSP/telephony Megaco-like API for the upper layer. Each combination of the vendor's DSP/SLIC drivers requires its own instance of a DSP abstraction code. The API that is provided by the abstraction code to the upper layer is specified in Section 35.4.2.1.

35.4.1.4 The Vendor DSP Driver

The DSP/SLIC interface device driver provided by the hardware vendor must have the following functionality:

- Addressing API calls from the kernel space, as calling is later done (DSP specific) in the kernel space.
- Receiving media, preferably in RTP format, from the device. Vocoder for media must be configurable.
- Sending media, preferably in RTP format, to the device.
- Receiving notification of telephony events of type DTMF and hook/flash state changes.
- Playing of DTMF tones and telephony tones.
- Ring command must be supported and must allow sending of the caller ID.

Additional functionality may be supported, such as configuration of Acoustic algorithms, supporting third-party conference calls, type 2 caller ID, etc. The DSP abstraction code will be adapted to support these, if possible.

35.4.1.5 The chan_jdsp Asterisk Channel Driver

The chan_jdsp Asterisk channel driver implements an FXS interface for Asterisk. It uses jdsp as the API that allows it to control the underlying telephone hardware.

35.4.1.6 The chan_jfxo Asterisk Channel Driver

The chan_jfxo Asterisk channel driver implements an FXO interface for Asterisk. It uses jdsp as the API that allows it to control the underlying telephone hardware.
35.4.1.7 Asterisk Manager

Asterisk has a built-in manager mechanism. The Asterisk Manager allows a client program to connect to an Asterisk instance and issue commands or read PBX events over a TCP/IP stream. It listens for manager requests on a certain (configurable) port. Asterisk's manager supports different types of requests.

The request most often used by OpenRG is the **Command**. This is a special manager request, which enables you to issue CLI commands through the manager. You may issue a manager request at any time, since OpenRG maintains a constant connection with the manager at all times. As soon as Asterisk is up, OpenRG logs in to the manager, and the manager session is kept open until Asterisk is shut-down. For further reading on the Asterisk Manager, refer to [http://www.voip-info.org/wiki-Asterisk+manager+API](http://www.voip-info.org/wiki-Asterisk+manager+API).

35.4.1.8 Asterisk CLI

You can also manage OpenRG's Asterisk via its CLI. To enter Asterisk's CLI, perform the following:

1. In OpenRG's CLI, run the `system shell` command. The OpenRG prompt is replaced with the shell prompt.

2. Run the `asterisk -r` command. This will hook into the running Asterisk process, and the following Asterisk prompt appears:

```
Connected to Asterisk currently running on openrg (pid = 86)
openrg*CLI>
```

Use the following commands to manage Asterisk:

- **set verbose <level>** Sets level of verbose messages to be displayed. 0 means no messages should be displayed. Equivalent to `-v[v[v[...]]] on startup.

- **set debug <level>** Sets level of core debug messages to be displayed. 0 means no messages should be displayed. Equivalent to `-d[d[d[...]]] on startup.

- **sip debug** Enables dumping of SIP packets for debugging purposes.

  - **sip debug ip <host[:port]>** Enables dumping of SIP packets to and from a host.

  - **sip debug peer <peernamename>** Enables dumping of SIP packets to and from a host. Requires a peer to be registered.

- **h.323 debug** Enables dumping of H.323 packets for debugging purposes. This command is only available on OpenRG compiled with H.323 (using the `CONFIG_RG_VOIP_ASTERISK_H323=y` flag).

- **rtp debug <ip host[:port]>** Enable dumping of all RTP packets sent to and from the host.

You can see Asterisk's debugging messages both in OpenRG's CLI and WBM. To see the Asterisk messages in OpenRG's CLI, perform the following:
1. Set a filter for the Asterisk's debug messages.

```
OpenRG> log filter set voip.debug
```

2. Filter out other components from the log.

```
OpenRG> log filter set *.none
```

Asterisk's debug messages will automatically be printed out.

To see only the Asterisk messages in the WBM, go to the 'System Log' screen under the 'Monitor' menu item, and add a filter for the VoIP component, after setting the default 'All' filter to 'None' (for more information, refer to the 'Log' section of the OpenRG User Manual).

**35.4.2 Digital Signal Processor**

The Digital Signal Processor (DSP) is responsible for manipulating analog voice information that has been converted into a digital form. Packets read from the DSP contain voice data or telephony signaling, that needs to be sent out across the network. Voice data packets arriving from the network are written to the DSP, in order to be played back to the user. Jungo supports several DSPs and provides APIs for customer-specific DSP driver integration.

The following diagram describes the conversion of an analog signal to digital VoIP.

![VoIP RTP Packet Traffic](image)

**35.4.2.1 API for DSP Driver Integration**

The following API is implemented by the DSP abstraction code. The closer the driver is to this API concept, the simpler the required abstraction code.
35.4.2.1.1 Phone Ringing

- \texttt{int ring\_start (int Port, CallerID *CID)}
  This function is used to command a specific phone port to start ringing.
  \texttt{Port} – Specifies on which device the ringing should start.
  \texttt{CID} – Specifies the caller identity. The structure is:

  ```
  typedef struct {
    unsigned char Enable : 1
    unsigned char CallerIdTransmitType : 1
    unsigned char Unused : 6
    char Date [8]
    int ContactNumLen
    char ContactNum [32]
    int NameLen
    char Name [32]
  } CallerID
  ```

- \texttt{int ring\_stop (int Port)}
  This function is used to command a specific phone port to stop ringing.
  \texttt{Port} – Specifies on which device the ringing should be stopped. The API will make an ioctl call to disable ringing.

35.4.2.1.2 Tones

- \texttt{int tone\_start (int Port, int ToneId)}
  This function will command a specific phone port to play tone.
  \texttt{Port} – Specifies on which device the tone is played.
  \texttt{ToneId} – Specifies the tone to be played.

- \texttt{int tone\_stop (int Port)}
  This function will command a specific phone port to stop any tone played.
  \texttt{Port} – Specifies on which device to stop playing the tone.

35.4.2.1.3 Event Receiving

- \texttt{int event\_poll\_start (int Port)}
  This function will allow receiving events on a specific phone port.
  \texttt{Port} – Specifies the device to be polled.

- \texttt{int event\_poll\_stop (int Port)} – obsolete
  This function will force stopping of receiving events on a specific phone port.
  \texttt{Port} – Specifies the device on which polling is to be stopped.

- \texttt{int event\_get (int Port, int *Pressed)}
  This function will return the first event from the event queue, and remove it from the queue.
  \texttt{Port} – Specifies the device from which to get the events.
  \texttt{Pressed} – In case of a dtmf event, a non-zero value specifies that the key is pressed, and a zero value specifies that the key is released.

- \texttt{int hook\_state\_get (int Port)}
This function will return the current hook state.
Port – Specifies the phone port for which the the hook state should be returned.

35.4.2.1.4 Channel Configuration

• int voice_start (int Port, int ChannelNumber, int Codec, int Ptime, int Inband_dtmf)
  This function will open the channel for voice transmission.
  Port – Specifies the phone port from/to which to start the voice transmission.
  ChannelNumber – 0 or 1. 1 is used for third-party conference calls.
  Codec – Specifies the vocoder type.
  Ptime – Specifies the packetization time.
  Inband_dtmf – If zero, DTMF tones should be suppressed.

• int voice_stop (int Port, int ChannelNumber)
  This function will stop voice transmission.
  Port – Specifies the phone port.
  ChannelNumber – Specifies the channel number to close.

35.4.2.1.5 Media Streaming

• void voip_dsp_packet_received(int line, int channel, u8 *buf, int len);
  This function is called to send the RTP buffer that originated from the network and is
  addressed to a specific phone port and channel.
  Port – Specifies the phone port to which the buffer should be sent.
  Channel – Specifies to which of the two possible channels of this port the buffer is meant.
  Buf – The RTP buffer to be played.
  Len – The length of data in the RTP buffer, including the RTP header.

• int voice_write(int line, int channel, u8 *buf, int len);
  This function is a called by the DSP specific layer in order to send the RTP buffer originated
  from the DSP and addressed to a network RTP stream.
  Port – Specifies the phone port from which the buffer is coming.
  Channel – Specifies from which of the two possible channels of this port the buffer
  originated.
  Buf – The RTP buffer to be sent.
  Len – The length of data in the RTP buffer, including the RTP header.

35.4.2.1.6 Init Sequence

• int device_init (int Port)
  This function is called to initialize the device. No API function will be used prior to this
  function.
  Port – Specifies the device.
Note: The device may be restarted several times after boot was completed, using subsequent calls of Uninit and Init.

- \texttt{int device\textunderscore uninit (int Port)}
  
  This function will disable the device.
  
  \texttt{Port} – Specifies the device.

### 35.4.3 RTP/RTCP

VoIP signaling logic runs in user mode, whereas the DSP driver and network stack run in kernel mode. Having the user mode application mediate the stream between the DSP and the network, has proven to be inefficient in high network traffic stress. In order to have the RTP traffic compete on equal terms with other network traffic, this module was completely integrated within the kernel.

### 35.4.4 SIP ALG

OpenRG's telephony system utilizes the Session Initiation Protocol (SIP) and Session Description Protocol (SDP), in order to connect to a remote VoIP server for transmitting and receiving the RTP traffic. The structure of packets transmitted in the SIP client-server sessions has been defined in RFC 3261, and it involves private IP addresses, UDP port numbers, and SIP/SDP data entries embedded into the packet payload. This packet structure poses a challenge to the NAT.

In order for the SIP and RTP traffic to pass through OpenRG's firewall (with the NAT enabled), it is necessary for the firewall to be able to access the payload of outgoing packets, and translate the embedded private addresses into the correct public addresses (and vice versa for incoming packets). The agent within the firewall that makes the necessary changes in a packet's payload is referred to as SIP Application Level Gateway (ALG). OpenRG's SIP ALG interoperates with various network elements, such as a local server, SIP proxy, Simple Traversal of UDP through NAT (STUN) client, SIP registrar, and outbound proxy—either residing in the LAN or WAN.

Besides performing NAT on SIP packets, SIP ALG prevents port collisions, by dynamically redistributing the SIP packets between unoccupied UDP ports. In addition, SIP ALG translates the IP addresses and port numbers of the paired RTP/RTCP packets, in order to allow the voice traffic across the firewall, once an end-to-end session is established. After establishing a voice session, OpenRG's SIP ALG keeps track of the SIP dialog between the VoIP user agents, by inspecting the \texttt{'Call-ID'}, \texttt{'From'}, and \texttt{'To'} header fields of the SIP packets.

SIP ALG translates the IP and port values located in the following header fields:

- Invite
- Subscribe
- Contact
Voice over IP (VoIP)

- Via/Received
- Route
- Record Route
- Reply-To
- Refer-To/Referred-By
- PRACK

In addition, SIP ALG processes the SDP data contained in the SIP body.

The SIP ALG mechanism involves the following applications:

- Parser – Handles parsing of the textual data embedded in a SIP packet's payload. The parser communicates via callbacks with one of the following clients.
- OpenRG's NAT – Communicates with the parser to enter the IP/port changes in a SIP packet.
- OpenRG's firewall – Communicates with the parser prior to opening a pinhole for transmission of the RTP traffic.
- A unit test utility – Runs as a stand-alone local application, emulating the NAT and firewall callbacks. Compile this utility from test_sip.c located in pkg/firewall/kernel/alg. The unit test contains more than two hundred SIP packet samples, and it is aimed at preventing regression that might be introduced while changing the code of the SIP parser.

### 35.4.5 Quality of Service

Quality of service issues were addressed by implementing the following into OpenRG's networking and VoIP modules:

- RTP/RTCP modules are implemented in the kernel, reducing latencies due to user mode/kernel mode task switching.
- Packets are sent with IP TOS/DiffServ bits set.
- Packets are sent with a high priority setting to the Ethernet driver's high priority queue.
35.5 Modifying the Voice Module

35.5.1 Integrating Additional DSPs

To integrate the OpenRG VoIP module with additional DSPs, Jungo provides a device driver "glue" layer that standardizes communication with the Phone Logic module. The integration of the DSP module is done by implementing the functions voip_dsp_register_device and voip_dsp_packet_received. These functions make use of the voip_dsp_dev_ops_t structure of callback functions, which must also be implemented. The functions and structure are located in rg/pkg/voip/dsp/phone_modules.h. The functions defined in the voip_dsp_dev_ops_t structure use line and channel parameters:

- The parameter line is a 0-based index of the FXS.
- The parameter channel is a channel within an FXS and may be either 0 or 1.
- Channel 1 is used in case of a 3-way call.

Following is a line-by-line explanation of the voip_dsp_dev_ops_t structure to be implemented by every DSP abstraction layer. Each explanation is followed by the relevant code.

- Tells the DSP to play the RTP packet in 'buf' on 'channel' of 'line':
  ```
  int (*voice_write)(int line, int channel, u8 *buf, int len);
  ```

- Returns (POLLIN | POLLRDNORM) if a new event is available from 'line', 0 otherwise. Analogous to the kernel poll() operation. See include/linux/fs.h. Note that in Linux OS, 'table' is of type poll_table_struct * and 'filp' is of type struct file *:
  ```
  int (*event_poll)(int line, void *filp, kos_poll_node_t table);
  ```

- DEPRECATED:
  ```
  int (*event_poll_stop)(int line, kos_poll_node_t table);
  ```

- Returns the first event in 'line' event queue. If the event is a key press, then upon return, 'pressed' is set to 1 if the key was pressed and 0 if the key was released:
  ```
  phone_key_t (*event_get)(int line, int *pressed);
  ```

- Gets hook state of 'line'. Returns 1 if 'line' is off-hook:
  ```
  int (*hook_state_get)(int line);
  ```

- Starts ringer on 'line'. 'cid' specifies the caller ID information that should be sent to the phone (may be NULL):
  ```
  int (*ring_start)(int line, phone_caller_id_t *cid);
  ```

- Stops ringer on 'line':
  ```
  int (*ring_stop)(int line);
  ```
• Starts playing a call waiting tone. 'cid' specifies the caller ID information that should be sent to the phone (may be NULL):

```c
int (*call_waiting_alert_start)(int line, phone_caller_id_t *cid);
```

• Stops playing a call waiting tone on 'line':

```c
int (*call_waiting_alert_stop)(int line);
```

• Starts playing 'tone' on 'line':

```c
int (*tone_start)(int line, phone_tone_t tone);
```

• Stops the currently playing tone on 'line':

```c
int (*tone_stop)(int line);
```

• Initializes 'line':

```c
int (*init)(int line);
```

• Un-initializes 'line':

```c
int (*uninit)(int line);
```

• DEPRECATED:

```c
int (*set)(int line, int cmd, int param);
```

• Tells the DSP to start generating and receiving audio frames on the specified 'line' and 'channel', using the specified 'codec' and 'ptime' (packetization time). If 'suppress_dtmf' is set to 1, the DSP should suppress generation of audio packets while a DTMF is detected:

```c
int (*voice_start)(int line, int channel, rtp_payload_type_t codec,
int ptime, int suppress_dtmf);
```

• Tells the DSP to stop generating audio frames on specified 'line' and 'channel':

```c
int (*voice_stop)(int line, int channel);
```

• Configures DSP-specific features. 'data' points to a private DSP structure of 'size' bytes that was filled by the application and can be parsed by the DSP abstraction layer:

```c
void (*configure_dsp)(void *data, int size);
```

The `voip_dsp_register_device` function registers a DSP abstraction layer that implements the operations in 'ops'. The `voip_dsp_packet_received` function is called by the DSP abstraction layer once it has a new RTP frame ready to be sent, from 'channel' of 'line'.

### 35.5.2 Adding New Features to Asterisk

The Asterisk source code is fully provided, as it consists of open-source libraries (LGPL/GPL). You can modify the code to add new Voice features.
35.5.2.1 Using Existing Features

OpenRG uses only part of the many features that Asterisk offers. Features that are not in use can be enabled through the Asterisk configuration files. Sample files, along with feature explanations, can be found under `rg/pkg/voip/asterisk/configs`.

35.5.2.2 Changing Existing Channels and Applications

You can use the existing channels and applications to modify them for your own needs. The channels that are currently in use by OpenRG are:

- chan_sip
- chan_h323
- chan_mgcp (PBX only)
- chan_jdsp
- chan_features

The applications that are currently in use by OpenRG are:

- app_dial
- app_cut
- app_voicemail (PBX only)
- app_record (PBX only)
- app_playback (PBX only)
- app_macro (PBX only)

35.5.2.3 Adding New Channels and Applications

New channels and applications can be added to the existing Asterisk code to make Asterisk support new protocols and have new abilities. You can do this in several ways:

1. Compile and link channels/applications that are included in the Asterisk code provided with OpenRG and are not currently used by OpenRG.

2. Integrate new channels/applications into the Asterisk code provided with OpenRG. These can be acquired from newer versions of Asterisk or from any Asterisk-related project.

3. Write your own channels/applications and integrate them into the Asterisk code provided with OpenRG. Refer to `rg/pkg/voip/asterisk/doc` for information about implementing
channels and applications. This library contains documents supplied by Asterisk, such as `channel.txt`, which explains how to implement a channel, README files for each application, etc.

### 35.5.2.4 Debugging Asterisk Using jgdb

You can use `gdb` to debug the Asterisk external process. In the OpenRG tree, enter the following:

```
$ cd
$ make -C pkg/gdb/gdb/gdbserver/ && cp -v build/pkg/gdb/gdbserver/gdbserver /tftpboot/ && jstrip /tftpboot/gdbserver
```

In the OpenRG prompt, enter the following:

```
OpenRG> system shell
/ # tftp -g 192.168.1.10 -r gdbserver
/ # chmod +x gdbserver
/ # ps
/ # ./gdbserver :9999 --attach [pid of the first asterisk process in ps output]
```

On your PC, perform the following:

1. Open the `~/.gdbinit` file, and enter the following:

   ```
   # set search path for dynamic libraries
define dynlib
     set solib-absolute-prefix /dev/null
     set solib-search-path
     $arg0/pkg/build/lib:$arg0/pkg/ulibc/lib:$arg0/pkg/lib:$arg0/pkg/freeswan/lib:$arg0/pkg/openssl/crypto:$arg0/pkg/voip/asterisk/channels:$arg0/pkg/voip/asterisk/apps:$arg0/pkg/voip/asterisk/res:$arg0/pkg/voip/asterisk/codecs:$arg0/pkg/voip/asterisk/formats:$arg0/pkg/voip/asterisk/pbx:/usr/local/openrg/i386-jungo-linux-gnu/i386-jungo-linux-gnu/lib:$arg0/pkg/openssh
   end

define dynlib_auto
  shell echo "dynlib <OpenRG tree's full path>/build" > .dynlib
  source .dynlib
  shell rm .dynlib
end

# Remote connection using a network interface
define trn
dynlib_auto
  target remote 192.168.1.1:9999
  handle SIGSTOP nostop
end
document trn
  Connect to a gdbserver running on OpenRG using a network connection. The command tells gdb to look for `.so` files in the OpenRG tree. gdbserver should be run on the target as follows:
  `gdbserver :9999 <additional parameters>`
end

   Note: The long path placed after `set solib-search-path` must be entered as one line (without line breaks).
```

2. Run the gdb client:

   ```
   $ jgdb build/pkg/voip/asterisk/asterisk
   ```

For more information on debugging OpenRG, refer to Chapter 20.
35.5.3 Web-based Interface Tailoring

The OpenRG VoIP module has a Web-based interface that provides control over its different features. This interface can be easily customized using OpenRG's WBM libraries and SSI directives. For more information, refer to Chapter 28.

35.5.4 Configuration File Entries

OpenRG's VoIP configuration file entries are listed in the OpenRG Configuration Entries Guide. They are used to configure telephony parameters and can be modified according to the desired configuration. In addition to using OpenRG's VoIP configuration file entries, Asterisk also uses its own configuration files. When Asterisk starts up, it extracts all the information it needs from Asterisk-specific configuration files. These are *.conf files that are kept in directory /etc/asterisk. Full explanations of these files can be found on the Internet.

Whenever a change occurs in the VoIP section of rg_conf, OpenRG re-writes the Asterisk configuration files according to the rg_conf changes while Asterisk is running, and then during "reload" Asterisk re-reads them. For further reading on Asterisk configuration files, refer to http://www.voip-info.org/wiki-Asterisk+config+files.
Secure Socket Layer (SSL) VPN

OpenRG includes a number of Secure Socket Layer (SSL) Virtual Private Network (VPN) applications. This chapter describes in detail the process of adding a new SSL VPN application (section Section 36.1).

Note: The new SSL application must be a Java application, supplied with its source code so you will be able to modify the code to communicate with OpenRG.

This chapter also explains how to modify jar files location (Section 36.2).

36.1 Adding a New SSL VPN Application

Follow these steps to add a new SSL VPN application:

1. Modify the application to use rg.RGSocket instead of a regular socket:
   1. Locate the point in the application source code in which a socket is created.
   2. Instead of a regular java.net.Socket, you should create an rg.RGSocket. Unlike java.net.Socket constructor, which expects host and port as its arguments, rg.RGSocket requires three extra arguments:
      • Proxy host
      • Proxy port
      • Session ID
3. The values for these arguments for the rg.RGSocket should be supplied to the application when it is invoked by the WebStart mechanism, therefore, you should also modify the application to expect these three extra arguments, and then propagate their values to the socket creation routine.

2. Recompile the application, thus creating a new application jar file.

3. Sign the new jar file with the same keys that signed **rg.jar**. Consult formal Java tutorials of how to sign jar files.

Note: If the application uses external packages given as jar files, you should also sign these jar files with the same keys.

4. Upload the new signed jar files (one or more) to the Web site where the jar files should reside. If you wish to change the location for these files, refer to **Section 36.2**.

5. Add the application to **rg/pkg/include/enums.h** in **ssl_vpn_app_t** enum:

```c
ENUM_START(ssl_vpn_app_t)
  /* Telnet */
  ESTR(SSL_VPN_APP_TELNET, 0, "telnet")
  /* Remote Desktop */
  ESTR(SSL_VPN_APP_RDP, 1, "remote_desktop")
  /* FTP */
  ESTR(SSL_VPN_APP_FTP, 2, "ftp")
  /* CIFS */
  ESTR(SSL_VPN_APP_CIFS, 3, "cifs")
  /* VNC */
  ESTR(SSL_VPN_APP_VNC, 4, "vnc")
ENUM_END
```

6. Add the application name to the **ssl_vpn_wbm_lang.csv** language file, located in **rg/pkg/ssl_vpn/wbm/**. This is the name that will appear in the drop-down menu of the applications.

7. Add an entry to the **app2str** array in **rg/pkg/ssl_vpn/wbm/ssl_vpn_advanced.c**.

```c
static code2str_t app2str[] = {
  {SSL_VPN_APP_TELNET, lang:T_PTR(Ttelnet)},
  {SSL_VPN_APP_RDP, lang:T_PTR(Tremote_desktop)},
  {SSL_VPN_APP_FTP, lang:T_PTR(Tftp)},
  {SSL_VPN_APP_CIFS, lang:T_PTR(Tcifs)},
  {SSL_VPN_APP_VNC, lang:T_PTR(Tvnc)},
  [-1]
};
```

8. Add an entry to the **app_options** array in **rg/pkg/ssl_vpn/wbm/ssl_vpn_advanced.c**.

```c
static app_options_t app_options[] = {
  { SSL_VPN_APP_TELNET, 0, 0, 0, 0, 0, 0, &p_gen_app_options, &s_gen_app_options },
  { SSL_VPN_APP_RDP, 1, 1, 1, 0, 1, 0, &p_gen_app_options, &s_gen_app_options },
  { SSL_VPN_APP_FTP, 1, 1, 1, 0, 1, 0, &f_gen_app_options, &s_f_gen_app_options },
  { SSL_VPN_APP_CIFS, 0, 1, 1, 0, 0, 1, &p_gen_app_options, &s_gen_app_options },
  { SSL_VPN_APP_VNC, 1, 0, 1, 0, 0, 0, &p_gen_app_options, &s_gen_app_options },
};
```
There are several fields like user name and password that are common to all applications. You should set them to 1 if and only if your new application requires these fields. If you set them to 0, they will not appear. If your application requires special fields that are not common to other applications, you should also set print and scan callbacks that will call p_gen_app_options() and s_gen_app_options() respectively, and will print and scan these special fields.

9. Add a handler to `rg/pkg/ssl_vpn/webstart/webstart_cgi.c`. Name the handler `<name>_handler`, where `<name>` is the name of the new application. The role of the handler is to set the environment and arguments for the execution of the application. The following fields must be set:

- `params->title` Should contain the name of the application.
- `params->desc` Should contain the description of the application.
- `params->main_jar` Should contain the name of the main jar file of the application.
- `params->main_class` Should contain the name of the class that starts the application (the program's entry point).
- `params->args` Should be set to contain all the application-specific arguments. In order that each argument will have its own line in `params->args`, you should use:

```c
lines_add_multi(&params->args, ..., NULL)
```

For example, suppose the Remote Desktop (RDP) application expects the following arguments: 

```
-l "DEBUG"  -T title  -h proxy_host:proxy_port:session_id  <IP address>
```

In this case, we should pass each argument (token) as a separate line:

```c
lines_add_multi(&params->args, "-l", "DEBUG", "-T", title, "-h", ...);
```

Note: The list of lines must be NULL terminated.

Note: Every application requires the proxy host, proxy port and session ID arguments (for rg.RGSocket construction), so these arguments must be passed to the application.

If your application requires external packages, given as jar files, you should provide the list of these jar files in `params->other_jars` (again, using `lines_add_multi()` function call).

10. Enable the handler by adding an entry to `handlers[]` array in `rg/pkg/ssl_vpn/webstart/webstart_cgi.c`.

```c
static webstart_handler_t handlers[] = {
    { SSL_VPN_APP_TELNET, &telnet_handler },
    { SSL_VPN_APP_RDP, &rdp_handler },
    { SSL_VPN_APP_FTP, &ftp_handler },
    { ...
```
36.2 How to Modify jar Files Location

The jar files location is determined by the **rg_jars_codebase** variable in **rg/pkg/ssl_vpn/webstart/rg_jars_codebase.c**. Typically, this variable is set to Jungo's Web site. You can modify it to be the URL of your Web site (e.g. "http://mywebsite.com/jars"). After modifying this file, recompile the tree. Then subsequent downloads will start from your new Web site address.
37

TR-069 and Extensions

37.1 Overview

TR-069 is a WAN management protocol intended for communication between Customer Premise Equipment (CPE) and an Auto-Configuration Server (ACS). It defines a mechanism that encompasses secure auto configuration of a CPE, and also incorporates other CPE management functions into a common framework.

Such a protocol is useful, for example, for remotely and securely controlling gateways by the ISP.

![Figure 37.1 TR-069 CPE WAN Management Protocol](image)

The TR-069 protocol allows an ACS to provision a CPE, or a collection of CPEs grouped according to various criteria. The provisioning mechanism includes specific provisioning parameters and a general mechanism for adding vendor-specific provisioning capabilities as needed.

The provisioning mechanism allows CPE configuration at the time of its initial connection to the broadband access network, and the ability to re-configure at any subsequent time. This includes support for asynchronous ACS-initiated CPE re-configuration. TR-069 defines several
Remote Procedure Call (RPC) methods, as well as a large number of parameters, which may be set or read. Some of these methods and parameters are defined as mandatory. The TR-069 protocol is expanded by the following extensions:

- TR-098 is the DSLHome Internet Gateway Device Version 1.1 Data Model used for Quality of Service (QoS).
- TR-104 is the DSLHome Internet Gateway Device Version 1.1 Data Model used for Voice over IP (VoIP).
- TR-111 is the DSLHome Internet Gateway Device Version 1.1 Data Model used for Voice over IP (VoIP).

### 37.2 Architecture Implementation

#### 37.2.1 File Structure

The TR-069/TR-098/TR-104 code resides in `rg/pkg/dslhome`. The code is divided as follows:

- `rg/pkg/dslhome/*.[hc]` – These files are the generic protocol implementation ("engine") of TR-069/TR-098/TR-104.
- `rg/pkg/dslhome/test_files/` – Files in this directory are used for unit-test of the generic code.
- `rg/pkg/dslhome/wbm/` – These files implement the Web-Based Management (WBM) section related to TR-069/TR-098/TR-104, i.e. the check box that appears in the 'Remote Administration' page.
- `rg/pkg/dslhome/main/` – These files implement all parameters as well as system-dependent Remote Procedure Calls (RPC), e.g. Download.

#### 37.2.2 Parameters

Parameters are implemented by writing a `cwmp_param_handler_t` structure for each parameter, and registering it with `cwmp_params_register()`. Once registered, the engine will support all parameter RPCs for it. The `cwmp_param_handler_t` structure is defined in `rg/pkg/dslhome/cwmp_params.h`. This section describes the name, type and notification fields of the `cwmp_param_handler_t` structure. Refer to the comments written in the structure's code for the explanation of the remaining fields.

1. **Parameter Name** – The 'name' field of the `cwmp_param_handler_t` structure is a string of the full name of the parameter, e.g. `InternetGatewayDevice.ManagementServer.PeriodicInformEnable`. To avoid using the same literal strings, the following preprocessor macro conventions are used:
• Each parameter name is defined by a macro, e.g.:
  
  #define PON_internet_gateway_device "InternetGatewayDevice."

• The macro uses the macro of the parameter's parent, e.g.:
  
  #define PON_device_info PON_internet_gateway_device "DeviceInfo."

• All simple parameter names start with "PN_", followed by the TR-069/TR-098/TR-104 name, where each capital letter is changed to "_" followed by the lower-case letter.

2. Parameter Type – There are two types of parameters: simple and non-simple.

- Simple types correspond to a parameter that has a value, e.g.
  "InternetGatewayDevice.ManagementServer.PeriodicInformEnable". It is a leaf in the parameter tree. Simple types are string, integer, unsigned integer, boolean, date and base-64 binary. The naming convention is PON_XXX representing the parameter name.

- Non-simple types correspond to an inner node (i.e. not a leaf) in the parameter tree. Names of non-simple typed parameters end with a period ("."). Non-simple types are further divided to multi-instance object and object types:
  
  • Multi-instance object types are parameters that contain indexed children, e.g.
    "InternetGatewayDevice.LANDevice.". Note that all immediate children of these parameters are in the form of "InternetGatewayDevice.LANDevice.%u", where %u represents an unsigned non-zero index. The naming convention is PMION_XXX representing the parameter multi-instance object name.

  • Object type are parameters that contain other parameters, e.g.
    "InternetGatewayDevice.LANDevice.%u". The naming convention is PON_XXX representing the parameter object name.

3. Notifications – All TR-069/TR-098/TR-104-defined parameters that are required to initiate active notifications when their value changes, or are required to be in each Inform message are already implemented. This means that new parameters will never use CWMP_NOTIFY_FORCE_INFORM or CWMP_NOTIFY_FORCE_ACTIVE in their notification field. CWMP_NOTIFY_DISABLED may be used to disable notification requests from the ACS on the parameter. This is typically used to prevent notifications on statistics parameters.

You will find the following RPCs and functions useful:

1. AddObject and DeleteObject Implementation – AddObject and DeleteObject are RPCs used to add and delete instances of the parameters.

   • When an AddObject RPC is received, the following is performed:

     a. The 'add' callback of the parent object is called to create the new instance.
b. The 'add' callback of each immediate child of the new instance is invoked. This may be used to initialize the fields of the new instance to default values.

Note: In currently implemented parameters, the 'add' callback of the parent creates a new, already initialized, instance; therefore all 'add' callbacks of the children are set to NULL.

• When an DeleteObject RPC is received, the 'delete' callback of all children is invoked in depth-first order (the delete callback of the referred parameter is invoked last).

Note: In currently implemented parameters, the 'delete' callback of the referred parameter removes all children parameters, therefore all 'delete' callbacks of the children are set to NULL.

2. The is_exists() Callback – This callback is used to check if a specific instance of the parameter exists. As an example, the is_exists() callback of "InternetGatewayDevice.WANDevice.%u.WANDSLConnectionManagement" verifies that the device associated with "InternetGatewayDevice.WANDevice.%u." is a DSL device.

Note: The is_exists() callback must never be NULL.

For parameters that always exist (e.g. "InternetGatewayDevice.WANDevice."), cwmp_params_always_exists() should be used.

3. General Helper Functions – The following parameter helper functions are defined in rg/pkg/dslhome/main/mt_cwmp_params.h:

• mt_cwmp_param_gen_get()

• mt_cwmp_param_gen_set()

These functions implement read and write operations from and to a specific rg_conf path. No interpretation or modification is performed on the stored/retrieved data. The functions expect to find the path in the 'data' field of the parameter's cwmp_param_handler_t structure. An example of use can be found in "InternetGatewayDevice.ManagementServer.PeriodicInformEnable".

37.2.3 Adding a New Parameter

1. Create an array of type cwmp_param_handler_t, with an entry for each new parameter. The last element of the array should have a NULL name. For example, to create a new parameter "InternetGatewayDevice.DemoObject.DemoInt" (note that "InternetGatewayDevice." is already implemented), the following new parameters should be added:
Object type parameter: "InternetGatewayDevice.DemoObject."

Simple integer type Parameter: "InternetGatewayDevice.DemoObject.DemoInt"

2. Replace <array_name> with the name of the new parameter array in the mt_cwmp_params_register() function, found in rg/pkg/dslhome/main/mt_cwmp_params.c. Do this by adding the following line to the mt_cwmp_params_register() function:

```c
    cwmp_params_register(t, <array_name>);
```

3. Replace <array_name> with the name of the new parameter array in the mt_cwmp_params_unregister() function, found in rg/pkg/dslhome/main/mt_cwmp_params.c. Do this by adding the following line to the mt_cwmp_params_unregister() function:

```c
    cwmp_params_unregister(t, <array_name>);
```

### 37.3 Jungo Extensions

Some OpenRG features (e.g. Web Server) cannot be configured directly by the TR-069 protocol. In order to resolve this, a new RPC has been developed by Jungo, which controls the CLI (that configures rg_conf). The following is an example of this RPC:

**X_JUNGO_COM_RGCommand**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGCommand</td>
<td>string</td>
<td>Any command that can be sent via the CLI. For example, <code>system ver</code>.</td>
</tr>
</tbody>
</table>

**Table 37.1 X_JUNGO_COM_RGCommand**

This RPC is sent to OpenRG from the ACS. OpenRG in return sends a status number and a result string, similar to the return data of a CLI command:

**X_JUNGO_COM_RGCommandResponse**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>integer</td>
<td>This integer indicates the command's outcome. It is the same integer that is printed in the CLI's command output (less the text). For example, &quot;Returned 0&quot; means that the command was successful.</td>
</tr>
<tr>
<td>Result</td>
<td>string</td>
<td>The command's output, supplying informative data relevant to the command.</td>
</tr>
</tbody>
</table>

**Table 37.2 X_JUNGO_COM_RGCommandResponse**
OpenRG supports the following RPCs:

- GetRPCMethods
- SetParameterValues
- GetParameterValues
- GetParameterNames
- SetParameterAttributes
- GetParameterAttributes
- AddObject
- DeleteObject
- Download
- Upload
- FactoryReset
- ScheduleInform
- Reboot
- X_JUNGO_COM_RGCommand
- SetVouchers
- GetOptions

For the full list of CLI commands, refer to Chapter 27.

### 37.4 JACS Implementation

Jungo Auto-Configuration Server (JACS) is a gateway management application. It utilizes the TR-069 protocol to control and customize the gateway's operation. JACS is activated from a shell. Its executable is located at `rg/build/pkg/dslhome/jacs`. To establish communication between JACS and OpenRG, perform the following:

1. Verify that your PC is connected to OpenRG's LAN port.
2. Open a shell and change to `rg/build/pkg/dslhome/jacs`.
3. Run the JACS executable.
4. In the **jacs** context, enter the following command:

```
jacs$ listen 5555
```

This command instructs JACS to listen on port 5555 of your PC. Let's assume that your PC's LAN address is `192.168.1.10`.

5. From OpenRG's CLI, execute the following commands:

```
OpenRG> conf set cwmp/acs_url http://192.168.1.10:5555/acs
OpenRG> conf set cwmp/enabled 1
OpenRG> conf reconf 1
```

**Note:** The `cwmp` part of the command means CPE WAN Management Protocol, which is another name of TR-069.

After these commands are executed, JACS will receive an **Inform** message. This message, as well as other OpenRG responses, are displayed in XML format, without being parsed. Therefore, you should be acquainted with XML to read the data.

6. Reply to the **Inform** message as follows:

```
jacs$ rpc InformResponse MaxEnvelopes=1
```

From this point on, JACS–OpenRG communication is an interactive session, where you enter commands and receive replies. The following are the supported commands:

- **get_params** `<param 1> <param 2> <param 3> ...` Returns values of the given parameters.
- **set_params** `<param 1=val 1> <param 2=val 2> <param 3=val 3> ...` Assigns the given values to the parameters.
- **get_attribs** `<param>` ... Returns attributes of a parameter.
- **set_attribs** `<param1_name> <param1_notification_change> <param1_notification> <param1_access_list_change> <param1_access_list> | None` Assigns attributes to a parameter.

**Note:** The 'None' part of the command is a non-standard entry, which means that no entities are present, and the Write access is allowed only from JACS.

- **user** `<username> <password>` Set username and password for the HTTP authentication.
- **rpc** `<rpc_method> <param 1=val 1> <param 2=val 2> <param 3=val 3> ...` Executes RPC methods with a given set of values, assigned to the method parameters. The following are the supported RPC methods and their functions:
  - **GetRPCMethods** Gets a list of supported methods.
  - **GetParameterNames** Gets a list of parameter names.
  - **InformResponse** Sends InformResponse message.
  - **AddObject, DeleteObject** Adds or deletes objects.
  - **Reboot** Reboots OpenRG.
TransferCompleteResponse  Sends the TransferCompleteResponse message.

Download  Downloads files to OpenRG (image/configuration file, etc.). If you enter these commands slowly, the session may timeout and you will have to begin a new one. For this purpose, enter the following CWMP command in OpenRG’s CLI:

```
OpenRG> cwmp session_start
```

In order to terminate a session, you must send an empty message to OpenRG, as follows:

```
jacs$ rpc
```

Note that in some cases, you will have to send an empty message twice in order to terminate the session.

### 37.5 TR-069 Parameter List

The DSL Forum TR-069 Technical Report specifies the TR-069 full parameter list, including the parameter type and description. For complementary information, refer to this document, which is available at [http://www.broadband-forum.org/technical/trlist.php](http://www.broadband-forum.org/technical/trlist.php).

The following is the list of the TR-069 objects supported by OpenRG. Objects that are partly supported have a detailed list of their implemented parameters. If not specified, the listed object is fully implemented.

- InternetGatewayDevice.
  - LANDeviceNumberOfEntries
  - WANDeviceNumberOfEntries

- InternetGatewayDevice.ManagementServer.
  - URL
  - Username
  - Password
  - PeriodicInformEnable
  - PeriodicInformInterval
  - PeriodicInformTime
  - ParameterKey
  - ConnectionRequestURL
  - ConnectionRequestUsername
  - ConnectionRequestPassword
- ManageableDeviceNumberOfEntries
- ManageableDeviceNotificationLimit
- InternetGatewayDevice.ManagementServer.ManageableDevice.{i}.
  - ManufacturerOUI
  - SerialNumber
  - ProductClass
- InternetGatewayDevice.DeviceInfo.
  - Manufacturer
  - ManufacturerOUI
  - ModelName
  - Description
  - ProductClass
  - SerialNumber
  - HardwareVersion
  - SoftwareVersion
  - SpecVersion
  - ProvisioningCode
  - UpTime
  - DeviceLog
- InternetGatewayDevice.IPPingDiagnostics.
  - DiagnosticsState
  - Interface
  - Host
  - NumberOfRepetitions
  - Timeout
- DataBlockSize
- DSCP
- SuccessCount
- FailureCount
- AverageResponseTime
- MinimumResponseTime
- MaximumResponseTime

- InternetGatewayDevice.LANDevice.{i}
  - LANEthernetInterfaceNumberOfEntries
  - LANUSBInterfaceNumberOfEntries
  - LANWLANConfigurationNumberOfEntries

- InternetGatewayDevice.LANDevice.{i}.LANHostConfigManagement.
  - DHCPServerEnable
  - MinAddress
  - MaxAddress
  - SubnetMask
  - DNSServers
  - AllowedMACAddresses
  - DHCPRelay
  - DomainName
  - IPRouters
  - DHCPLeaseTime

- InternetGatewayDevice.LANDevice.{i}.Hosts.
  - HostNumberOfEntries

- InternetGatewayDevice.LANDevice.{i}.Hosts.Host.{i}.
- IPAddress
- AddressSource
- MACAddress
- HostName
- InterfaceType
- Active

- InternetGatewayDevice.LANDevice.{i}.LANEthernetInterfaceConfig.{i}.
  - Enable
  - Status
  - MACAddress
  - MaxBitRate

- InternetGatewayDevice.LANDevice.{i}.LANEthernetInterfaceConfig.{i}.Stats.
  - BytesSent
  - BytesReceived
  - PacketsSent
  - PacketsReceived

- InternetGatewayDevice.LANDevice.{i}.WLANConfiguration.{i}.
  - Enable
  - Status
  - BSSID
  - MaxBitRate
  - Channel
  - SSID
  - BeaconType
  - MACAddressControlEnabled
- Standard
- WEPKeyIndex
- KeyPassphrase
- WEPEncryptionLevel
- BasicEncryptionModes
- BasicAuthenticationMode
- WPAEncryptionModes
- WPAAuthenticationMode
- IEEE11iEncryptionModes
- IEEE11iAuthenticationMode
- PossibleChannels
- BasicDataTransmitRates
- OperationalDataTransmitRates
- PossibleDataTransmitRates
- InsecureOOBAccessEnabled
- BeaconAdvertisementEnabled
- RadioEnabled
- AutoRateFallBackEnabled
- LocationDescription
- RegulatoryDomain
- TotalPSKFailures
- TotalIntegrityFailures
- ChannelsInUse
- DeviceOperationMode
- DistanceFromRoot
- PeerBSSID
- AuthenticationServiceMode
- TotalBytesSent
- TotalBytesReceived
- TotalPacketsSent
- TotalPacketsReceived
- TotalAssociations
- InternetGatewayDevice.LANDevice.{i}.WLANConfiguration.{i}. AssociatedDevice.{i}.
  - AssociatedDeviceMACAddress
  - AssociatedDeviceIPAddress
  - AssociatedDeviceAuthenticationState
  - LastRequestedUnicastCipher
  - LastRequestedMulticastCipher
  - LastPMKId
- InternetGatewayDevice.LANDevice.{i}.WLANConfiguration.{i}. WEPKey.{i}.
  - WEPKey
- InternetGatewayDevice.LANDevice.{i}.WLANConfiguration.{i}. PreSharedKey.{i}.
  - PreSharedKey
  - KeyPassphrase
  - AssociatedDeviceMACAddress
- InternetGatewayDevice.WANDevice.{i}.
  - WANConnectionNumberOfEntries
- InternetGatewayDevice.WANDevice.{i}.WANCommonInterfaceConfig.
  - EnabledForInternet
  - WANAccessType
- Layer1UpstreamMaxBitRate
- Layer1DownstreamMaxBitRate
- PhysicalLinkStatus
- TotalBytesSent
- TotalBytesReceived
- TotalPacketsSent
- TotalPacketsReceived

- InternetGatewayDevice.WANDevice.{i}.WANDSLConnectionManagement.
  - ConnectionServiceNumberOfEntries

- InternetGatewayDevice.WANDevice.{i}.WANDSLConnectionManagement. ConnectionService.{i}.
  - WANConnectionDevice
  - WANConnectionService
  - DestinationAddress
  - LinkType
  - ConnectionType
  - Name

- InternetGatewayDevice.WANDevice.{i}.WANConnectionDevice.{i}.
  - WANIPConnectionNumberOfEntries
  - WANPPPConnectionNumberOfEntries

- InternetGatewayDevice.WANDevice.{i}.WANConnectionDevice.{i}. WANDSLLinkConfig.
  - Enable
  - LinkStatus
  - LinkType
  - AutoConfig
- DestinationAddress
- ATMTTransmittedBlocks
- ATMRceivedBlocks
- InternetGatewayDevice.WANDevice.{i}.WANConnectionDevice.{i}.WANPPPConnection.{i}.
  - Enable
  - ConnectionStatus
  - LastConnectionError
  - ConnectionType
  - PossibleConnectionTypes
  - Name
  - Uptime
  - Username
  - Password
  - ExternalIPAddress
  - DNS Servers
  - TransportType
  - PPPoEACName
  - PPPoEServiceName
  - RSIPAvailable
  - NATEnabled
  - RemoteIPAddress
  - RouteProtocolRx
  - ShapingRate
  - PortMappingNumberOfEntries
- MACAddress

- InternetGatewayDevice.WANDevice.{i}.WANConnectionDevice.{i}.WANPPPConnection.{i}.Stats.
  - EthernetBytesSent
  - EthernetBytesReceived
  - EthernetPacketsSent
  - EthernetPacketsReceived

- InternetGatewayDevice.WANDevice.{i}.WANConnectionDevice.{i}.WANPPPConnection.{i}.PortMapping.
  - PortMappingEnabled
  - PortMappingLeaseDuration - Always 0
  - RemoteHost - Always empty (wildcard)
  - ExternalPort
  - InternalPort
  - PortMappingProtocol
  - InternalClient
  - PortMappingDescription

- InternetGatewayDevice.WANDevice.{i}.WANConnectionDevice.{i}.WANIPConnection.{i}.
  - Enable
  - ConnectionStatus
  - LastConnectionError
  - PossibleConnectionTypes
  - ConnectionType
  - Name
  - Uptime - not resetted when device is down
- RSIPAvailable
- NATEnabled
- AddressingType
- ExternalIPAddress
- SubnetMask
- DefaultGateway
- DNSEnabled
- DNSOverrideAllowed
- DNSServers
- MaxMTUSize
- RouteProtocolRx
- ShapingRate
- PortMappingNumberOfEntries
- MACAddress

- InternetGatewayDevice.WANDevice.{i}.WANConnectionDevice.{i}.WANIPConnection.{i}.Stats.
  - EthernetBytesSent
  - EthernetBytesReceived
  - EthernetPacketsSent
  - EthernetPacketsReceived

- InternetGatewayDevice.WANDevice.{i}.WANConnectionDevice.{i}.WANIPConnection.{i}.PortMapping.
  - PortMappingEnabled
  - PortMappingLeaseDuration - Always 0
  - RemoteHost - Always empty (wildcard)
  - ExternalPort
- InternalPort
- PortMappingProtocol
- InternalClient
- PortMappingDescription

- InternetGatewayDevice.WANDevice.{i}.WANCoaxInterfaceConfig.
- InternetGatewayDevice.WANDevice.{i}.WANCoaxInterfaceConfig.Stats.
  - BytesSent
  - BytesReceived
  - PacketsSent
  - PacketsReceived

- InternetGatewayDevice.Time.
  - NTPServer1
  - NTPServer2
  - NTPServer3
  - NTPServer4
  - NTPServer5
  - CurrentLocalTime
  - LocalTimeZone
  - DaylightSavingsUsed
  - DaylightSavingsStart
  - DaylightSavingsEnd

- InternetGatewayDevice.Layer2Bridging.BridgeNumberOfEntries
- InternetGatewayDevice.Layer2Bridging.MaxBridgeEntries
- InternetGatewayDevice.Layer2Bridging.AvailableInterfaceNumberOfEntries
- InternetGatewayDevice.Layer2Bridging.Bridge.{i}.BridgeKey
• InternetGatewayDevice.Layer2Bridging.Bridge.{i}.BridgeEnable
• InternetGatewayDevice.Layer2Bridging.Bridge.{i}.BridgeStatus
• InternetGatewayDevice.Layer2Bridging.Bridge.{i}.BridgeName
• InternetGatewayDevice.Layer2Bridging.AvailableInterface.{i}.AvailableInterfaceKey
• InternetGatewayDevice.Layer2Bridging.AvailableInterface.{i}.InterfaceType
• InternetGatewayDevice.Layer2Bridging.AvailableInterface.{i}.InterfaceReference
• InternetGatewayDevice.Layer3Forwarding.
  ■ DefaultConnectionService - Not changeable from default
  ■ ForwardNumberOfEntries
• InternetGatewayDevice.Layer3Forwarding.Forwarding.{i}
  ■ Enable - Cannot be disabled
  ■ Status
  ■ Type - Changes are silently ignored
  ■ DestIPAddress
  ■ DestSubnetMask
  ■ GatewayIPAddress
  ■ Interface
  ■ ForwardingMetric
• InternetGatewayDevice.QueueManagement.
  ■ Enable
  ■ MaxQueues
  ■ MaxClassificationEntries
  ■ ClassificationNumberOfEntries
  ■ MaxPolicerEntries
  ■ PolicerNumberOfEntries
- MaxQueueEntries
- QueueNumberOfEntries
- DefaultPolicer
- DefaultQueue
- DefaultDSCPMark
- DefaultEthernetPriorityMark
- AvailableAppList
- InternetGatewayDevice.QueueManagement.Policer.{i}.
  - PolicerKey
  - PolicerEnable
  - PolicerStatus
  - CommittedRate
  - MeterType
  - PossibleMeterTypes
  - NonConformingAction
- InternetGatewayDevice.QueueManagement.Queue.{i}.
  - QueueKey
  - QueueEnable
  - QueueStatus
  - QueueInterface
  - QueueBufferLength
  - QueueWeight
  - QueuePrecedence
  - DropAlgorithm
  - SchedulerAlgorithm
- ShapingRate

- InternetGatewayDevice.QueueManagement.Classification.{i}.
  - ClassificationKey
  - ClassificationEnable
  - ClassificationStatus
  - ClassificationOrder
  - ClassInterface
  - DestIP
  - DestIPExclude
  - DestMask
  - SourceIP
  - SourceIPExclude
  - SourceMask
  - Protocol
  - ProtocolExclude
  - DestPort
  - DestPortRangeMax
  - SourcePort
  - SourcePortRangeMax
  - SourceVendorClassID
  - SourceVendorClassIDExclude
  - DestVendorClassID
  - DestVendorClassIDExclude
  - SourceClientID
  - SourceClientIDExclude
• DestClientID
• DestClientIDExclude
• SourceUserClassID
• SourceUserClassIDExclude
• DestUserClassID
• DestUserClassIDExclude
• SourceMACAddress
• SourceMACExclude
• DestMACAddress
• DestMACExclude
• TCPACK
• TCPACKExclude
• IPLengthMin
• IPLengthMax
• IPLengthExclude
• DSCPCheck
• DSCPExclude
• DSCPMark
• EthernetPriorityCheck
• EthernetPriorityExclude
• EthernetPriorityMark
• ClassPolicer
• ClassQueue
• ClassApp

• InternetGatewayDevice.QueueManagement.App.{i}.}
- AppKey
- AppEnable
- AppStatus
- ProtocolIdentifier
- AppName
- AppDefaultForwardingPolicy
- AppDefaultPolicer
- AppDefaultQueue
- AppDefaultDSCPMark
- AppDefaultEthernetPriorityMark

- InternetGatewayDevice.QueueManagement.Flow.{i}.
  - FlowKey
  - FlowEnable
  - FlowStatus
  - FlowType
  - FlowTypeParameters
  - FlowName
  - AppIdentifier
  - FlowForwardingPolicy
  - FlowPolicer
  - FlowQueue
  - FlowDSCPMark
  - FlowEthernetPriorityMark

- InternetGatewayDevice.Services.VoiceService.{i}.
  - VoiceProfileNumberOfEntries
  - MaxProfileCount
  - MaxLineCount
  - MaxSessionsPerLine
  - MaxSessionCount
  - SignalingProtocols
  - Regions
  - RTCP
  - SRTP
  - RTPRedundancy
  - DSCPCoupled
  - EthernetTaggingCoupled
  - PSTNSoftSwitchOver
  - FaxT38
  - FaxPassThrough
  - ModemPassThrough
  - ToneGeneration
  - RingGeneration
  - NumberingPlan
  - ButtonMap
  - VoicePortTests
- InternetGatewayDevice.Services.VoiceService.{i}.Capabilities.SIP.
  - Role
  - Extensions
  - Transports
- EventSubscription
- ResponseMap

- InternetGatewayDevice.Services.VoiceService.{i}.Capabilities.Codecs.{i}.
  - EntryID
  - Codec
  - PacketizationPeriod

- InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}. - readonly list
  - Enable
  - NumberOfLines
  - Name
  - DTMFMethod
  - PSTNFailOver

- InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.RTP.
  - LocalPortMin - changes global value (not per-profile)

- InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.SIP.
  - ProxyServer
  - ProxyServerPort
  - ProxyServerTransport
  - RegistrarServerTransport
  - UserAgentDomain
  - UserAgentPort - changes global value (not per-line)
  - UserAgentTransport
  - OutboundProxy
  - OutboundProxyPort
  - RegisterExpires
• InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.Line.{i}.Enable

• InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.Line.{i}.SIP.AuthUserName - changes also affect user-ID

• InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.Line.{i}.Codec.List.{i}.EntryID

• InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.Line.{i}.Codec.List.{i}.Codec

• InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.Line.{i}.Codec.List.{i}.Enable - changes global value (not per-line or per-profile)

• InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.NumberingPlan.MinimumNumberOfDigits

• InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.NumberingPlan.MaximumNumberOfDigits

• InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.NumberingPlan.InterDigitTimerOpen


• InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.NumberingPlan.PrefixInfo.{i}.PrefixMinNumberOfDigits

• InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.NumberingPlan.PrefixInfo.{i}.PrefixMaxNumberOfDigits


OpenRG supports the following Remote Procedure Calls (RPCs):

• GetRPCMethods

• SetParameterValues

• GetParameterValues

• GetParameterNames

• SetParameterAttributes

• GetParameterAttributes
• AddObject
• DeleteObject
• Download
• Upload
• FactoryReset
• ScheduleInform
• Reboot

Note that OpenRG does not implement access control as defined in the specification.

37.6 TR-098 Parameter List

The DSL Forum TR-098 Technical Report specifies the TR-098 full parameter list, including the parameter type and description. For complementary information, refer to this document, which is available at: http://www.broadband-forum.org/technical/trlist.php

The following is the list of the TR-098 objects supported by OpenRG. Objects that are partly supported have a detailed list of their implemented parameters. If not specified, the listed object is fully implemented.

• InternetGatewayDevice.

• InternetGatewayDevice.ManagementServer.
  ■ URL
  ■ Username
  ■ Password
  ■ PeriodicInformEnable
  ■ PeriodicInformInterval
  ■ PeriodicInformTime
  ■ ParameterKey
  ■ ConnectionRequestURL
  ■ ConnectionRequestUsername
  ■ ConnectionRequestPassword
• InternetGatewayDevice.DeviceInfo.
  • Manufacturer
  • ManufacturerOUI (Organizationally Unique Identifier)
  • ModelName
  • Description
  • Product Class
  • SerialNumber
  • HardwareVersion
  • SoftwareVersion
  • ModemFirmwareVersion
  • SpecVersion
  • ProvisioningCode
  • UpTime
  • DeviceLog
  • EnabledOptions
  • FirstUseDate
• InternetGatewayDevice.LANDevice.{i}
• InternetGatewayDevice.LANDevice.{i}.LANHostConfigManagement.
  • DHCPServerEnable
  • MinAddress
  • MaxAddress
  • SubnetMask
  • DNSServers
  • AllowedMACAddresses
  • DHCPRelay
- DomainName
- IPRouters
- DHCPLeaseTime
- IPAddress
- AddressSource
- MACAddress
- HostName
- InterfaceType
- Active
- Status
- MACAddress
- MaxBitRate
- EnabledForInternet
- WANAccessType
- Layer1UpstreamMaxBitRate
- Layer1DownstreamMaxBitRate
- PhysicalLinkStatus
- TotalBytesSent
- TotalBytesReceived
- TotalPacketsSent
- TotalPacketsReceived
- InternetGatewayDevice.WANDevice.{i}.WANDSLConnectionManagement.
- InternetGatewayDevice.WANDevice.{i}.WANDSLConnectionManagement. ConnectionService.{i}.
- InternetGatewayDevice.WANDevice.{i}.WANConnectionDevice.{i}.
- InternetGatewayDevice.WANDevice.{i}.WANConnectionDevice. {i}.WANDSLLinkConfig.
  - Enable
  - LinkStatus
  - LinkType
  - AutoConfig
  - ModulationType
  - DestinationAddress
  - ATMEncapsulation
  - FCSPreserved
  - VCSearchList
  - ATMAAL
  - ATMTransmittedBlocks
  - ATMReceivedBlocks
  - ATMQoS
  - ATMPeakCellRate
  - ATMMaximumBurstSize
  - ATMSustainableCellRate
- InternetGatewayDevice.WANDevice.{i}.WANConnectionDevice.{i}.WANPPPConnection.{i}.
  - ConnectionStatus
  - ConnectionType
  - Name
  - Uptime – not reset when the device is down
  - Username
  - Password
  - ExternallIPAddress
  - DNSServers
  - TransportType
  - PPPoEServiceName
  - PortMappingNumberOfEntries
  - ShapingRate
- InternetGatewayDevice.WANDevice.{i}.WANConnectionDevice.{i}.WANPPPConnection.{i}.PortMapping.{i}.
- InternetGatewayDevice.WANDevice.{i}.WANConnectionDevice.{i}.WANIPConnection.{i}.
  - Enable
  - ConnectionStatus
  - PossibleConnectionType
  - ConnectionType
  - Name
  - Uptime – not reset when the device is down
  - RSIPAvailable
  - NATEnabled
  - AddressingType
- ExternalIPAddress
- SubnetMask
- DefaultGateway
- DNSEnabled
- DNSOverrideAllowed
- DNServers
- MaxMTUSize
- MACAddress
- MACAddressOverride
- RouteProtocolRx
- PortMappingNumberOfEntries
- ShapingRate

- `InternetGatewayDevice.WANDevice.{i}.WANConnectionDevice.{i}.WANIPConnection.{i}.PortMapping.{i}.`
- `InternetGatewayDevice.WANDevice.{i}.WANDSLInterfaceConfig.`
  - Enable
  - Status
  - ModulationType
  - LineEncoding
  - DataPath
  - InterleaveDepth
  - UpstreamCurrRate
  - DownstreamCurrRate
  - UpstreamMaxRate
  - DownstreamMaxRate
- UpstreamNoiseMargin
- DownstreamNoiseMargin
- UpstreamAttenuation
- DownstreamAttenuation
- UpstreamPower
- DownstreamPower
- ShowtimeStart
- QuarterHourStart
- ATURVendor
- ATURCountry
- ATUCVendor
- ATUCCountry

- InternetGatewayDevice.WANDevice.{i}.WANDSLInterfaceConfig.Stats.Showtime.
  - TransmitBlocks
  - LossOfFraming
  - ErroredSecs
  - SeverelyErroredSecs
  - ATUCFECErrors

- InternetGatewayDevice.WANDevice.{i}.WANDSLInterfaceConfig.Stats.CurrentDay
- InternetGatewayDevice.WANDevice.{i}.WANDSLInterfaceConfig.Stats.QuarterHour.
  - CRCErrors

- InternetGatewayDevice.QueueManagement.
  - Enable
  - MaxQueues
  - MaxClassificationEntries
- ClassificationNumberOfEntries
- MaxPolicerEntries
- PolicerNumberOfEntries
- MaxQueueEntries
- QueueNumberOfEntries
- DefaultPolicer
- DefaultQueue
- DefaultDSCPMark
- AvailableAppList

- InternetGatewayDevice.QueueManagement.Classification.{i}.
  - ClassificationKey
  - ClassificationEnable
  - ClassificationStatus
  - ClassificationOrder
  - ClassInterface
  - DestIP
  - DestMask
  - DestIPExclude
  - SourceIP
  - SourceMask
  - SourceIPExclude
  - Protocol
  - DestPort
  - DestPortRangeMax
  - SourcePort
- SourcePortRangeMax
- SourceMACAddress
- SourceMACExclude
- DestMACAddress
- DestMACExclude
- DSCPCheck
- DSCPExclude
- DSCPMark
- EthernetPriorityCheck
- EthernetPriorityExclude
- EthernetPriorityMark
- ClassPolicer
- ClassQueue
- ClassApp
- InternetGatewayDevice.QueueManagement.App.{i}.
  - AppKey
  - AppEnable
  - AppStatus
  - ProtocolIdentifier
  -AppName
  - AppDefaultForwardingPolicy
  - AppDefaultPolicer
  - AppDefaultQueue
  - AppDefaultDSCPMark
  - AppDefaultEthernetPriorityMark
• InternetGatewayDevice.QueueManagement.Flow.{i}.

• InternetGatewayDevice.QueueManagement.Policer.{i}.
  - PolicerKey
  - PolicerEnable
  - PolicerStatus
  - CommittedRate
  - MeterType
  - PossibleMeterTypes
  - NonConformingAction

• InternetGatewayDevice.QueueManagement.Queue.{i}.
  - QueueKey
  - QueueEnable
  - QueueStatus
  - QueueInterface
  - QueueBufferLength
  - QueueWeight
  - QueuePrecedence
  - DropAlgorithm
  - SchedulerAlgorithm
  - ShapingRate

• InternetGatewayDevice.LANConfigSecurity.

37.7 TR-104 Parameter List

The DSL Forum TR-104 Technical Report specifies the TR-104 full parameter list, including the parameter type and description. For complementary information, refer to this document, which is available at: http://www.broadband-forum.org/technical/trlist.php
The following is the list of the TR-104 objects supported by OpenRG. Objects that are partly supported have a detailed list of their implemented parameters. If not specified, the listed object is fully implemented.

- InternetGatewayDevice.Services.VoiceService.{i}.
  - VoiceProfileNumberOfEntries
  - MaxProfileCount
  - MaxLineCount
  - MaxSessionsPerLine
  - MaxSessionCount
  - SignalingProtocols
  - Regions
  - RTCP
  - SRTP
  - RTPRedundancy
  - DSCPCoupled
  - EthernetTaggingCoupled
  - PSTNSoftSwitchOver
  - FaxT38
  - FaxPassThrough
  - ModemPassThrough
  - ToneGeneration
  - RingGeneration
  - NumberingPlan
  - ButtonMap
  - VoicePortTests
• InternetGatewayDevice.Services.VoiceService.{i}.Capabilities.SIP.
  - Role
  - Extensions
  - Transports
  - EventSubscription
  - ResponseMap

• InternetGatewayDevice.Services.VoiceService.{i}.Capabilities.Codecs.{i}.
  - EntryID
  - Codec
  - PacketizationPeriod

• InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.
  - Enable
  - NumberOfLines
  - Name
  - Reset
  - DTMFMethod
  - PSTNFailOver

• InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.SIP.
  - ProxyServer
  - ProxyServerPort
  - ProxyServerTransport
  - RegistrarServerTransport
  - UserAgentDomain
  - UserAgentPort
  - UserAgentTransport
- OutboundProxy
- OutboundProxyPort
- RegisterExpires
- InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.RTP.
  - LocalPortMin
- InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.Line.{i}.
  - Enable
  - DirectoryNumber
  - PhyReferenceList
- InternetGatewayDevice.Services.VoiceService.{i}.VoiceProfile.{i}.Line.{i}.SIP.
  - AuthUserName
  - AuthPassword
  - EntryID
  - Codec
  - Enable
  - MinimumNumberOfDigits
  - MaximumNumberOfDigits
  - InterDigitTimerOpen
  - PrefixInfo.{i}.PrefixRange
  - PrefixInfo.{i}.PrefixMinNumberOfDigits
  - PrefixInfo.{i}.PrefixMaxNumberOfDigits
  - PrefixInfo.{i}.FacilityAction
PhyPort

InterfaceID

Description

### 37.8 TR-111 Parameter List

The DSL Forum TR-111 Technical Report specifies the TR-111 full parameter list, including the parameter type and description. For complementary information, refer to this document, which is available at: [http://www.broadband-forum.org/technical/trlist.php](http://www.broadband-forum.org/technical/trlist.php)

The following is the list of the TR-111 objects supported by OpenRG. Objects that are partly supported have a detailed list of their implemented parameters. If not specified, the listed object is fully implemented.

- **InternetGatewayDevice.ManagementServer.**
  - ManageableDeviceNumberOfEntries
  - ManageableDeviceNotificationLimit

- **InternetGatewayDevice.ManagementServer.ManageableDevice.{i}.**
  - ManufacturerOUI
  - SerialNumber
  - ProductClass
38
TR-064

38.1 Overview

As residential gateways offer increasingly complex services, customer premise installation and configuration increase the operators' operational costs. DSL Forum's LAN-Side DSL CPE Configuration protocol, known as TR-064, provides a zero-touch solution for automating the installation and configuration of gateways from the LAN side.

The TR-064 interface includes the following implementations:

- **Management Protocol**: Standards based XML over SOAP protocol.
- **Parameter Model**: Parameters defined using UPnP model as a base.
- **CPE Type**: Supports Bridge/Router/PPPoE on-board IP pass-through CPEs.
- **Management Usage**: CPE turn-up, status determination, monitoring, diagnostics.
- **CPE discovery**: Standards-based DHCP and SSDP device discovery.
- **OS Support**: CPE management application with integrated XML over SOAP stack to operate on any OS, native XML/SOAP OS support is not required or desired.
- **Proprietary/Open**: Standards-based with a published open interface.

38.2 Architecture Implementation

This section describes the structure of TR-064 and Internet Gateway Device (IGD) modules. It also explains how to add a new:
• Device
• Service
• Action
• State variable

38.2.1 Engine Structure

TR-064 implementation in OpenRG uses IGD, which in turn uses the generic UPnP stack. The
generic UPnP stack code resides in $rg/pkg/upnp/stack$, the IGD code resides in $rg/pkg/upnp/
igd$ and TR-064 in $rg/pkg/upnp/TR-064$.

38.2.1.1 UPnP Terms

UPnP, and as a consequence TR-064 and IGD as well, uses the following terms to describe a
UPnP device:

DEVICE A container object, which contains zero or more other DEVICES. In addition,
each DEVICE contains SERVICES. An example DEVICE is **WANDevice**.

SERVICE A container object, which contains STATE VARIABLES and ACTIONS. An
example SERVICE is **WANCommonInterfaceConfigService**, which belongs to **WANDevice**
DEVICE.

STATE VARIABLE Models specific property. An example STATE VARIABLE is **DNSServers**, which belongs to **LANHostConfigManagement** SERVICE.

ACTION An operation that may be applied to the SERVICE, which uses zero or more
arguments and may return a result or results. An example ACTION is **GetDNSServers**, which
belongs to **LANHostConfigManagement** SERVICE.

38.2.1.2 UPnP Tree Model

UPnP creates a tree model, which represents all defined DEVICES, SERVICES, STATE
VARIABLES and ACTIONS. The tree is re-created whenever needed, e.g. when a new
connection is created by the user. Currently, the tree structure is hard-coded in $rg/pkg/upnp/
igd/igd_tree.c$, which means that whenever a new DEVICE or SERVICE is implemented—this
file has to be updated. STATE VARIABLES and ACTIONS are defined in their own files, and
addition of any of them to existing SERVICE does **not** require any changes outside their own
file.

38.2.1.3 DEVICE

DEVICE is represented by the **upnp_device_desc_t** structure, defined in $rg/pkg/upnp/
stack/upnp.h$. Note that since TR-064 uses the same DEVICES as IGD, these definitions apply
to both IGD and TR-064.
38.2.1.4 SERVICE

SERVICE is represented by the upnp_service_desc_t structure, defined in \textit{rg/pkg/upnp/stack/upnp.h}. The \texttt{app_id} field in each SERVICE determines if the SERVICE is used in IGD, TR-064 or both. Each SERVICE is defined separately, most of them in separate files. Each SERVICE exports a creation function, which receives a \texttt{ServiceId} string and an owning DEVICE. The creation function allocates a new upnp_service_t structure and fills it with the necessary fields. The creation function is used in \textit{rg/pkg/upnp/igd/igd_tree.c} when the UPnP tree is created.

38.2.1.5 STATE VARIABLE

STATE VARIABLE is represented by the upnp_statevar_desc_t structure, defined in \textit{rg/pkg/upnp/stack/upnp.h}. As in SERVICE, the \texttt{app_id} field in each STATE VARIABLE determines if the STATE VARIABLE is used in IGD, TR-064 or both.

38.2.1.6 ACTION

ACTION is represented by the upnp_statevar_desc_t structure, defined in \textit{rg/pkg/upnp/stack/upnp.h}. As in SERVICE, the \texttt{app_id} field in each ACTION determines if the ACTION is used in IGD, TR-064 or both.

38.2.2 Implementation

38.2.2.1 SERVICE

SERVICE implementation includes four main structures:

1. \texttt{upnp_notify_subsc_cb_t notify_subcr} This function is called when a subscription request is made from a UPnP client. This function should return all evented STATE VARIABLEs values. The prototype for the function is as follows:

   ```c
   static void demo_subscr(upnp_service_t *s, upnp_app_id_t app_id, attrib_t **vars)
   {
     /* Get evented state variables value */
     char *evented_var_value = get_evented_var_value();

     /* Store in result */
     attrib_addset(vars, "EventedVarName", evented_var_value);
   }
   ```

2. \texttt{upnp_notify_cb_t notify} This function is called when UPnP needs to send updated values to the subscribed UPnP client. This function should return all evented STATE VARIABLES values whose values were changed. The prototype for the function is as follows:

   ```c
   static void igd_wan_conn_eh(upnp_service_t *s, u32 event_mask, attrib_t **vars)
   {
     if (event_mask & DEMO_MASK)
   ```
Note: DEMO_MASK is the type of the event that caused the change. Currently four types are defined in igd_tree.h, all named IGD_EVENT_XXX.

3. upnp_statevar_desc_t *vars SERVICE STATE VARIABLEs—explained below.

4. upnp_action_desc_t *actions SERVICE ACTIONs—explained below.

### 38.2.2.2 STATE VARIABLE

The main part of STATE VARIABLE implementation is the callback (cb) function. This function is called when the UPnP client executes QueryStateVariable on the STATE VARIABLE. The function receives a pointer to the SERVICE structure and should allocate and return a string that will be used in the response. The prototype for the function is as follows:

```c
static char *demo_state_var_cb(upnp_service_t *s)
{
    int state;
    /* Get the state of the variable */
    state = function_to_get_state();
    /* Allocate string representation */
    return strdup_e(itoa(state));
}
```

### 38.2.2.3 ACTION

The main part of action implementation is the cb function. This function is called when the UPnP client executes the ACTION. The function receives a pointer to the SERVICE structure and a structure that contains input arguments. Results, if defined, should also be placed in this structure. The prototype for the function is as follows:

```c
static void demo_action_cb(upnp_service_t *s, upnp_event_action_t *e)
{
    char *result;
    /* Get input arguments */
    char *val1 = attrib_get(e->arg_in, "NewInVar1Name");
    char *val2 = attrib_get(e->arg_in, "NewInVar2Name");
    int int_val = upnp_var_list_get_int(e->arg_in, "NewIntVarName");
    /* Perform action using input values, set result to result of action */
    /* If error - set e->err to error code, and optionally set e->err_desc to a STATIC error description string */
    /* Store results of the action */
    attrib_addset(e->arg_out, "NewOutVarName", result);
}
```
38.2.3 Adding New Objects

38.2.3.1 DEVICE

1. Define the new `upnp_device_desc_t` structure in `rg/pkg/upnp/igd/vendor.c`.

2. In the `rg/pkg/upnp/igd/igd_tree.c` file:
   - Add a call to `upnp_device_alloc()`, with the new structure as its argument, in order to register the new DEVICE as a child of an existing one.
   - Add the new structure to the external `upnp_device_desc_t` declaration in the beginning of the file.

38.2.3.2 SERVICE

1. Define the new `upnp_service_desc_t` structure in a separate file.

2. Define an exported function to allocate the new SERVICE. The function should be similar to the following:
   ```c
   upnp_service_t *demo_service_alloc(char *id, igd_ifc_t *ifc)
   {
     return upnp_service_alloc(id, &demo_service_desc, ifc);
   }
   ```

3. Add a call to the new SERVICE allocation function in the appropriate DEVICE in `rg/pkg/upnp/igd/igd_tree.c`.

38.2.3.3 STATE VARIABLE and ACTION

These needs to be defined in the SERVICE file, as described in the Implementation section (Section 38.2.2).

38.3 Parameter List

The DSL Forum TR-064 Technical Report specifies the TR-064 full parameter list, including the parameter type and description. For complementary information, refer to this document, which is available at: http://www.broadband-forum.org/technical/trlist.php

The following is the list of the TR-064 services supported by OpenRG. Each service has a detailed list of its implemented state variables and actions.

1. DeviceInfo
   - State Variables
     - ManufacturerName
- ManufacturerOUI
- ModelName
- Description
- ProductClass
- SerialNumber
- SoftwareVersion
- HardwareVersion
- SpecVersion
- ProvisioningCode
- UpTime
- DeviceLog

**Actions**
- GetInfo
- SetProvisioningCode
- GetDeviceLog

**DeviceConfig**

- State Variables
  - PersistentData
    - A_ARG_TYPE_Status – RebootRequired is not implemented
    - A_ARG_TYPE_UUID
  
- Actions
  - GetPersistentData
  - SetPersistentData
  - ConfigurationStarted – Locking not implemented
  - ConfigurationFinished
- FactoryReset
- Reboot

3. ManagementServer

- State Variables
  - URL
  - Password
  - PeriodicInformEnable
  - PeriodicInformInterval
  - PeriodicInformTime
  - ParameterKey
  - ParameterHash
  - ConnectionRequestURL
  - ConnectionRequestUsername
  - ConnectionRequestPassword
  - UpgradesManaged

- Actions
  - GetInfo
  - SetManagementServerURL
  - SetManagementServerPassword
  - SetPeriodicInform
  - SetConnectionRequestAuthentication

4. LANHostConfigManagement

- State Variables
  - DHCPServerConfigurable
  - DHCPRelay
- SubnetMask
- DNS Servers
- Domain Name
- Min Address
- Max Address
- IP Routers
- Reserved Addresses
- DHCP Lease Time
- DHCP Server Enable
- Enable
- IP Interface IP Address
- IP Interface Subnet Mask
- IP Interface Addressing Type
- IP Interface Number Of Entries

- Actions
  - Get Info
  - Set DHCP Server Configurable – cannot be set to 0
  - Get DHCP Server Configurable
  - Set DHCP Relay
  - Get DHCP Relay
  - Set Subnet Mask
  - Get Subnet Mask
  - Set IPRouter
  - Delete IPRouter
  - Get IPRouters List
SetDomainName
GetDomainName
SetAddressRange
GetAddressRange
GetReservedAddresses
SetDNSServer
DeleteDNSServer
GetDNServers
SetDHCPLeaseTime
SetDHCPServerEnable
GetIPInterfacetNumberOfEntries
SetIPInterface
GetIPInterfaceSpecificEntry
GetIPInterfaceGenericEntry

5. LANEthernetInterfaceConfig

• State Variables
  • Enable
  • Status – status is not read from device
  • MACAddress
  • MaxBitRate
  • DuplexMode – duplex mode is not read from device
  • Stats.BytesSent
  • Stats.BytesReceived
  • Stats.PacketsSent
  • Stats.PacketsReceived
• Actions
  ■ SetEnable
  ■ GetInfo
  ■ GetStatistics

6. WLANConfiguration

• State Variables
  • Enable
  • Status
  • MaxBitRate
  • Channel
  • SSID
  • BeaconType
  • WEPKeyIndex
  • WEPKey
  • WEPEncryptionLevel
  • PreSharedKeyIndex
  • AssociatedDeviceMACAddress
  • PreSharedKey
  • KeyPassphrase
  • MACAddressControlEnabled
  • Standard
  • BSSID
  • TotalBytesSent
  • TotalBytesReceived
  • TotalPacketsSent
- TotalPacketsReceived
- BasicEncryptionModes
- BasicAuthenticationMode
- WPAEncryptionModes
- WPAAuthenticationMode
- PossibleChannels
- BasicDataTransmitRates
- OperationalDataTransmitRates
- PossibleDataTransmitRates
- IEEE11iEncryptionModes
- IEEE11iAuthenticationMode
- TotalAssociations
- AssociatedDeviceMACAddress
- AssociatedDeviceIPAddress
- AssociatedDeviceAuthenticationState
- LastRequestedUnicastCipher
- LastRequestedMulticastCipher
- LastPMKId
- RadioEnabled
- AutoRateFallBackEnabled
- InsecureOOBAccessEnabled
- BeaconAdvertisementEnabled
- LocationDescription
- RegulatoryDomain
- TotalPSKFailures
• TotalIntegrityFailures
• ChannelsInUse
• DeviceOperationMode
• DistanceFromRoot
• PeerBSSID
• AuthenticationServiceMode

• Actions
  • SetEnable
  • GetInfo
  • SetConfig
  • SetSecurityKeys
  • GetSecurityKeys
  • SetDefaultWEPKeyIndex
  • GetDefaultWEPKeyIndex
  • SetBasBeaconSecurityProperties
  • GetBasBeaconSecurityProperties
  • SetWPABeaconSecurityProperties
  • GetWPABeaconSecurityProperties
  • GetStatistics
  • SetSSID
  • GetSSID
  • GetBSSID
  • SetBeaconType
  • GetBeaconType
  • SetChannel
- GetChannelInfo
- SetDataTransmitRates
- GetDataTransmitRateInfo
- GetByteStatistics
- GetPacketStatistics
- Set11iBeaconSecurityProperties
- Get11iBeaconSecurityProperties
- GetTotalAssociations
- GetGenericAssociatedDeviceInfo
- GetSpecificAssociatedDeviceInfo
- GetSpecificAssociatedDev11iInfo
- GetByteStatsForAssociatedDev
- GetPacketStatsForAssociatedDev
- SetRadioMode
- GetRadioMode
- SetAutoRateFallBackMode
- GetAutoRateFallBackMode
- SetInsecureOutOfBandAccessMode
- GetInsecureOutOfBandAccessMode
- SetBeaconAdvertisement
- GetBeaconAdvertisement
- SetLocationDescription
- GetLocationDescription
- SetRegulatoryDomain
- GetRegulatoryDomain
• GetFailureStatusInfo
• GetChannelsInUse
• SetDeviceOperationMode
• GetDeviceOperationMode
• SetAuthenticationServiceMode
• GetAuthenticationServiceMode
• SetPreSharedKey
• GetPreSharedKey
• FactoryDefaultReset
• ResetAuthentication

7. WANCommonInterfaceConfig

• State Variables
  • WANAccessType
  • Layer1UpstreamMaxBitRate
  • Layer1DownstreamMaxBitRate
  • PhysicalLinkStatus
  • TotalBytesSent
  • TotalBytesReceived
  • TotalPacketsSent
  • TotalPacketsReceived
  • EnabledForInternet

• Actions
  • SetEnabledForInternet
  • GetEnabledForInternet
  • GetCommonLinkProperties
- GetTotalBytesSent
- GetTotalBytesReceived
- GetTotalPacketsSent
- GetTotalPacketsReceived

8. WANEthernetInterfaceConfig

- State Variables
  - Enable
  - Status – status is not read from device
  - MACAddress
  - MaxBitRate
  - DuplexMode – duplex mode is not read from device
  - Stats.BytesSent
  - Stats.BytesReceived
  - Stats.PacketsSent
  - Stats.PacketsReceived

- Actions
  - SetEnable
  - GetInfo
  - GetStatistics

9. WANDSLLinkConfig

- State Variables
  - Enable
  - LinkType
  - LinkStatus
  - AutoConfig
- DestinationAddress

- Actions
  - SetEnable
  - GetInfo
  - GetDSLLinkInfo
  - GetAutoConfig
  - SetDestinationAddress
  - GetDestinationAddress

10. WANEthernetLinkConfig

- State Variables
  - EthernetLinkStatus

- Actions
  - GetEthernetLinkStatus

11. WANIPConnection

- State Variables
  - Enable
  - ConnectionStatus
  - Name
  - Uptime
  - LastConnectionError
  - RSIPAvailable
  - NATEnabled
  - ExternalIPAddress
  - SubnetMask
  - PortMappingNumberOfEntries
- PortMappingEnabled
- RemoteHost – must be 0.0.0.0
- ExternalPort
- InternalPort – does not work (see B15970)
- PortMappingProtocol
- InternalClient
- PortMappingDescription
- AddressingType
- DefaultGateway
- MACAddress
- DNSEnabled – hardcoded to "1"
- DNSOverrideAllowed – hardcoded to "1"
- DNS Servers
- ConnectionTrigger
- RouteProtocolRx

- Actions
  - SetEnable
  - GetInfo
  - GetConnectionTypeInfo
  - RequestConnection
  - RequestTermination
  - ForceTermination
  - GetStatusInfo
  - GetNATRSIPStatus
  - GetPortMappingNumberOfEntries
- GetGenericPortMappingEntry
- GetSpecificPortMappingEntry
- AddPortMapping
- DeletePortMapping
- GetExternalIPAddress
- SetRouteProtocolRx

12. WANPPPConnection

- State Variables
  - Enable
  - ConnectionStatus
  - Name
  - Uptime
  - UpstreamMaxBitRate
  - DownstreamMaxBitRate
  - LastConnectionError
  - IdleDisconnectTime
  - ConnectionTrigger
  - RSIPAvailable
  - NATEnabled
  - UserName
  - Password
  - ExternalIPAddress
  - PortMappingNumberOfEntries
  - PortMappingEnabled
  - RemoteHost – must be 0.0.0.0
- ExternalPort
- InternalPort – does not work
- PortMappingProtocol
- InternalClient
- PortMappingDescription
- MACAddress – no MAC returned, not even for PPPoE
- DNSEnabled – hardcoded to "1"
- DNSOverrideAllowed – hardcoded to "1"
- DNSServers
- TransportType
- PPPoEACName
- PPPoEServiceName
- RouteProtocolRx

**Actions**
- SetEnable
- GetInfo
- GetConnectionTypeInfo
- ConfigureConnection
- RequestConnection
- RequestTermination
- ForceTermination
- SetIdleDisconnectTime
- GetStatusInfo
- GetLinkLayerMaxBitRates
- GetUserName
- SetUserName
- GetPassword
- SetPassword
- GetIdleDisconnectTime
- GetNATRSIPStatus
- GetPortMappingNumberOfEntries
- GetGenericPortMappingEntry
- GetSpecificPortMappingEntry
- AddPortMapping
- DeletePortMapping
- GetExternalIPAddress
- SetRouteProtocolRx
Network Acceleration

OpenRG incorporates a mechanism for accelerating the flow of traffic through your network, by utilizing a Packet Streaming Engine (PSE). PSE is implemented by either hardware fastpath or software fastpath. While the first method is hardware dependant (requires a dedicated processor), software fastpath is a software-based general solution that intelligently manipulates data in order to shorten its travel time through the network.

39.1 Fastpath Hardware Acceleration

Hardware fastpath is a solution that enables flows entering the processor to skip the software handling, thus shortening latency. When a flow of data arrives at OpenRG’s network stack, the first packet goes through the network stack, where it is analyzed and configured. Based on this analysis and configuration, the rest of the stream goes through the fastpath, at the hardware level.

Hardware fastpath is available on boards that have a hardware capability to filter and route packets using a dedicated processor. To implement this, the hardware needs to know which streams to accelerate, according to rules based on traffic MACs, IPs, ports, etc. OpenRG has a hardware independent layer for supporting these capabilities. The hardware fastpath module in OpenRG needs to configure the board’s hardware-specific layer with acceleration rules. In order to do so, it takes a client’s (e.g. firewall, bridge) filtering rules as the base for the fastpath rules. The rules are then passed to the hardware-specific layer, which configures the hardware acceleration.
Figure 39.1 Fastpath Hardware Acceleration

The firewall’s connections are the context from which the fastpath acceleration rules are derived. After a connection is chosen to be accelerated, it is marked as CONN_HW_FASTPATH_ENABLED. An interface is given to the connection through which it can signal to the hardware that the connection is no longer needed. After accelerating a firewall connection, matching packets will be routed at the hardware level, meaning OpenRG will not be aware of them at all. The only packets in the connection that will not be accelerated are flagged packets of a TCP connection (SYN, FIN, RST, etc.). The fastpath module prevents the firewall connection from being closed, as long as there is accelerated traffic. The firewall’s client hardware fastpath code is implemented in pkg/firewall/kernel/fw_fastpath.c.

Hardware fastpath is limited in number of accelerated connections, and cannot handle all session types, typically resulting in 50% of traffic that cannot be accelerated. In addition, fastpathed streams do not enjoy the functionality offered by the software—security, filtering, fragmentation, application-level gateways, quality of service and more.

39.2 Fastpath Software Acceleration

OpenRG's PSE is a data path acceleration technology that offers intelligent, optimized and policy-based handling of hardware fastpath, as well as software fastpath, as described in the following figure.
Figure 39.2 PSE Technology

The PSE profiling mechanism examines streams and packets and applies policy-based optimization to decide on its optimal handling. PSE bases its smart profiling and decision-making on:

- Acceleration potential: session-types with high acceleration factors are prioritized
- Session throughput: high-throughput sessions are prioritized, instead of the simple first-come-first-accelerated typically implemented
- Time sensitivity: applications which are time-sensitive enter acceleration first

PSE offers a unique software fastpath, which directs packets only to the required stations, shortening the processing path, and achieving 100% boost over regular software packet handling. In addition, PSE accelerates processing of the multicast traffic.

OpenRG with PSE incorporates a set of rules and policies which are chipset-specific, according to the capabilities and acceleration parameters provided by the hardware. These capabilities are encapsulated within a standard API, so that the same Jungo software and new services run on different hardware platforms, each with its own unique performance engine, achieving optimal performance on every chip.

39.2.1 Architecture

PSE consists of two main parts:

- The PSE engine, a basic but very efficient match-and-modify engine. It knows nothing about networking but it knows how to modify packets according to pre-defined rules, and transmit them to designated devices. OpenRG has a software implementation of this engine that can function on all types of hardware. OpenRG also supports hardware implementations of PSE using the same infrastructure and APIs.

- The PSE infrastructure, which is the 'brain' of the PSE mechanism. It exists through the system (firewall, bridge, routing, IGMP, QoS, etc.), and its role is to gather information on the data path, decide which connections should be accelerated and configure the acceleration
layer accordingly. Additionally, it removes the acceleration when the connection is closed or on a timeout.

The goal of PSE is to configure rule(s) to the firmware, causing it to mimic OpenRG’s operation on current packet flows. This goal is achieved by the following principles:

- **Learn** – Gather information on the packet during its life cycle (network stack, knet hooks)
- **Analyze** – Examine gathered information, determine whether it should be accelerated and how
- **Configure** – Add acceleration rule according to gathered information

The impact PSE has on the network's performance varies significantly based on three parameters: CPU strength, chipset vendor driver quality, and data path scenario. However, research shows that overall performance is improved by a minimum of 30% and can reach up to 200% of improvement.

PSE also affects the CPU load. In real world scenarios of large packet sizes and limited bandwidth scenarios (ADSL2+, Wireless, etc), PSE provides a more efficient data path which frees up CPU resources to handle new services that can be added to the image.
39.2.2 An In-depth Look at PSE

To understand the PSE mechanism, you must first be familiar with PSE terminology:

**Flow Info** Information of the packet's flow, attached to Socket Buffer (SKB).

**FP Client** An external module to PSE providing flow analysis knowledge (e.g. Firewall, Bridge, IGMP Proxy).

**Client Info, Client Context** Part of flow info that is client specific (e.g. reference to FW connection/Bridge node/IGMP MC Group node).

**PSE Entry** Representation of an accelerated flow. A PSE entry contains a reference to a PSE engine acceleration rule. After construction, it is stored and maintained by the PSE client.

The PSE scheme of operation is consisted of three main steps:

1. The PSE RX hook initializes the flow info, and performs an RX analysis.

2. A packet goes through the PSE Client (Firewall/Bridge/IGMP Proxy). The client decides that flow needs to be accelerated, and 'attaches' its context (fw conn, brt node, etc.) to the packet's flow info.

3. The PSE TX hook performs further analysis. Upon the bottom-most interface, it extracts the fastpath 'knowledge' from the client context attached to the flow info. It then configures a PSE engine acceleration rule, using that knowledge, and attaches the created PSE entry to the client's context.
Flow Info

- Holds flow information of the packet
  - Receive and transmit interfaces (physical and logical)
  - VLAN tagging, PPPoE session ID
  - Reference to client's context (FW connection)
  - Platform specific information
- Allocated and initialized at the RX hook of the Fast Path module, upon the earliest entry to RX hooks (bottom most level network interface)
- Copied and destructed when SKB is copied/freed
- Modified only within the PSE module itself
  - During RX/TX analysis (within PSE module RX/TX hooks)
  - By the PSE Client, when invoking fp_flow_info_client_ctx_attach()

PSE Entry

- PSE module's opaque representation of an acceleration rule
- Holds both source and destination logical interfaces and a reference to PSE engine acceleration rule
• Allocated and initialized upon PSE engine acceleration Rule creation (at the end of PSE TX Hook)

• Stored within the Client Context responsible for creating this rule

• Destructed upon Client Context destruction
Asynchronous JavaScript and XML (AJAX)

AJAX is a group of inter-related Web development techniques used for creating interactive Web applications. A primary characteristic is the increased responsiveness and interactivity of Web pages, achieved by exchanging small amounts of data with the server "behind the scenes", so that entire Web pages do not have to be reloaded each time there is a need to fetch data from the server.

AJAX is asynchronous in that extra data is requested from the server and loaded in the background without interfering with the display and behavior of the existing page.

OpenRG utilizes AJAX technology to enable you to "hook" into your deployed gateways, in order to increase the WBM pages' interactivity, speed, functionality, and usability. Remotely enhancing the gateway's capabilities and performance prevents or delays the need to update its software, thus lengthening the gateway's life span.

Examples where AJAX technology is implemented in OpenRG include the Wireless Password Reminder and the Wireless Web Authentication features (refer to the 'Reducing Support Calls' chapter of the User Manual).

40.1 OpenRG AJAX Architecture

AJAX technology implements a client-server architecture. OpenRG is the client which is served by an AJAX server. This server may be a Jungo.net server (if available) or a designated AJAX server. When a WBM page loads on a browser, it sends the AJAX server a query asking whether there are any updates. This is done without interference with the loading of the page. If the AJAX server replies with additions or amendments, the browser instantaneously applies them without having to reload the page.
AJAX is enabled on OpenRG by default. To disable it, use the following CLI command:

```
OpenRG> conf set /wbm/ajax/disabled 1
```

If your OpenRG image does not contain the AJAX feature, compile an image using `CON GilF_RG_WBM_AJAX=y`

The AJAX client code on OpenRG is located under `rg/pkg/web_mng/cgi`

The AJAX server(s) with which OpenRG interacts are defined in OpenRG's configuration file, `rg_conf`. A single gateway may be served by multiple servers for redundancy and load balancing. To view the AJAX configuration entries, type the following CLI command:

```
OpenRG> conf print wbm/ajax
```

When adding a server entry to `rg_conf`, the following format must be used:

```
<domain name or IP address>/<cgi name>
```

For example,

```
www.ajax_server.com/ajax.cgi
```

For more information, refer to the 'WBM' chapter of the Configuration Entries Guide.

Since the browser limits the connection to the host from which the Web page is received, an **AJAX Proxy** was created on OpenRG to provide a pipeline to the AJAX server.
Figure 40.1 AJAX Flow

Every WBM page contains JavaScript code that opens an inline connection with the AJAX server when the page is loaded, via OpenRG's proxy module. The proxy merely serves as a bidirectional pipeline with the AJAX server. The connection is in the form of an HTTP GET command, through which the page passes an attribute list to the AJAX server. This list may include page information, version number, hardware type, and other context related identifiers. The structure for such a command is:

```
/ajax_proxy.cgi?<attribute list>
```

The AJAX server may reply in one of two ways:

- **400 Bad Request** – There are no updates or amendments to the page, therefore an empty reply is sent. The loaded page remains unchanged.

- **200 OK** – There are updates/amendments for the page. The reply contains inline JavaScript code that performs specific changes to the page. These changes can be added functionality, button behavior changes, altered events—basically any JavaScript behavior changes that you wish to perform on the page.
A connection with the AJAX server may not be established for various reasons, such as a bad connection or a server failure. In this case, the page loads regularly and remains unchanged. There is no dependency on the AJAX server, as it only provides extra functionality over the basic behavior. The end user is not aware of this technology at any point, whether the page is or is not updated by the AJAX server.

### 40.2 AJAX Server Functionality

Since all WBM pages open inline connections with the AJAX server when loaded, you can implement functionality on your server to control any page in any form that JavaScript will allow. This section describes how OpenRG’s Wireless Password Reminder feature is remotely deployed using the AJAX server. Use this example to implement your own WBM page manipulation.

In OpenRG, the Jungo.net server implements the AJAX technology. Therefore, the AJAX server code resides in `pkg/jnet/server/ajax`. This is not mandatory; you can utilize a designated AJAX server.

The AJAX engine is implemented in `ajax cgi.c`, which is a generic Common Gateway Interface (CGI) that listens to requests and answers them. When a page request is received, the CGI identifies the WBM page by extracting its information and parameters from the URL. It then picks a function (from a function table) that handles that specific page. This is performed by the `get_page_cb_by_page_id()` function:

```c
page_cb = get_page_cb_by_page_id(ajax_wbm_info->page_id);
```

This function is implemented in the `pages cb.c` file, and contains the table of callback functions that handle specific pages.

```c
14 page_cb_t *get_page_cb_by_page_id(char *page)
15 {
16    page_item_t page_table[] = {
17      {page_fp_main, ajax_forgot_password_cb},
18      {page_web_auth_login, ajax_web_auth_cb},
19      {NULL, NULL}
20    };
21    page_item_t *iter = NULL;
22    for (iter = page_table; iter->page_id && strcmp(iter->page_id, page);
23      iter++);
24    return iter->page_cb;
25 }
```

You must add your page handling callback function to this table, after line 18, in the following format:

```c
{<page identifier>, <page handling function>},
```

Each callback function is implemented in its own designated file. In this example, `ajax_forgot_password_cb()` is implemented in `forgot_password.c`. The function receives the page information and processes it, determining whether or not the page should be updated. If changes are required, the function uses `forgot_password.js` to manipulate the
Asynchronous JavaScript and XML (AJAX)

This file is not an actual JavaScript file; it represents a template of a JavaScript file, but compiles into a C string. This template file includes the add_radio_options() function, which adds two radio buttons to the page. The callback function fills out the required parameters in the file and returns a JavaScript string, which is then implemented on the page. When no changes are required, it returns a null string. The following code is the main part of the callback function:

```
pkg/jnet/server/ajax/forgot_password.c
43  table = p_table(&base, MAKE_PERCENT(100), TABLE_TRANSPARENT, 0);
44  p_wiz_option(table, SYM_FORGOT_PASSWORD_SELECTION, "-2",
45     Tsend_me_the_password_by_sms, NULL, 1);
46  p_wiz_option(table, SYM_FORGOT_PASSWORD_SELECTION, "-1",
47     Tcreate_user_by_jnet, Tcreate_user_by_jnet_desc, 0);
48  str_printf(&frame_data_id, "%s_%s0", ID_STR_FRAME_DATA, "ts.ts0",
49     ID_STR_FRAME_DATA);
50  str_printf(&btn_id, "%s_%s", ID_STR_BUTTON, "wizard_next");
51
52  str_printf(js_str, forgot_password_js_str, SYM_NAV "_" SYM_AJAX,
53     SYM_FORGOT_PASSWORD_SELECTION, page_jnet_forgot_wl_password_sms,
54     page_jnet_forgot_wl_password_create_user, JNET_POST_LOGIN_PAGE_VAR,
55     "http_interception_ajax_redirect", jnet_conf_get(JNET_CFG_URL),
56     jnet_conf_get(JNET_CFG_CGI), FIELD_ACTIVE_PAGE,
57     JNET_CPE_INFO_VAR, ajax_cpe_info->encoded_query, frame_data_id,
58     *html_to_str(&str, base), btn_id);
```

What does this code do?

**Lines 43-47** Create HTML code for two radio buttons that will be inserted to the WBM page.

**Lines 52-59** Fill the template represented by the forgot_password_js_str string with parameters. The str_printf() function contains the parameters that fill the forgot_password.js template, which is compiled into a JavaScript string and sent back to the page.

The following code is the add_radio_options() function, implemented in the JavaScript template:

```
pkg/jnet/server/ajax/forgot_password.js
33  function add_radio_options()
34  {
35      var frame_data, tbody, row, cell;
36      var radio;
37      var i = 0;
38      var ajax_hidden;
39      var d = document;
40
41      /* Find the frame to manipulate. */
42      frame_data = d.getElementById("%s");
43      if (!frame_data)
44         return;
45
46      /* Add another radio option for sending SMS. */
47      tbody = frame_data.firstChild;
48      row = tbody.insertRow(0);
49      cell = row.insertCell(0);
50      cell.innerHTML = "ts";
51
52      ajax_hidden = d.getElementsByTagName(nav_ajax_prefix + "_" +
53          SYM_FORGOT_PASSWORD_SELECTION)[0];
54
55      /* Uncheck the currently selected radio button. */
56      while (1)
```
What does this code do?

**Lines 42-44** Locate the frame of the HTML page on which the change will be applied.

**Lines 47-53** Add the radio buttons for sending an SMS password reminder and entering login details via Jungo.net, as the first two radio buttons.

**Lines 56-62** Uncheck the original radio button.

**Lines 64-69** Check the new first radio button by default.

**Line 72** Assume control over the 'Next' button, which must now generate a different outcome.

**Line 75** This `ajax_exec_on_dom_content_loaded()` function invokes the `add_radio_options()` function as soon as the Document Object Model (DOM) content of the HTML page is loaded (before the graphical content is loaded). The purpose is to integrate the AJAX update with the loading of the page, and prevent the page from refreshing with the AJAX content after it is loaded. This line must be added to each new JavaScript file that you create, calling your relevant function.

The following figure represents the original WBM page that is loaded when there is no connectivity with the AJAX server.

![Forgotten Password for Wireless Network](image)

**Figure 40.2  Forgotten Password for Wireless Network**
The following figure represents the WBM page when an AJAX update is available.

![Forgotten Password for Wireless Network](image)

**Figure 40.3  Forgotten Password for Wireless Network**

Note that the `forgot_password.js` JavaScript template contains include calls to the `ajax.js` and `ajax_util.js` files. These files contain supporting functions and functions common to multiple files. For example, the implementation of `ajax_exec_on_dom_content_loaded()`, common to both `forgot_password.js` and `web_auth.js`.

Finally, add the new C file name to the AJAX_MODULES list in the Makefile, omitting the `.c` file extension:

```
pkg/jnet/server/ajax/Makefile
AJAX_MODULES+=forgot_password web_auth
```
41
Testing OpenRG's Networking Performance

41.1 Hardware Requirements

The following hardware components are required:

- A development board
- A SmartBits device
- A SmartBits console PC
- Ethernet cables, including one crossover cable for connecting a PC to Smartbits

41.2 Testing OpenRG with SmartBits

SmartBits is a tool for running performance tests, conforming with RFC2544. It can be configured and managed using a software such as "SmartApplications" or "SmartFlow". The following example uses SmartApplications version 2.50. For proper set up and configuration, refer to your software's manuals and help files.

SmartApplications is a simple tool for running performance tests on the SmartBits device. The SmartBits's first port is used for simulating a LAN PC, while the second port is used for simulating a WAN PC. The SmartBits software sends data traffic from one SmartBits port to another through the device under test (OpenRG, in your case), measuring, analyzing, and logging its performance. The following example describes how to run Throughput tests on
OpenRG in its different modes: Routing, NAPT, Bridge, and IPSec. A *Throughput* is the amount of data that may be transferred in a data channel or through a device in one second.

### 41.2.1 Physical Connection and Configuration Reference

Using Ethernet cables, connect the SmartBits device to the console PC and OpenRG as follows.

1. Connect the console PC’s network card to a dedicated port on the SmartBits device, using an Ethernet crossover cable.
2. Connect the SmartBits port #1 to OpenRG’s LAN port.
3. Connect the SmartBits port #2 to OpenRG’s WAN port.
4. Configure the console PC’s network card according to the SmartBits device IP address, and check connectivity between them using the SmartApplications utility.
5. Configure OpenRG and SmartBits according to the test you would like to perform:
• For testing routing performance, refer to Section 41.3.
• For testing NAPT performance, refer to Section 41.4.
• For testing bridge performance, refer to Section 41.5.
• For testing IPSec connection performance, refer to Section 41.6.

### 41.2.2 Preparing OpenRG For Testing

Prior to running any of the mentioned tests, perform the following preparatory steps:

1. **Restore OpenRG to its factory settings:**
   ```
   OpenRG> system restore_factory_settings
   ```

2. **When OpenRG is up and running, disable unused services (for example, File Server, Media Sharing, etc.).**

3. **Disable unused network interfaces (for example, LAN Wireless, LAN USB, DSL/Ethernet in dual WAN).**

4. **Disable the 'IP Address Distribution' and the 'Multicast - IGMP Proxy Internal' options on each of the active network interfaces, unless they are required by your specific setup. To do so, perform the following:**
   
   a. **Under the 'System' tab, click the 'Network Connections' menu item.**
   b. **Click a connection's name to enter its 'Properties' screen.**
   c. **Click the 'Settings' sub-tab, and select the 'Disabled' option from the 'IP Address Distribution' drop-down menu.**
   d. **Click the 'Routing' sub-tab, and deselect the 'Multicast - IGMP Proxy Internal' option.**
   e. **Click 'Apply'.**

### 41.2.3 Setting an ARP Table on OpenRG

When testing OpenRG's routing and NAPT modes, as well as an IPSec connection, it is necessary to associate the Smartbits devices' static IP addresses with its physical MAC addresses, in order to route packets through OpenRG. This is done by setting an Address Resolution Protocol (ARP) table on OpenRG, either from a serial console\(^1\) or by telneting OpenRG, as described here:

1. **Connect a Windows PC to OpenRG's LAN port.**

---

\(^{1}\text{Requires a serial connection to the development board.}\)
2. Access the Windows command line interface by choosing 'Run' from the 'Start' menu, and typing `cmd`.

3. Telnet OpenRG by typing: `telnet 192.168.1.1`, and log in with your username and password.

4. When you see the `OpenRG>` prompt, type `system shell` to enter OpenRG's shell.

5. In the shell, type:
   ```
   arp -s <Smartbits LAN device IP> <Smartbits LAN device MAC>
   arp -s <Smartbits WAN device IP> <Smartbits WAN device MAC>
   ```

For example:
```plaintext
C:\Documents and Settings\johns>telnet 192.168.1.1
Username: admin
Password: *******
OpenRG> system shell
/ # arp -s 192.168.1.10 00:00:00:01:01:01
/ # arp -s 192.168.3.10 00:00:00:01:02:01
```

After configuring the ARP table, disconnect the PC from OpenRG.

### 41.3 Testing OpenRG in Routing Mode

In order to test OpenRG's performance in routing mode, you should configure both OpenRG and SmartBits separately.

#### 41.3.1 OpenRG Configuration

To configure OpenRG's routing mode:

1. Log into OpenRG's WBM.

2. Click the 'Network Connections' menu item under the 'System' tab. If a bridge exists, remove it by clicking the 'Remove' action icon. The WAN and LAN connections should be independent.

![Network Connections](image)

**Figure 41.2 Network Connections**

3. Verify that the LAN Ethernet parameters are at their default values:
a. In the 'Network Connections' screen under 'System', click the 'LAN Ethernet' link, and then click the 'Settings' sub-tab.

b. In the 'Internet Protocol' section, the combo box should be set to "Use the following IP Address", the IP address should be 192.168.1.1, and the subnet mask should be 255.255.255.0.

![Figure 41.3 LAN Ethernet Properties](image1)

4. Set the WAN Ethernet parameters:

   a. Click the 'WAN Ethernet' link, and then click the 'Settings' sub-tab.

   b. In the 'Internet Protocol' section, select "Use the following IP Address" and enter 192.168.3.5 as the IP address, 255.255.255.0 as the subnet mask, and 192.168.3.1 as the default gateway. Click 'Apply' to save the settings.

![Figure 41.6 WAN Ethernet Properties](image2)

c. Click the 'Routing' sub-tab, and select 'Route' in the 'Routing Mode' drop-down menu. Click 'Apply' to save the settings.
5. Stop OpenRG's firewall, either from a serial console (requires a serial connection to the development board) or by telneting OpenRG, as described here.

a. Access the Windows command line interface by choosing 'Run' from the 'Start' menu and typing `cmd`.

b. Telnet OpenRG by typing `telnet 192.168.1.1`, and log in with your username and password.

```
C:\Documents and Settings\johns> telnet 192.168.1.1
Username: admin
Password: *****
```

c. Enter the shell, and type `firewall stop`.

```
OpenRG> system shell
/ # firewall stop
```

d. Verify that the firewall is not running on any device under test:

```
OpenRG> firewall dump -pd
```

6. Configure OpenRG's ARP table as described in Section 41.2.3.

### 41.3.2 SmartBits Configuration

1. Start SmartApplications and verify that it is connected.

2. Select the source and destination cards (see Figure 41.9):

   • The source card, representing the first port connected to OpenRG's LAN, will simulate the LAN host. Select (01,01,01).
- The destination card, representing the second port connected to OpenRG's WAN, will simulate the WAN host. Select (01,02,01).

3. Press the ">" button to create a test pair from the chosen cards.

![Figure 41.9 SmartApplications Main Screen](image)

4. Configure both cards by selecting their respective tabs.

   a. The following parameters should be configured for **both** cards:

   **Bi-directional** Select this check box for testing bidirectional data traffic.

   **Reverse Tx Rx port** Press this button to reverse the traffic direction when testing unidirectional data traffic. The default direction is from the source card to the destination card (represented by the arrow between the test pairs).

   **Signal Rate** Select 100M.

   **Duplex** Select between full or half duplex according to your test requirements.

   **Protocol** Select UDP.

   **Auto Negotiation** Select Force.

   b. The following parameters should be configured for the LAN card (see **Figure 41.10**):
**Destination MAC**  This is OpenRG's LAN MAC address. To obtain this address, click the 'LAN Ethernet' link in OpenRG's 'Network Connections' screen under 'System'. The 'LAN Ethernet Properties' screen will appear, displaying the MAC address.

**SmartCard MAC**  This value is auto-filled by the application and can be left at default: 000000010101.

**SmartCard's IP**  This parameter should be in the same subnet as OpenRG's LAN: 192.168.1.10.

**Router's IP**  This parameter should be OpenRG's LAN IP: 192.168.1.1.

![Figure 41.10 SmartApplications LAN Card](image1)

---

c. The following parameters should be configured for the WAN card (see **Figure 41.11**):

**Destination MAC**  This is OpenRG's WAN MAC address. To obtain this address, click the 'WAN Ethernet' link in OpenRG's 'Network Connections' screen under 'System'. The 'WAN Ethernet Properties' screen will appear, displaying the MAC address.

**SmartCard MAC**  This value is auto-filled by the application and can be left at default: 000000010201.

**SmartCard's IP**  This parameter should be in the same subnet as OpenRG's WAN: 192.168.3.10.

**Router's IP**  This parameter should be OpenRG's WAN IP: 192.168.3.5.

![Figure 41.11 SmartApplications WAN Card](image2)
5. Configure the following advanced options, by choosing 'Test Configuration' from the 'Setup' toolbar option.

- Decrease the 'Learning Retries' to 1.
- Set the 'Duration' to 120 seconds.
- All other values should remain at default.

![Setup Test Configuration](image)

**Figure 41.12 Setup Test Configuration**

6. You can configure the packet sizes to be used in the test, by pressing the 'Sizes' button on the Test Configuration screen. The defaults contain a variety of sizes, improving the test's versatility. Change the resolution percentage for all packet sizes to 0.01.

![Custom Packet Sizes](image)

**Figure 41.13 Custom Packet Sizes**

7. Run the test by pressing the "Throughput" button.
41.4 Testing OpenRG in NAPT Mode

To configure OpenRG's NAPT mode:

1. Log into OpenRG's WBM.

2. Click the 'Network Connections' menu item under the 'System' tab. If a bridge exists, remove it by clicking the 'Remove' action icon. The WAN and LAN connections should be independent.

3. Verify that the LAN Ethernet parameters are at their default values:
   a. In the 'Network Connections' screen under 'System', click the 'LAN Ethernet' link, and then click the 'Settings' sub-tab.
   b. In the 'Internet Protocol' section, the combo box should be set to "Use the following IP Address", the IP address should be 192.168.1.1, and the subnet mask should be 255.255.255.0.
   c. Click the 'Routing' sub-tab, and verify that the 'Routing Mode' drop-down menu is set to "Route".
   d. Click the 'Advanced' sub-tab, and verify that the 'Internet Connection Firewall' check box is deselected.
4. Set the WAN Ethernet parameters:

   a. Click the 'WAN Ethernet' link, and then click the 'Settings' sub-tab.

   b. In the 'Internet Protocol' section, select "Use the following IP Address" and enter 192.168.3.5 as the IP address, 255.255.255.0 as the subnet mask, and 192.168.3.1 as the default gateway. Click 'Apply' to save the settings.

   c. Click the 'Routing' sub-tab, and select 'NAPT' in the 'Routing Mode' drop-down menu. Click 'Apply' to save the settings.

   d. Click the 'Advanced' sub-tab, and deselect the 'Internet Connection Firewall' check box. Click 'Apply' to save the settings.

5. Set a Port Forwarding rule:

   a. In the 'Firewall' menu item under 'Services' select the 'Port Forwarding' option, and click the 'New Entry' link. The 'Add Port Forwarding Rule' screen appears.
6. Configure OpenRG's ARP table as described in Section 41.2.3.

Prior to running the NAPT test, configure SmartBits exactly as in the routing test (refer to Section 41.3.2).

41.5 Testing OpenRG in Bridge Mode

In order to test OpenRG's performance in bridge mode, you should configure both OpenRG and SmartBits separately.

41.5.1 OpenRG Configuration

To configure OpenRG's bridge mode:

1. In most cases, OpenRG is pre-configured with a LAN bridge, spanning over the LAN Ethernet and the LAN wireless access point. To test OpenRG's bridge mode, add the WAN Ethernet as an underlying device:

   a. Log into OpenRG's WBM.
b. Enter the 'Network Connections' menu item under the 'System' tab, and click the 'Advanced' button for a more detailed view.

![Network Connections](image1)

**Figure 41.22 Network Connections**

c. Click 'LAN Bridge' and then click the 'Bridging' sub-tab. Select the 'WAN Ethernet' check box, and click 'OK'.

![LAN Bridge Properties](image2)

**Figure 41.23 LAN Bridge Properties**

d. Click 'OK' on the confirmation screen. You may need to log in again to OpenRG.

e. Click the 'Network Connections' menu item again. The updated 'Network Connections' screen appears, displaying the bridge spanning over the WAN Ethernet.

![Network Connections](image3)

**Figure 41.24 Network Connections**
f. In the 'Routing' screen under 'System', set the 'Software Acceleration' option to 'High'. Click 'Apply' to save the settings.

2. Stop OpenRG's firewall, either from a serial console (requires a serial connection to the development board) or by telneting OpenRG, as described here.
   
a. Access the Windows command line interface by choosing 'Run' from the 'Start' menu and typing `cmd`.

b. Telnet OpenRG by typing `telnet 192.168.1.1`, and log in with your username and password.

```
C:\Documents and Settings\johns> telnet 192.168.1.1
Username: admin
Password: *****
```

c. Enter the shell, and type `firewall stop`.

```
OpenRG> system shell
# firewall stop
```

d. Verify that the firewall is not running on any device under test:

```
OpenRG> firewall dump -pd
```

### 41.5.2 SmartBits Configuration

1. Start SmartApplications and verify that it is connected.

2. Select the source and destination cards (see Figure 41.25):
   
   - The source card, representing the first port connected to OpenRG's LAN, will simulate the LAN host. Select (01,01,01).
   - The destination card, representing the second port connected to OpenRG's WAN, will simulate the WAN host. Select (01,02,01).

3. Press the ">" button to create a test pair from the chosen cards.
4. Configure both cards by selecting their respective tabs.

   a. The following parameters should be configured for both cards:

      **Bi-directional**  Select this check box for testing bidirectional data traffic.

      **Reverse Tx_Rx port**  Press this button to reverse the traffic direction when testing unidirectional data traffic. The default direction is from the source card to the destination card (represented by the arrow between the test pairs).

      **Signal Rate**  Select 100M.

      **Duplex**  Select between full or half duplex according to your test requirements.

      **Protocol**  Select UDP.

      **Auto Negotiation**  Select Force.

   b. The following parameters should be configured for the LAN card (see Figure 41.26):

      **Destination MAC**  This auto-filled value is the SmartBits other card MAC address, and can be left at default: 000000010201.

      **SmartCard MAC**  This auto-filled value is the SmartBits current card MAC address, and can be left at default: 000000010101.
**SmartCard's IP**  This parameter is not used in bridge mode. It can be left in the same subnet as OpenRG's LAN: 192.168.1.10.

**Router's IP**  This parameter is not used in bridge mode. It can be left as OpenRG's LAN IP: 192.168.1.1.

---

**Destination MAC**  This auto-filled value is the SmartBits other card MAC address, and can be left at default: 000000010101.

**SmartCard MAC**  This auto-filled value is the SmartBits current card MAC address, and can be left at default: 000000010201.

**SmartCard's IP**  This parameter is not used in bridge mode. It can be left in the same subnet as OpenRG's WAN: 192.168.3.10.

**Router's IP**  This parameter is not used in bridge mode. It can be left as OpenRG's WAN IP: 192.168.3.5.

---

5. Configure the following advanced options, by choosing 'Test Configuration' from the 'Setup' toolbar option.

   • Decrease the 'Learning Retries' to 1.
   
   • Set the 'Duration' to 120 seconds.
• All other values should remain at default.

Figure 41.28 Setup Test Configuration

6. You can configure the packet sizes to be used in the test, by pressing the 'Sizes' button on the Test Configuration screen. The defaults contain a variety of sizes, improving the test’s versatility. Change the resolution percentage for all packet sizes to 0.01.

Figure 41.29 Custom Packet Sizes

7. Run the test by pressing the "Throughput" button.
### 41.6 Testing OpenRG's IPSec Connection

In order to test OpenRG's performance through an IPSec connection, you must first establish such a connection, using two gateways running OpenRG and connected back to back through the WAN, as shown in Figure 41.30.

![Figure 41.30 Connecting SmartBits to an OpenRG IPSec Connection](image)

Note: The LAN IP address of Gateway B should be in a different subnet. In this example, it is set to 192.168.3.1 (see Figure 41.30).

#### 41.6.1 OpenRG Configuration

Assuming both gateways are running OpenRG, their configurations are the same, except for their IP addresses. This section describes only the configuration of Gateway A.

1. Log into OpenRG's WBM.

2. Click the 'Network Connections' menu item under the 'System' tab. If a bridge exists, remove it by clicking the 'Remove' action icon. The WAN and LAN connections should be independent.
3. Verify that the LAN Ethernet parameters are at their default values:

a. In the 'Network Connections' screen under 'System', click the 'LAN Ethernet' link, and then click the 'Settings' sub-tab.

b. In the 'Internet Protocol' section, the combo box should be set to "Use the following IP Address", the IP address should be 192.168.1.1, and the subnet mask should be 255.255.255.0.

c. Click the 'Routing' sub-tab, and verify that the 'Routing Mode' drop-down menu is set to "Route".

d. Click the 'Advanced' sub-tab, and verify that the 'Internet Connection Firewall' check box is deselected.

4. Stop OpenRG's firewall, either from a serial console (requires a serial connection to the development board) or by telneting OpenRG, as described here.

a. Access the Windows command line interface by choosing 'Run' from the 'Start' menu and typing `cmd`. 
b. Telnet OpenRG by typing `telnet 192.168.1.1`, and log in with your username and password.

```bash
C:\Documents and Settings\johns> telnet 192.168.1.1
Username: admin
Password: ******
```

c. Enter the shell, and type `firewall stop`.

```
OpenRG> system shell
/ # firewall stop
```

d. Verify that the firewall is not running on any device under test:

```
OpenRG> firewall dump -pd
```

5. Set the WAN Ethernet parameters:

a. Click the 'WAN Ethernet' link, and then click the 'Settings' sub-tab.

b. In the 'Internet Protocol' section, select "Use the following IP Address" and enter 10.10.10.10 as the IP address (10.10.10.11 on Gateway B), 255.255.255.0 as the subnet mask, and 10.10.10.11 as the default gateway (10.10.10.10 on Gateway B). Click 'Apply' to save the settings.

![Figure 41.35 WAN Ethernet Properties](image)

```
<table>
<thead>
<tr>
<th>Internet Protocol</th>
<th>Use the Following IP Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP Address:</td>
<td>10 10 10 10</td>
</tr>
<tr>
<td>Subnet Mask:</td>
<td>255 255 255 0</td>
</tr>
<tr>
<td>Default Gateway:</td>
<td>10 10 10 11</td>
</tr>
</tbody>
</table>
```

![Figure 41.36 WAN Ethernet Properties](image)

c. Click the 'Routing' sub-tab, and select 'Route' in the 'Routing Mode' drop-down menu. Click 'Apply' to save the settings.

![Figure 41.36 WAN Ethernet Properties](image)

d. Click the 'Advanced' sub-tab, and deselect the 'Internet Connection Firewall' check box. Click 'Apply' to save the settings.

![Figure 41.37 WAN Ethernet Properties](image)

6. Create an IPSec connection to the second gateway. To learn how to do so, refer to the 'IPSec Gateway-to-Gateway Connection Scenario' section of the OpenRG Administrator
Manual. After the connection is created, enter its 'IPSec' sub-tab, and select your preferred encryption settings.

7. Configure OpenRG's ARP table as described in Section 41.2.3.

### 41.6.2 SmartBits Configuration

1. Start SmartApplications and verify that it is connected.

2. Select the source and destination cards (see Figure 41.38):
   - The source card, representing the port connected to OpenRG A’s LAN, will simulate a LAN host. Select (01,01,01).
   - The destination card, representing the port connected to OpenRG B’s LAN, will simulate a second LAN host. Select (01,02,01).

3. Press the ">" button to create a test pair from the chosen cards.

4. Configure both cards by selecting their respective tabs.
   - The following parameters should be configured for **both** cards:
     - **Bi-directional** Select this check box for testing bidirectional data traffic.
Reverse Tx_Rx port  Press this button to reverse the traffic direction when testing unidirectional data traffic. The default direction is from the source card to the destination card (represented by the arrow between the test pairs).

Signal Rate  Select 100M.

Duplex  Select between full or half duplex according to your test requirements.

Protocol  Select UDP.

Auto Negotiation  Select Force.

b. The following parameters should be configured for the first LAN card (see Figure 41.39):

Destination MAC  This is OpenRG A's LAN MAC address. To obtain this address, click the 'LAN Ethernet' link in OpenRG's 'Network Connections' screen under 'System'. The 'LAN Ethernet Properties' screen will appear, displaying the MAC address.

SmartCard MAC  This value is auto-filled by the application and can be left at default: 000000010101.

SmartCard's IP  This parameter should be in the same subnet as OpenRG's LAN: 192.168.1.10.

Router's IP  This parameter should be OpenRG's LAN IP: 192.168.1.1.

c. The following parameters should be configured for the second LAN card (see Figure 41.40):

Destination MAC  This is OpenRG B's LAN MAC address. To obtain this address, click the 'LAN Ethernet' link in OpenRG's 'Network Connections' screen under 'System'. The 'LAN Ethernet Properties' screen will appear, displaying the MAC address.

SmartCard MAC  This value is auto-filled by the application and can be left at default: 000000010201.
**SmartCard's IP**  This parameter should be in the same subnet as OpenRG's LAN: 192.168.3.10.

**Router's IP**  This parameter should be OpenRG's LAN IP: 192.168.3.1.

![Figure 41.40 SmartApplications OpenRG B LAN Card](image)

5. Configure the following advanced options, by choosing 'Test Configuration' from the 'Setup' toolbar option.

   - Decrease the 'Learning Retries' to 1.
   - Set the 'Duration' to 120 seconds.
   - All other values should remain at default.

![Figure 41.41 Setup Test Configuration](image)
6. You can configure the packet sizes to be used in the test, by pressing the 'Sizes' button on the Test Configuration screen. The defaults contain a variety of sizes, improving the test's versatility. Change the resolution percentage for all packet sizes to 0.01.

![Custom Packet Sizes](image)

**Figure 41.42 Custom Packet Sizes**

7. Run the test by pressing the "Throughput" button.
42
Supporting Separate Networks with Routing Groups

42.1 Overview

"Routing Groups" is the ability to support several completely separated "routing entities" or "networks" on the same OpenRG gateway. A real life example of such a group is FON (or any other NationZone variant), which has the following limitations:

- Dedicated WAN and LAN interfaces
- Traffic of FON clients should only use the dedicated interfaces
- There is no connectivity between FON clients and the OpenRG home network

Routing Groups maps and answers all the routing needs of the evolving world of home/SOHO/SMB gateways.

42.2 Routing Groups

A routing group is a set of one or more devices that should be fully routable between themselves (under security and functionality limitations), and completely non-routable to and from other routing groups.

Available routing group types include:

- The main group – this is the traditional group of devices that is used by the basic gateway services. It includes (as a minimum) the LAN interfaces serving the home network, and a WAN interface that is used by home users and most gateway applications to reach the
Internet. The main group may also include additional WAN interfaces acting as default gateways, with optional failover and/or load balancing between them.

- **Non-forwarding devices** – a special group of all devices that can be used only for local delivery of packets and are not routable to other devices/networks (e.g. a WAN interface dedicated to VoIP—all packets arriving on it must go up to Asterisk and never be routed).

- **Other groups** – groups defined per project requirements. Each group contains one or more devices. Each group is generally used for a service that is external to the home network. There is no failover/load balancing between devices in "other" groups, so normally such a group contains only a single default gateway. An example of such a network is the FON network (or any other NationZone variant).

### 42.3 Routing Tables

The default Linux configuration (non-OpenRG) contains three routing tables:

- **Local** – for serving packets destined to the gateway itself.

- **Main** – the "usual" routing table that contains automatic routes to directly connected subnets, default gateway routes, and everything else inserted to it manually.

- **Default** – empty in the default configuration. As part of the multi-WAN implementation, all OpenRG's default gateway routes are currently in this routing table.

Other non-standard tables used in OpenRG:

- **Device tables** – a table for each device, which contains a route to the directly connected subnet of this device and the default gateway route using this device.

- **Group tables** – a table for each group that contains all routes using the devices which are part of this group (it is a super set of all device tables of devices that belong to this group).

The Main table holds routing entries for all non-special interfaces, except default gateway routes (these are stored in the Default table or in special group tables, in order to support load balancing and failover scenarios). There is a group table for all non-forwarding devices, but there is no rule directing to this table.

### 42.4 Adding Routes to Tables

Special care is taken to insert routes to the correct tables:

- All routes are added to the correct group table and to the correct device table.

- Adding an IP to the device automatically adds a route of the directly connected subnet to the main table (this is the default Linux behavior). This route is immediately removed and added to the correct group table, and also to the device table.
• Default gateway routes are always added to the device table, as well as to the group table if it is not main, or to default table if it is.

## 42.5 Routing Decision Rules

The following is a list of the rules required for making the routing decision along with the content of the route tables. The rules are listed in their order of priority.

1. Locally destined packets:
   from all lookup local

2. Incoming packets to devices that belong to "other" groups (not main and not non-forwarding) are matched according to their incoming interface and redirected to the group table for routing inside the group:
   from all iif dev1 lookup 2
   from all iif dev2 lookup 2
   ...
   from all iif dev7 lookup 4

3. Fallback for 2: For incoming packets to devices that do not belong to main (i.e. "other" and non-forwarding) that were not served by 2, the answer is "unreachable", so that they will not be served by the main table:
   from all iif dev1 unreachable
   from all iif dev2 unreachable
   ...
   from all iif dev7 unreachable

4. For all traffic generated by sockets bound to a device on the gateway, use the device's table (this rule is added for all devices, but it is not important for devices that belong to the main group, because they are served by the main and default tables which contain the relevant routes, and only a route with the correct interface can be used if the traffic is bound to a device):
   from all oif dev1 lookup 21
   from all oif dev2 lookup 22
   ...
   from all oif dev7 lookup 27

5. For traffic generated on the gateway with its source IP matching the IP of a non-forwarding device, use the device table (this traffic cannot be routed to other devices):
   from 192.168.71.10 lookup 23
   ...

6. For traffic generated on the gateway with its source IP matching the IP of a device that belongs to one of the "other groups" (not main and not non-forwarding), use the group table of this device (as a substitute to main):
   from 192.168.55.10 lookup 2
   from 192.168.66.10 lookup 2
   ...
7. Fallback for 5,6: For traffic generated on the gateway with its source IP matching the IP of a device that does not belong to main – unreachable. This together with 5,6 ensures correct routing for traffic generated on the gateway that is not bound to an interface and does not belong to the main table (and hence that should not use it):
   from 192.168.71.10 unreachable
   from 192.168.55.10 unreachable
   from 192.168.66.10 unreachable
   ...

8. Serving traffic of the main group (other groups' traffic handled by rules before):
   from all lookup main

9. For traffic generated on the gateway with its source IP matching the IP of a device from the main group, use its device table. The purpose of these rules is (in a state of multi WANs, each serving as a default gateway) that traffic generated on the gateway will use the correct interface to access the Internet (based on source IP):
   from 192.168.1.1 lookup 31
   from 192.168.81.10 lookup 32
   ...

10. The default rule:
    from all lookup default
Part V  Appendix
### Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>43. Working with the Concurrent Versions System</td>
<td>527</td>
</tr>
<tr>
<td>44. Using Eclipse IDE for OpenRG Development</td>
<td>534</td>
</tr>
<tr>
<td>45. General Public License (GPL)</td>
<td>537</td>
</tr>
<tr>
<td>46. Converting Old Makefiles</td>
<td>539</td>
</tr>
<tr>
<td>47. Contact Jungo</td>
<td>542</td>
</tr>
</tbody>
</table>
43

Working with the Concurrent Versions System

43.1 What is CVS?

Concurrent Versions System (CVS) is a version control system. The CVS enables you to record the history of your source files. For example, bugs sometimes creep in when software is modified, which you may not detect until a long time after you had made the modification. With CVS, you can easily retrieve old versions to discover the change that caused the bug.

Using CVS allows you to save every version of every file you have ever created. CVS stores all versions of a file in a single file, by only recording the differences between versions. CVS is widely deployed and used to synchronize most, if not all, open source projects, such as sourceforge.net or kernel.org.

43.2 The Advantages

CVS can assist you if you are part of a group of people working on the same project. It is all too easy to overwrite each others’ changes unless you are extremely careful. Some editors, like GNU Emacs, try to ensure that the same file is never modified by two people at the same time. Unfortunately, if someone is using another editor, this safeguard will not work. CVS solves this problem by insulating the different developers from each other. Every developer works in his own directory, and CVS merges the work when each developer is done.

Of the commercial and non-commercial source code control products, CVS is an obvious choice: the software is free (GNU), it supports many platforms, it comes with all the source, and it has ample support via mailing lists or Usenet news groups. Also, its versions database engine is Vanilla Revision Control System (RCS) engine, which has been proven successful.
Incidentally, since RCS revision files are simply ASCII and RCS operation is mostly straightforward, manual troubleshooting is easy. This should be considered, even if a proper backup strategy is being employed.

CVS adds the notion of "modules", or hierarchical collections of files, to the flat file space that RCS handles. CVS operation is not limited to single files but can actually refer to a module in its entity. There is no locking mechanism in CVS. Developers work concurrently in private areas (sandboxes) and merge their work with that of co-workers on demand. This is a positive feature of CVS, but it requires overhead in terms of communication between project teams, and a procedure for integrating changes into the source base. These invaluable advantages are the reason Jungo has selected CVS to control and maintain OpenRG's source code.

43.3 Setting up CVS on Your Computer

This section describes how to set up a CVS Main Trunk on your computer. The main trunk is the default branch that will contain both Jungo's source code and your own modifications and additions. Enable the CVS via the command line by performing the following:

**Warning: Perform the following commands in their exact order of appearance.**

1. Change to super user mode:
   ```bash
   ~$ su
   ```

2. Install the required CVS utilities:
   ```bash
   # apt-get install cvs cvsutils
   ```

3. Change the `/etc/xinetd.d/cvs` file to resemble the following:
   ```plaintext
   service cvspserver
   {
     disable = no
     socket_type = stream
     protocol = tcp
     wait = no
     user = root
     server = /usr/sbin/cvspserver
   }
   ```

4. Modify the CVS configuration file `/etc/cvs/cvs.conf` to include the following line:
   ```plaintext
   CVS_REPOS="/cvsroot"
   ```

5. Create a CVS repository by performing the following:
   ```bash
   # mkdir /cvsroot
   # cvs -d /cvsroot init
   # mkdir -p /cvsroot/rg
   # chmod -R 777 /cvsroot
   # exit
   ```

6. On any host you would like to access CVS, add the following line to the `/etc/profile` file or to your `.bashrc` file.
   ```bash
   export CVSROOT=:pserver:<user_name>@<host_name>:/cvsroot
   ```
7. Open a new shell and verify that $CVSROOT is defined.

8. Log into the CVS server:

   ```bash
   ~$ cvs login
   ```

9. Check that the correct file types are defined for all the binary files, including *.o, *.so and *.gif.

   ```bash
   ~$ cvs co CVSROOT
   ~$ cd CVSROOT
   ~/CVSROOT$ gvim cvswrappers
   ```

10. The binary files should include the following file extensions (you can download the sample cvswrappers file from the Jungo Web site at http://www.jungo.com/openrg/download/cvswrappers.sample).

   ```
   *.[jJ][pP][gG] -k 'b' # *.jpg
   *.[jJ][pP][eE][gG] -k 'b' # *.jpeg
   *.[gG][iI][fF] -k 'b' # *.gif
   *.so -k 'b' # linux shared object
   *.so.* -k 'b' # linux shared object
   *.a -k 'b' # linux library
   *.o -k 'b' # linux object
   ```

11. Check in the updated cvswrappers file:

   ```bash
   ~/CVSROOT$ cvs commit cvswrappers
   ```

   The following figure depicts the basic main trunk, created on your computer.

   ![Figure 43.1 CVS Main Trunk](image)

### 43.4 Initial Import of OpenRG into CVS

This section describes how to import the OpenRG distribution into a separate branch—vendor-branch, which is the branch that will contain the source code distribution. Modifications to the software, however, should be performed on the main trunk of CVS.

Note: All CVS actions must be performed on a CVS client host, and not on the server.
43.4.1 Downloading JCVS

JCVS is a tool provided by Jungo, which prepares the source tree for commit into vendor-branch. The JCVS tool is available for download at http://www.jungo.com/openrg/download/jcvs. Copy the JCVS tool into /usr/local/bin. Make sure that $PATH includes /usr/local/bin, by adding it either to the /etc/profile or to your ~/.bashrc (if it is not already included).

43.4.2 Downloading the Source Code Packages

To download the development tree source code packages, run install.sh with the license(s) provided, and extract all of them to ~/rg-4.3.5. This will install a distribution of OpenRG version 4.3.5 on your home directory, under the ~/rg-4.3.5 directory. Use the following procedure to import the OpenRG source tree into CVS. The initial release of 4.3.5 will be committed on a 4.3 vendor-branch for version 4.3.

Note: The mentioned version numbers should be regarded as examples only.

43.4.3 Importing the Source Tree into CVS

Importing the OpenRG source tree into CVS requires:

1. Creating vendor-branch.
2. Commiting the source code.
3. Creating a tag (a logic timestamp on the branch).

Perform these actions by running the following commands:

```bash
$ cd ~/rg-4.3.5
$ jcvs importprep -b vendor-jungo-4_3 rg
$ cvs commit
$ cvs tag tag-jungo-4_3_5
```

Note: A distribution may contain several architectures (for example, UML and IXP425), therefore you should repeat the installation process until all architectures have been installed.

The source tree is now loaded into the CVS vendor-branch vendor-jungo-4_3, which you can check out on any computer configured to access the CVS server.
The final step is to port the source tree into the main trunk.

### 43.4.4 Porting the Source Tree into the Main Trunk

Any further modification to the source code should be performed on the main trunk. Therefore, you must port the source tree into the CVS main trunk. This also allows an easy comparison to the distribution source code. To port the source code, perform the following:

```
$ cvs co rg
$ cd rg
/ rg$ cvs up -j tag-jungo-4_3_5
/ rg$ cvs commit
```

### 43.5 Importing Newer Versions

When you receive a newer version of an OpenRG distribution, you should import it into the vendor-branch and merge the changes into the main trunk. The following procedure...
describes loading a newer version to the vendor-branch. Assuming you have installed a newer
distribution of OpenRG, for example version 4.3.7, on your home directory under the dev
directory (~dev/openrg), use the following procedure to import the OpenRG source tree into
CVS.

```
$ cd ~/dev/openrg
$ cvs co -d rg -r jungo-vendor-branch rg
$ jcvs importprep -b vendor-jungo-4_3
$ cvs commit
$ cvs tag tag-jungo-4_3_7
```

Merging the changes between version 4.3.5 and 4.3.7 into the main trunk can be done using the
following commands:

```
~$ cvs co rg
~$ cd rg
~/rg$ cvs up -j openrg-4_3_5 -j openrg-4_3_7
```

You should resolve any conflicts that may occur due to changes made in both OpenRG and
your own development. Once all conflicts are resolved, build the new tree and test it. If the
tests pass, you can commit the new tree into CVS.

```
~/rg$ ./rg_config.sh
~/rg$ make
~/rg$ cvs commit
```

![Diagram](image.png)

**Figure 43.4 Importing A New Version**

### 43.6 Generating Distribution Upgrade Patch Files

The `gen_maint_patch.sh` utility located in `rg/pkg/tools` enables you to upgrade your OpenRG
distribution with enhancements and bugfixes of the newer version provided by Jungo. To run
this utility, you must have two RG trees—your current development tree (for example, version
4.2.5) and the newer one (for example, version 4.2.8). When executed, `gen_maint_patch.sh`
compares between the two distributions, and creates the following two files:

- A patch containing a `diff` *(PATCH_FILE_NAME)* between your current distribution and
  the newer one.
- A tarball achieve with the new binary files *(BINTGZ_FILE_NAME)*
When the patches are installed, the changes indicated by `diff` and the new binary files should be merged with the older distribution's source files.

Note: In order to avoid conflicts during the merge, verify that the source files in both of the development trees have not been modified.

To perform the upgrade, proceed as follows:

1. In the old distribution, change to the `rg/pkg/tools` directory.

2. Run the `gen_maint_patch.sh` utility:
   ```
   $ gen_maint_patch.sh <path to the old RG directory> <path to the new RG directory>
   ```
   The `diff` file and the new binaries archive are created.

3. Change to the `rg` directory.

4. Patch the old distribution with the changes indicated by `diff`:
   ```
   $ patch -p0 < PATCH_FILE_NAME
   ```

5. Extract the new distribution's binary files from the tarball archive:
   ```
   $ tar xvzf BINTGZ_FILE_NAME
   ```
44

Using Eclipse IDE for OpenRG Development

Eclipse Integrated Development Environment (IDE) provides a vendor-neutral open development platform. This chapter takes you through the Eclipse installation steps, and then explains how to create, configure and build an OpenRG project on Eclipse.

44.1 Installation

Note: Make sure that you are using Sun's Java Runtime Environment.

Follow these instructions to install Eclipse:

1. Install Eclipse Version 3.2 or newer. If your Linux version has the command `apt-get`, you can try logging in as super-user, and running

   ```sh
   # apt-get install eclipse
   ```

   Otherwise, you can download the software from: [http://www.eclipse.org/downloads/](http://www.eclipse.org/downloads/).

2. Run Eclipse. It is recommended to run Eclipse with parameters that increase the size of the Java heap, for instance:

   ```sh
   # eclipse -vmargs -Xmx256M
   ```

   These parameters will change the maximal size of the Java heap to 256M.

3. Install the C/C++ Development Tools (CDT) Eclipse add on, from version 3.0.2 and up:

   - From the 'Help' menu choose 'Software Updates->Find and Install'
• Choose 'Search for new features to install'

• Choose 'New Remote Site'

• In the Name box type 'CDT'

• In the URL box type: http://download.eclipse.org/tools/cdt/releases/eclipse3.1

• Choose 'OK'

• Keep choosing 'Finish', 'OK', 'Next', 'I accept' and 'Install' until installation is complete. When asked to 'Select the features to install', choose 'CDT'.

44.2 Creating an OpenRG Project on Eclipse

1. Before creating a project on Eclipse, make sure that there are no build directories in your OpenRG directory. From your shell, go to the root directory of your OpenRG tree and type:

   # rm -rf build*

2. In Eclipse: from the 'File' menu choose 'New->Project'.

3. Choose 'C->Standard Make C Project' and choose 'Next'.

4. In the 'Project name' box type 'OpenRG'. In 'Project contents' make sure to UNMARK the 'use default' box. In the 'Directory' box type the root directory of your OpenRG tree. Choose 'Next'.

5. In the 'C/C++ Indexer' tab make your choice of indexer. Note that since the OpenRG tree contains plenty of source files, indexing the entire tree may take a few hours. Therefore it is suggested to select 'No Indexer' for this option. Choose 'Finish'.

6. Exclude all build directories from the project:

   • Choose the 'OpenRG' project.

   • From the 'File' menu choose 'Properties'.

   • Choose 'C/C++ Project Paths'.

   • Under OpenRG, double-click 'Exclusion filter'.

   • Choose 'Add'.

   • In the 'Exclusion pattern' box type 'build*'
7. From the 'Project' menu choose 'Create Make Target'.

8. In the 'Create a New Make Target' window that opens type 'config' in the 'Target Name' box. Type the following configuration parameters in the 'Make Target' box:

```
cfg DIST=<DIST NAME> LIC=<LICENSE FILE>
```

Replace `<DIST NAME>` with the distribution name, e.g. MONTEJADE. Replace `<LICENSE FILE>` with the license file name with the complete path, e.g. `/home/john/rg-4.2.2/pkg_iXP425.lic`. Choose 'Create'.

9. Once again, choose 'Create Make Target' from the 'Project' menu.

10. Keep the default values, i.e. 'all' in the 'Target Name' and the 'Make Target' boxes. Choose 'Create'.

### 44.3 Project Configuration

1. Single-click the 'OpenRG' project (on the left) to select it.

2. From the 'Project' menu choose 'Build Make Target'.

3. Choose 'config' and press 'Build'.

### 44.4 Building Your Project

1. Single-click the 'OpenRG' project (on the left) to select it.

2. From the 'Project' menu choose 'Build Make Target'.

3. Choose 'all' and press 'Build'.

---

Note: Eclipse is rather slow and cumbersome. It is therefore recommended to exclude all unnecessary directories from the project. You might, for instance, choose to exclude directories Jungo, os, pkg/gdb and pkg/ulibc.
45

General Public License (GPL)

45.1 Licensing Acknowledgement and Source Code Offering

The OpenRG/OpenSMB product may contain code that is subject to the GNU General Public License (GPL), GNU Lesser General Public License (LGPL), and BSD (BSDS) license. The OpenRG/OpenSMB Open Source and GNU Public Licenses page contains:

- With respect to GPL/LGPL: the code package names, license types and locations for the license files, and

- With respect to BSD (BSDS): the code package names with the license texts.

To receive the source code of the GPL/LGPL packages, refer to http://www.jungo.com/openrg/download_gpl.html.

45.2 Installing OpenRG’s GPL Distribution

As required by the GPL license, Jungo provides a GPL distribution. To install this distribution, refer to the original OpenRG distribution received from Jungo and locate the GPL license (jpkg_<arch>_gpl.lic) file. Next, perform the following:

```
$ mkdir rg-5.5.0-gpl
$ cd rg-5.5.0-gpl
$ jpkg -x -T jpkg_<arch>_gpl.lic
```

This will create the rg directory, into which it will extract the GPL packages. To compile this GPL distribution, execute:
45.3 Creating Your GPL Distribution

When developing a new feature for OpenRG, you may add or link the feature's code to specific source code areas, which are used in OpenRG under the GPL. As stated by the Free Software Foundation, a piece of code, added to the GPL source code or linked to it in any way, should be distributed under the GPL. Therefore, as you develop with OpenRG, you must maintain the GPL distribution with the files containing or linked to the GPL source code, and make them publicly available.

The following procedure assumes that you have installed the GPL distribution provided by Jungo (rg-5.5.0-gpl) as described in the previous section.

After developing with OpenRG, maintain the GPL distribution by performing the following:

1. Use the `diff` command to compare between the OpenRG development tree and the `rg-5.5.0-gpl` tree. Verify that all modifications made to files (containing or linked to the GPL source code) in the development tree are merged into the GPL tree.

2. If you have added files to certain GPL directories in your development tree, add these files to the GPL tree as well.

   Note: Verify that the Makefile has been updated to handle the new files.

3. If you have added GPL packages or packages that are linked with any of the GPL directories to your development tree, they automatically become subject to the GPL. Therefore, you must add them to the GPL tree.

4. Ensure that you had not made changes, which resulted in linking the GPL objects with objects that are under a different license, such as JUNGO. By doing so, you transform Jungo's proprietary files into GPL-licensed files, thus violating Jungo's license. There are several techniques for avoiding GPL and other copyright violations. If you would like to learn these techniques, contact Jungo's support team and describe the nature of your problem (relationship between files, their licenses, the reason for linking them together, and other relevant details).

5. As the last step, verify that the new GPL distribution compiles as a standalone. To compile it in `rg-5.5.0-gpl`, enter the following command:

   ```
   $ make config DIST=<...> CONFIG_RG_GPL=y LIC=<your GPL license file> && make
   ```

   Note: The `CONFIG_RG_GPL` flag does not appear in the compilation instructions provided by the installation script.
46

Converting Old Makefiles

OpenRG's traditional method of build used recursive make. This resulted in build times which were unacceptably large, even when changing a single file. In examining the source of the overly long build times, it became evident that a number of apparently unrelated problems had combined to produce the delay.

The solution was a flat make system, utilizing a single makefile for the entire system. This chapter will help you convert old makefiles to the new method.

46.1 Guidelines

- Change include *envir.mak* and *rg.mak*. It is recommended that you copy from another new Makefile.

- Code after include *rg.mak*: most vars are cleaned away and should not be used.

- LDLIBS (and most LDFLAGS) vars – use new vars instead.

- Do not use CURDIR or PWD, and beware of JMKE_IS_BUILDDIR. The old code would run once for the source directory and once for the build directory in the root compiling directory, and for subdirs only in build-dir. For new code the current directory may be the source directory or the build directory of the root new-jmk directory, and will normally be the root of the tree.

- Search for wildcard or other reference to local directory. Relative path is legal in many JMK_ variables.

  - Usage of JMKE_PWD_* is allowed in the file but not in the target's action.
  
  - Creation rules should have the directory added to the targets and to their dependencies.
- Programs that create files should not create them in the local directory. If necessary, add a command line parameter.

An example:
Old:
```makefile
go_default.c: gen_rg_def
./gen_rg_def
```
New:
GOOD
```makefile
$(JMKE_PWD_BUILD)/rg_default.c: $(JMKE_PWD_BUILD)/gen_rg_def
$< $(@D)
```
BAD
```makefile
$(JMKE_PWD_BUILD)/rg_default.c: $(JMKE_PWD_BUILD)/gen_rg_def
$(JMKE_PWD_BUILD)/gen_rg_def $(@D)
```

- Search for implied order and linearize/parallelize.

- archconfig:: rules are usually a problem – convert to archconfig-tasks. They are a problem because they assume a known order of execution. Also note that with a chain of dependencies the top and all leaves of the chain should be defined as tasks. Refer to pkg/openssl/crypto/objects/Makefile for an example.

Old:
```makefile
archconfig:: obj_dat.h
obj_dat.h: obj_dat.pl obj_mac.h
perl ...
obj_mac.h: ... obj_mac.num
perl ...
obj_mac.num:
...
```
New:
```makefile
JMK_ARCHCONFIG_FIRST_TASKS+=$(JMKE_PWD_BUILD)/obj_dat.h
$(JMKE_PWD_BUILD)/obj_mac.num
```

- ramdisk

- JMK_RAMDISK_FILES should look like exported src__dst instead of absolute path in the ramdisk.

- Direct calls to RAMDISK_CP_*_FUNC are illegal now. Use JMK_RAMDISK_FILES/JMK_RAMDISK_RW_FILES instead.

- Do not use first/other/last/ramdisk tasks. Use dependencies instead, or JMK_RAMDISK_TASKS/JMK_RAMDISK_DIRS.

- If the makefile has phony targets, their names should be unique in the tree.

- You might get unexpected phony prerequisites (e.g. if you are in JMK_ARCHCONFIG_FIRST_TASKS). In such a case, use $(JMKE_DEPS) instead of $^.
• A program that generates more than one file may cause a problem that the generating program may be run twice. Create a fake common target to avoid double running. An example:

Old:

```
load_kernel.c load_kernel_modules.c: gen_load_kernel_modules
   ./gen_load_kernel_modules
```

New:

```
$(JMKE_PWD_BUILD)/load_kernel.c $(JMKE_PWD_BUILD)/load_kernel_modules.c: \ 
   $(JMKE_PWD_BUILD)/.gen_load_kernel_once
$(JMKE_PWD_BUILD)/.gen_load_kernel_once: \ 
   $(JMKE_PWD_BUILD)/gen_load_kernel_modules
   $< $(@D)
   touch $@

JMK_CLEAN+=$(JMKE_PWD_BUILD)/.gen_load_kernel_once
```

• In the past, JMK_O_OBJS JMK_L_OBJS etc. were defaults that were used only for targets in the directory that did not have specific listings. Now they are applied to all targets of the directory. An example:

Old:

```
JMK_O_TARGET=av.o
JMK_O_OBJS=avt.o vendor.o
JMK_LOCAL_TARGET=avd
JMK_O_OBJS_avd=av.o avd.o
```

New:

```
JMK_O_TARGET=av.o
JMK_O_OBJS_av.o=avt.o vendor.o
JMK_LOCAL_TARGET=avd
JMK_O_OBJS_avd=av.o avd.o
```

• Too generic variable names are a problem as variables are shared across the tree. For example, firewall/wbm/Makefile and ppp/wbm/Makefile both had vars called CONN_SETTINGS_OBJS and were changing contents for each other.

• Dependency on an autogenerated 'h' file should be made explicit in order to cause the autogenerated file to be created.

46.2  API Notes

JMK_LIBS_<target> contains full path libraries needed for <target>. For libraries that are not part of the rg-tree or libc, use __local_<lib>. This will add -l<lib> to the linkage flags but will not add any dependency.
47

Contact Jungo

For additional support, please contact Jungo Ltd.:

Web site: http://www.jungo.com
E-mail:
Sales: openrg@jungo.com
Support: rg_support@jungo.com

Jungo Headquarters
3031 Tisch Way
San Jose, CA 95128
U.S.A
Tel. +1 (408) 423 9540
+1 (877) 514 0537
Fax. +1 (877) 514 0538

EMEA
One Heathrow Blvd.
286 Bath Road
West Drayton
Middlesex UB7 0DQ
United Kingdom
Tel. +44 (20) 8476 8481
Fax. +44 (20) 8476 8482

Asia Pacific
P.O.Box 118-757 Taipei
Taipei City 10599
Taiwan (R.O.C)
Tel. +886 (9) 1938 2709

R&D Center
1 Hamachshev Street
Netanya 42504
Israel
Tel. +972 (74) 721 2121
Fax. +972 (74) 721 2122